Developer's Guide

Core Calypso Development Guide

Core Calypso Development Guide — Developer's Guide

29 April 2011

Document History

Revision date	Edition number	Comment
29 April 2011	First Edition	Rel 12.0
14 July 2011	First Ed. Rev. A.	Added Authentication Service info. See "Release 12.0 — 29 April 2011" on page 15

Contents

Tables

Figures

Changes to the Calypso Developer's Guide

This section notes changes made between releases.

Documentation Changes

This documentation makes use of Change Bars to indicate new or modified material from one publication to the next. Only the most current changes are marked. Change bars appear in the left-hand margin as shown to the left of this paragraph.

Updated — 07 February 2012

This is a documentation maintenance update. New information on several engines as well as corrections and updates are included. Note that new material marked by a Change Bar. The following is a list of significant changes

- Section 6.2.2, "BOSimpleRepoHandler," on page 44
- Section 6.3.1, "Creating a Custom Update Transfer," on page 46
- Section 6.11, "Customizing the Hedge Relationship Engine," on page 60
- Section 6.12, "Creating a Custom Hedge Relationship Config Handler," on page 61
- Section 6.13, "Customizing the Diary Engine," on page 61
- Section 6.14, "Customizing the Billing Engine," on page 61
- Section 9.1, "Balance Engine," on page 69
- Section 16.10, "Authentication Service," on page 166

Release 12.0 — 29 April 2011

Added Authentication Service information. See Section 16.10, "Authentication Service," on page 166.

1 Introduction

The developer's guide is intended for developers using the Calypso API to build custom components, or to customize and extend existing functionality. Extension capabilities are described through a comprehensive collection of examples.

Refer to the class library documentation on the support web site (also referred to as Javadoc) for a comprehensive description of all the classes and methods mentioned in the examples.

2 Getting Started

2.1 How to Compile and Execute Samples

2.1.1 Compiling

To compile the samples located in the **samples** package, go to $CALYPSO_HOME or to the root directory where the Calypso system resides, and type the following at the command line (this is a one-line command):

```
javac -d ./build -classpath "./jars/calypso.jar"
               -sourcepath ./src src/samples/<class name>.java
```

For example:

```
javac -d ./build -classpath "./jars/calypso.jar"
               -sourcepath ./src src/samples/LoadTrade.java
```

To compile the samples located in the **calypsox** package, rename the classes from **Sample<*ClassName*>** to **<*ClassName*>**. In $CALYPSO_HOME or in the root directory where the Calypso system resides, type the following at the command line (this is a one-line command):

```
javac -d ./build -classpath "./jars/calypso.jar"
-sourcepath ./src src/calypsox/<the fully qualified name of the class>.java
```

2.1.2 Executing

Prior to executing the samples, you must populate the database with sample data using the database scripts located in `samples/cookbook/sql/cookbook_sybase.sql`.

To execute any of the samples, type the following at the command line in `$CALYPSO_HOME/build` (this is a one-line command):

```
javac -d ./build -classpath "./jars/calypso.jar:./jars/calypso-core.jar:
         ./jars/calypso-logging.jar:
         ./jars/j2ee/j2ee-1.5.jar:<java_libdir>/tools.jar:
         ./jars/spring/spring.jar:./jars/jcommon.jar:
         ./jars/jide/lib/jide-grids.jar"
```

```
-sourcepath ./src src/samples/*.java
```

where *java_libdir* is the location of the Java lib directories, for example `/usr/lib/jvm/java-6-sun-1.6.0.26/lib/`.

To view a program's argument usage, simply run the program without arguments to display it's usage help.

In all the samples we will assume there is a Calypso environment name (myEnv), a Calypso user name (calypso_user), and password (myPassword). You can specify your Calypso settings using the User Env or System Env applications.

2.1.3 Sample Code

Throughout the document, there are references to sample code. You will find the code in `$CALYPSO_HOME` after uncompressing the Calypso release jar:

```
$CALYPSO_HOME\jars-src\jar-src\calibration-calypsox-src.jar
$CALYPSO_HOME\jar-src\core-calypsox-src.jar
```

Uncompress the jars to extract the sample code.

2.1.4 Classpath

Refer to `calypso.bat` or `calypso.sh` in `$CALYPSO_HOME/bin/` for the layout and necessary items in the Calypso classpath.

2.2 Creating a Custom Package

All calypso files are placed under the **`com.calypso`** package. You can place any extension to the system under the **`calypsox`** package. The system, by default, searches for new files in **`calypsox`**.

Alternatively, you can create a custom package name and place your extension there. The following sample illustrates the steps required to create your custom package.

Procedure to Create a Custom Package:

1. Create a directory for the custom package.

Note: If the custom package directory is not placed under $CALYPSO_HOME, you must modify your class path to include its location.

---

2. Create a class named **`calypsox.tk.util.CustomGetPackages`** that implements the interface **`com.calypso.tk.util.GetPackages`**.

   Implement the *getPackages()* method. The method will return a vector containing the names of all the packages you want to add.

   `com.calypso.tk.util.InstantiateUtil` invokes **`CustomGetPackages`**.

3. Restart all Calypso engines and applications.

Tip ⓘ Choosing the "Instantiate" logging category allows you to debug class instantiation.

Sample Code in `calypsox/tk/util/`

`CustomGetPackages.java`

In this sample, we created a package called **`samples.cookbook`**

```
package calypsox.tk.util;

import com.calypso.tk.util.GetPackages;
import java.util.*;
import java.lang.String;

/**
 * A utility interface to search for the particular package.
 * That class will be used to get the Packages in Calypso.
 */
public class CustomGetPackages implements GetPackages  {
    public Vector getPackages() {
        Vector v = new Vector();
        v.addElement("samples.cookbook");

        return v;
    }
}
```

2.3 How to Create Custom Code

Custom code must reside under the `$CALYPSO_HOME/custom` directory with the proper package name, either `calypsox` for example `custom/calypsox/tk/service/CustomDSStartUp`, or your own custom package name.

The `$CALYPSO_HOME/src/calypsox` directory is reserved for sample custom code provided by Calypso.

Details are provided in `$CALYPSO_HOME/custom/ReadMe.txt`.

2.4 Creating a Client Application

2.4.1 Connecting to Data Server and Event Server

When you create a client application that requires access to Calypso data such as domain data (currency code list, reference index list, etc.) and persistent data (trades, curves, positions, etc.), the client application must establish a connection to the Data Server. Furthermore, if a client application must publish or subscribe to Calypso events, it must also establish a connection to the Event Server.

Once the a Data Server connection is established, the application can access Calypso data using the data services (see Section 3, "Data Services," on page 23).

The example below illustrates how to create a connection to the Data Server and the Event Server and how to disconnect before the program exits.

Creating Connections to the Data Server and Event Server:

1. Use **`com.calypso.tk.util.ConnectionUtil::connect`** to create a Data Server connection. This method returns a **`com.calypso.tk.service.DSConnection`** object.
2. Use *com.calypso.tk.event.ESStarter::startConnection* to create an Event Server connection. This method returns a **`com.calypso.tk.event.PSConnection`** object.
3. Use *PSConnection::stop* to disconnect from the Event Server.
4. Use *DSConnection::disconnect* to disconnect from the Data Server.

Tips

- The *connect* method in ConnectionUtil is overloaded and allows different arguments and options to create a connection to the Data Server. Refer to **`com.calypso.util.ConnectionUtil`** for details. If you do not want to use ConnectionUtil, you may directly use the *connect* method in DSConnection.
- The *startConnection* method in ESStarter is overloaded to allow using different arguments and options to create a connection to the Event Server. Refer to **`com.calypso.event.ESStarter`** for details.
- Once connected, the DSConnection object contains all of the settings from your Calypso environment. You can access those settings using the different *get* methods in DSConnection. The static method *DSConnection.getDefault* returns the connection created, making it unnecessary to pass or store the DSConnection object in your code.
- The static methods *PSConnection.setCurrent* and *getCurrent* allow you to store the PSConnection created in your client application, thus permitting your client application to access it from anywhere within your code.
- Socket connection to DataServer has been removed and an RMI call will be retried in case of network issues.
- Properties that control the retry behaviour are
  - TIMEOUT_RECONNECT in msec that defines the time between retries
  - MAX_NUMBER_OF_RECONNECTION defines the number of times an RMI failed due to network related issues will be retried.

Sample Code

```
public class PSSample {

  private static final String LOGCAT = "PSSample";
  static public void main(String args[]) throws Exception {

    Log.system(LOGCAT, "Connecting to ds...");

      DSConnection ds = ConnectionUtil.connect(args, "PSSample");


         // create a subscriber
         MySubscriber eventListener = new MySubscriber();

          // events we are interested in
         Class[] subscriptionList = new Class[] {
                                            PSEventTrade.class,
                                            PSEventMessage.class,
                                            PSEventTime.class,
                                            };

         Log.system(LOGCAT, "Connecting to jms bus...");
         PSConnection ps =
                   ESStarter.startConnection(eventListener, subscriptionList);

         Log.system(LOGCAT, "Waiting before publishing 5 events...");
         Thread.sleep(3000);

         for(int i=0; i < 5; i++) {
              // build an event
              PSEventTime eventTime = new PSEventTime();
              eventTime.setTime(System.currentTimeMillis());
              eventTime.setComment("PSSample generated event "+i);

              Log.system(LOGCAT, "Publishing event "+eventTime+"...");
              ps.publish(eventTime);
         }
      }

      /**
      * MySubscriber class will be the call back point for
      * all incoming events.  newEvent will be invoked when
      * an event matching the subscription list is recieved.
      */
      private static class MySubscriber implements PSSubscriber {

         public void newEvent(PSEvent event) {
           Log.system(LOGCAT, "Recieved event"
                              +", id="+event.getId()
                              +", type="+event.getEventType()
                              +", event="+event
                              );
           }

           public void onDisconnect() {
                Log.system(LOGCAT, "Event bus has disconnected!");
```

```
            }
        }
}
```

2.4.2 Handling a Lost Connection to the Data Server

If the Data Server connection is lost, your application may require a notification and might subsequently attempt to reconnect to the Data Server. The Data Server provides the following mechanisms to auto-reconnect if a connection is lost:

- *DSConnection.setAutoReconnect()* — Performs automatic reconnections to the Data Server.
- ConnectionListener — Monitors the connection to the Data Server.

2.4.2.1 Using DSConnection.setAutoReconnect()

Note: This method has no effect and is slated for removal.

Set *setAutoReconnect()* to true and set the following parameters as applicable:

- RETRY = Number auto-reconnect attempts
- TIMEOUT_RECONNECT = The interval in milliseconds between attempts to reconnect to the DataServer.

2.4.2.2 Implementing ConnectionListener

To implement ConnectionListener:

1. Implement **`com.calypso.tk.service.ConnectionListener`** in the client application.
2. Call *DSConnection::addListener(…)* to register the client application with the Data Server.

Sample Code in `samples/`

`UseDSListener.java`

Illustrates how to extend the client application from `TestCon.java` to auto-reconnect to the Data Server.

2.4.3 Creating Custom Initialization Code

The Data Server provides two ways to invoke custom code:

- DSStartUp — Invokes initialization code when the Data Server is started.
- DSConnectionInit — Invokes initialization code on the client each time a DSConnection is created.

2.4.3.1 Using DSStartUp

DSStartUp invokes your initialization code when the Data Server is started, after the SQL init, but before the start of the RMI Services (i.e. before any client can connect). One usage of this mechanism is to allow the Data Server to preload frequently accessed data for the use of the client application(s). For example, you can preload certain books in the system. This will prevent the Data Server from having to retrieve the trades for those books individually.

Create a class named **`tk.service.CustomDSStartUp`** that implements the interface **`com.calypso.tk.service.DSStartUp`**, and then implement your code in the *onStartUp()* method.

`com.calypso.tk.service.DataServer` invokes **`CustomDSStartUp`**

Sample Code in `calypsox/tk/service/`

`CustomDSStartup.java`

This sample loads PricingEnv and TradeFilters set by properties PRICING_ENV_STARTUP and TRADE_FILTER_STARTUP.

2.4.3.2 Using DSConnectionInit

DSConnectionInit allows you to specify a custom initialization class for client applications which is called after a DSConnection is created and started.

Create a class named **`tk.service.CustomDSConnectionInit`** that implements the **`com.calypso.tk.service.DSConnectionInit`** interface.

`com.calypso.tk.service.DSConnection` invokes **`DSConnectionInit`**.

Sample Code in `calypsox/tk/service/`

`CustomDSConnectionInit.java`

2.4.4 Customizing the RMI Socket Factory

The default implementation allows you to bind to a specific interface and to also configure SSL for RMI communication.

- BIND_TO_SPECIFIC_INTERFACE=true — permits the sockets to bind to a specific interface on the host as against the default. Requires the use of the the java.rmi.server.hostname JVM property.
- BIND_TO_SPECIFIC_INTERFACE=false — bind to all the interfaces.

To enable SSL for RMI calls:

1. Set the property USE_SSL=true
2. Generate a key/trust store

```
keytool -genkey -keystore keystore -keyalg RSA
```

3. For a Server such as the Data Server, add the VM arguments:

   (Assuming that the keystore and truststore are both located at: `/home/some_user_name/keystore`)

```
-Xmx512m -XX:MaxPermSize=128m
-Djavax.net.ssl.keyStore=/home/some_user_name/keystore
-Djavax.net.ssl.keyStorePassword=calypso
-Djavax.net.ssl.trustStore=/home/ravi_somepalli/keystore
-Djavax.net.ssl.trustStorePassword=calypso
```

4. For a client, such as MainEntry, add the VM arguments:

```
-Xmx384m -XX:MaxPermSize=128m
-Djavax.net.ssl.trustStore=/home/some_user_name/keystore
-Djavax.net.ssl.trustStorePassword=calypso
```

2.4.5 Creating a User Startup Routine

A startup routine permits you to implement your own class to perform tasks such as pre-loading C++ libraries, for example, which would be useful for Scheduled Task EOD_VALUATION.

Create a class named **`apps.main.UserStartup`**.

`com.calypso.apps.startup.AppStarter` invokes **`UserStartup`**.

3 Data Services

3.1 Using the Data Server

The Data Server is a single point of access for all Calypso data. No client application should ever access the database directly. Once you have a connection to the Data Server as described in Section 2.4.1, "Connecting to Data Server and Event Server," on page 18, the Data Server is accessed through a set of remote services located under the **`com.calypso.tk.service`** package, each of which is responsible for a different group of data:

- RemoteMarketData — Handles pricing information (e.g., interest rate curves).
- RemoteReferenceData — Handles static data (e.g., counterparty definitions).
- RemoteProduct — Handles financial instrument definitions (e.g., futures contracts).
- RemoteTrade — Handles trade information.
- RemoteAccess — Handles access permission and system security data.
- RemoteAccounting — Handles accounting rules data.
- RemoteBackOffice — Handles back office-specific data.

The Calypso online documentation provides detailed descriptions of the objects handled by each service under **`com.calypso.tk.service`**.

The Data Server in turn accesses the database or the data stored in cache. Caches maintained by the Data Server are configured and administrated from the Calypso Administrator window. However, for data that

is frequently accessed such as reference data, it is more efficient to locally cache those data in the client. The BOCache and LocalCache classes are provided for that purpose, respectively. It is their responsibility to access the Data Server.

Hence, to access Calypso data:

1. Check whether BOCache or LocalCache handle those data and use BOCache and LocalCache.
2. If BOCache or LocalCache do not contain the data, then use the appropriate remote service.

3.2 Using a Local Cache

LocalCache allows client applications to maintain client caches for a number of static data including Currency Indexes, Rate Indexes, Rate Index Defaults, Currencies, Domains, and FX resets, thereby enhancing the performance.

When data retrieval from LocalCache is attempted, if the data is not available in the cache, LocalCache then retrieves the data from the data server and updates the cache, thus making the data available for subsequent requests.

For caching other types of static data, use Section 4, "Event Services," on page 34.

Tips

- Using an instance of **`tk.util.CacheConnection`** allows you to automatically maintain cache consistency. This class subscribes to the the events PSEventDomainChange, PSEventQuoteRemoved, PSEventQuote, and PSEventCreditRating to obtain updates for cached data, and also updates cached data according to the timer specified in the Admin window.

```
CacheConnection cacheConnection = new CacheConnection(DSConnection.getDefault());
```

- Use **`tk.service.RemoteReferenceData`** to retrieve static data not handled by LocalCache or BOCache.
- Do not use LocalCache for code that will be executed within the Data Server.

Warning: When the client application must modify domain values (e.g., for display purposes) you must clone the data using *cloneDomainValues()*, and then modify the cloned data. **Never directly modify the returned list directly**. Doing so changes the master list in LocalCache and causes data inconsistency.

See also:

- **`com.calypso.apps.util.AppUtil`** class for helpful object loading methods.
- Refer to **`com.calypso.tk.service.LocalCache`** for available methods and details.

Sample Code in `samples/`

`UseLocalCache.java`

Illustrates how to use LocalCache in a client application.

The code creates a connection to the Data Server and calls the appropriate static method on LocalCache to retrieve the values of the "Principal Structure" domain without the Mortgage value.

3.3 Using BOCache

BOCache allows client applications to maintain client caches for a number of static data including Accounts, Books, Contacts, Legal Entities, Settlement and Delivery Instructions, Exchange Traded Products, and Netting Configurations. BOCache also maintains client caches for quotes.

When data retrieval from BOCache is attempted, if the data is not available in the client cache, BOCache retrieves the data from the data server and updates the client cache, thereby making the data available for subsequent requests.

For example, for a loading a trade, you would use BOCache to load the relevant TradeFilter to be passed to the *getRemoteTrade().getTrades(TradeFilter, Jdatetime)* method:

```
TradeFilter tf = BOCache.getTradeFilter(ds, "MyTradeFilter");
```

For caching other types of static data, refer to Section 4, "Event Services," on page 34.

Tips

- Using an instance of **`tk.util.CacheConnection`** will allow you to automatically maintain cache consistency. This class subscribes to the the events PSEventDomainChange, PSEventQuoteRemoved, PSEventQuote, and PSEventCreditRating to obtain updates for cached data, and also updates cached data according to the timer specified in the Admin window.

```
CacheConnection cacheConnection = new CacheConnection(DSConnection.getDefault());
```

`samples/TestMultiThreadQuotes.java`

Illustrates how to obtain refreshed quotes from BOCache.

- Use **`tk.service.RemoteReferenceData`** to retrieve static data not handled by LocalCache or BOCache. See also, Extending BOCache, below.
- Do not use BOCache for code that will be executed within the Data Server.

Extending BOCache

If the data you want to access is not handled by BOCache or LocalCache, and you want to cache that locally, you can extend BOCache by providing an implementation of **`CustomClientCache`** in **`tk.bo.CustomClientCacheImpl`**.

BOCache invokes **`CustomClientCacheImpl`**.

You must also publish a PSEventDomainChange for modified data so that the BOCache publishes updates to **`CustomClientCacheImpl`**. PSEventDomainChange contains a set of static integers used to identify what data has changed (legal entity, book, etc.). You must add your own set of static integers to identify changes specific to **`CustomClientCacheImpl`**.

Create a class named **`tk.event.PSEventDomainChangeCustom`** that extends **`com.calypso.tk.event.PSEventDomainChange`** and defines integers for each custom data as in the example shown below.

```
final static public int MY_DATA1 = ID_MAX+1;
final static public int MY_DATA2 = ID_MAX+2;
```

Publish a PSEventDomainChangeCustom for each custom data.

Sample Code in `calypsox/tk/`

- `bo/CustomClientCacheImpl.java`
- `event/PSEventDomainChangeCustom.java`

See also:

- **`com.calypso.apps.util.AppUtil`**
  A class with helpful object loading methods.
- **`com.calypso.tk.bo.BOCacheOCache`**
  Shows all available methods and details for BOCache.

3.4 Using a Remote Service

The following example demonstrates how to obtain a trade and product from the Data Server and how to subsequently save them as a new trade and product.

Using a remote service:

1. Create a connection to the Data Server.
2. Obtain the appropriate remote object from the Data Server connection.
3. Use the appropriate method in the remote object to retrieve or save the desired data.

Tip

ⓘ For objects having a unique ID, the ID is assigned by the Data Server the first time the object is saved. The save methods in the remote services for those objects return the assigned ID after a successful save. It is important to set the object's ID to the returned ID after a successful the save. Otherwise, the object retains its initial null ID and any subsequent saves will result in a new copy of the object being created.

Sample Code in `samples/cookbook/`

- `UseDataServer.java`

To output all of the IDs of the trades in a TradeFilter, call *getTrades(TradeFilter, JDatetime)* on RemoteTrade to return all the trades associated with the TradeFilter and whose trade date is before the JDatetime.

```
DSConnection ds = null;
try {
      ds = ConnectionUtil.connect(args, "UseRemoteBO");
}
   catch (ConnectException e) {
      Log.error(Log.CALYPSOX, e);
      System.out.println("ERROR: Connection to data server failed.");
      System.exit(-1);
  }

JDatetime now = new JDatetime();
try {
      TradeArray v = ds.getRemoteTrade().getTrades(tradeFilter, now);

      for(int i=0;i<v.size();i++) {
      Trade trade = (Trade) v.elementAt(i);
      System.out.println("Trade : " + trade.getId());
      }
}
   catch (Exception exc) {}
```

To retrieve a price quote for a product, use the RemoteProduct interface to load the product and the RemoteMarketData interface to load the quote. The *getProduct(int)* method of RemoteProduct returns the product for a given Product ID. The *getQuoteValue(QuoteValue)* method of RemoteMarketData returns a QuoteValue object that contains the quote value and type.

```
DSConnection ds = null;
try {
      ds = ConnectionUtil.connect(args, "UseRemoteBO");
}
   catch (ConnectException e) {
      Log.error(Log.CALYPSOX, e);
      System.out.println("ERROR: Connection to data server failed.");
      System.exit(-1);
  }

Product product = null;

try {
      product = ds.getRemoteProduct().getProduct(inputProductId);
}
      catch (Exception exc) {}

JDatetime now = new JDatetime();
JDate quoteDate = now.getJDate(TimeZone.getDefault());

// Initialize a QuoteValue object
// use quote type NONE - will be set by getQuoteValue method
```



```
QuoteValue q = new QuoteValue(quoteSetName, product.getQuoteName(),
      quoteDate, QuoteValue.NONE, 0, 0);
try {
      q = ds.getRemoteMarketData().getQuoteValue(q);
}
   catch (Exception exc) {}
```

Sample Code in samples/

QuoteLoaderSample.java
(complete example)

3.5 Extending the Data Server

When a custom object that requires persistence is added to the system, the Data Server must be extended to save, load, and cache the object. If necessary, the extension should also support event publishing when the object is first saved or when it is modified.

The Calypso API provides two ways to extend the Data Server:

- The DSTransactionHandler
- Creating a custom remote service

The DSTransactionHandler is intended to accommodate a small number of custom objects. As illustrated in the next section, the user must create a DSTransactionHandler and other related classes for each custom object which is time consuming when a large number of custom classes is needed. However, for a small number custom objects, the DSTransactionHandler is an efficient mechanism to extend the Data Server.

Note: The extension of the Data Server is only required for custom objects that **do not** extend from an existing persistent Calypso object. The system has other mechanisms to support children of persistent Calypso objects and these dedicated mechanisms should be used instead. For example adding a new product or market data **does not** require extending the Data Server.

The following sections demonstrate how to extend the Data Server using both methods. In the examples, we will extend the Data Server to handle the persistency, caching, and event publishing of a custom "equity basket."

3.5.1 Persistence and Caching

3.5.1.1 Persistence

Handling persistence for the equity basket class requires you to write an SQL class and create a database table. These steps are required regardless of which extension method is used.

Sample Code in samples/cookbook/

tk/product/EquityBasket.java

EquityBasket sample. **The EquityBasket class example is only a sample, it is not suitable for production**.

`tk/product/sql/EquityBasketSQL.java`

EquityBasketSQL sample.

`sql/cookbook_sybase.sql`

Database scripts examples.

Tips

SQL Error Handling —The SQL methods should throw a PersistenceException so that the user of the client application receives an error message if an SQL error occurs. The remote methods that call your SQL methods will catch PersistenceException and then create and throw a RemoteException containing that PersistenceException.

For example:

```
void sqlfoo() throws PersistenceException {
   try {
      ...
   }
   catch (SQLException e) {throw new PersistenceException(e.getMessage()):}
}
The typical RMI method that calls an SQL method will handle the error as follows:
void rmiCall() throws RemoteException {
   try {
      sqlfoo();
   }
   catch(Exception e) {throw new RemoteException(e.getMessage());}
}
```

JResultSet — Calypso recommends using **`com.calypso.core.sql.JResultSet`** to work with query results when you retrieve multiple records from the database. JResultSet is a wrapper class (around a ResultSet object) that adds the methods *getJDate()* and *getJDatetime()* to return a date or datetime from any cell in a table of query results.

The example below shows how one might use a JResultSet to compile a Vector of trades with their Trade ID and Trade Date/Settle Date stamps. In this example, all Trade Date and Settle Date stamps are expressed in the system's local time zone:

```
static public Vector getAllTradeTimestamps() {
   Vector tradeVector = new Vector();
   Connection con = ioSQL.newConnection();
   Statement stmt = null;
   try {
      stmt = ioSQL.newStatement(con);
      JResultSet rs = new JResultSet(stmt.executeQuery("SELECT \
         trade_id, trade_date_time, settlement_date FROM trade,book \
         where book.book_name='TRADINGC' and trade.book_id=book.book_id"));
      int j;
      while(rs.next()) {
         j=1;
```

```
         Trade trade = new Trade();
         tradeVector.addElement(trade);
         trade.setId(rs.getInt(j++));
         trade.setTradeDate(rs.getJDatetime(j++));
         trade.setSettleDate(rs.getJDate(j++));
      }
      rs.close();
   }
   catch( Exception e ) { display(e); }
   finally {
      try {ioSQL.close();}
      catch (Exception e) {}
      ioSQL.releaseConnection(con); }
   return tradeVector;
}
```

- **Dates** — Calypso distinguishes between timestamps stored in the database and those displayed to users. Saved timestamps are expressed in the designated reference time zone, while timestamps displayed to and set by users are expressed in the local time zone, which is the time zone of the user's workstation. By default, the reference time zone for stored timestamps is GMT.

  When using SQL queries to retrieve information from the Calypso database, keep in mind that all dates and times are expressed in the system's designated reference time zone. Thus any dates and times in your WHERE clauses must be expressed in the reference time zone, and you must convert all dates/times in your query results to the desired local time zone.

  Calypso provides a set of methods to convert dates and times from system reference timezone to local timezone. These methods are contained in the **`com.calypso.tk.core.Util`** class. To convert a local date to a String for use in a WHERE clause, use the method *date2SQLString()* or *datetime2SQLString()*.

  Alternatively, you can use *Util.ReferenceTZ2Local(Date)* to convert a date from the reference time zone to the local time zone. For details, see the online class documentation for **`com.calypso.tk.core.Util`**.

- **Commit and Rollback** — Whenever you have a commit in an SQL file, ensure that you also have a rollback as shown in the example below.

```
Void save(myObject){
try{
   Connection con=ioSQL.newConnection();
   save(myObject, con);
   commit(con);
   Update the Cache or any hash Only after commit
}
catch(PersistenceException e){
   rollback(Con);
   Log.error(e,e)
}
finally{ releaseConnection(con)}
}
```

Note: Do not put the commit inside *save(myObject, con)*.

3.5.1.2 Caching

It is logical to cache frequently accessed objects so that the Data Server need not repeatedly retrieve them from the database. Refer to Section 18, "Cache Framework," on page 183 for details.

3.5.2 Using DSTransactionHandler and DSTransactionInput

The mechanism in the Data Server that accommodates custom objects involves the DSTransactionHandler and DSTransactionInput classes in the **`com.calypso.tk.service`** package. Custom DSTransactionHandler and DSTransactionInput classes for the object must be added to the system. An SQL class that contains the persistency code for the object is also required. A client application that must access the custom object must also create an instance of the custom DSTransactionInput class and then pass it to the Data Server. The Data Server will internally invoke the custom DSTransactionHandler which in turn uses the SQL class of the object to perform persistence and caching.

Overview of Steps

- Step 1 — Create an SQL class for the object as described in Section 3.5.1
- Step 2 — Create a DSTransactionInput class
- Step 3 — Create a DSTransactionHandler class

Step 1 — Create an SQL class for the object

See Section 3.5.1, "Persistence and Caching," on page 28.

Step 2 — Create a Custom DSTransactionInput

Create a class named **`tk.service.<object name>TransactionInput`** that extends **`com.calypso.tk.service.DSTransactionInput`**.

The *getHandler()* method specifies the name of the associated handler class, and the member variable transactionType specifies the task that the client program wants the Data Server to perform.

Sample Code in `samples/cookbook/tk/service/`

`EquityBasketTransactionInput.java`

Step 3 — Create a Custom DSTransactionHandler

Create a class named **`tk.service.<object name>TransactionHandler`** that extends **`com.calypso.tk.service.DSTransactionHandler`**.

The handler class invokes the object's SQL class to perform the necessary persistence tasks. It also publishes an event after the save is completed.

This class is invoked from **`com.calypso.tk.service.AccessServerImpl`**.

In this sample, we are treating the event as a persistent event. As such, the handler class saves the event to the database.

Note: PSEventEquityBasket is a new event type. See Section 4.3, "Creating a Custom Event," on page 35 for details on creating new event types.

Tips

- For the Data Server to properly handle a custom persistent object, the class must implement the Serializable interface.
- When publishing a persistent event inside the Transaction Handler, the event *must* be published and saved within the same database transaction that the persistent object is handled, to ensure transactional atomicity. This way the event will not be published if any error is encountered while saving the object and the event.

Sample Code in `samples/cookbook/`

- `tk/service/EquityBasketTransactionHandler.java`
- `UseEquityBasket.java`

  A sample client application that illustrates how to use DSTransactionInput to access an EquityBasket object from the Data Server.

3.5.3 Creating a Custom Remote Service

1. Implement the business interface
   a. It must not implement java.rmi.Remote
   b. It should not throw RemoteException, it should instead throw application Exceptions
2. Develop the Service that implements the businessInterface

- Step 4 — Register the custom server with the Data Server.

Step 1 — Create an SQL class for the object

See Section 3.5.1, "Persistence and Caching," on page 28.

Step 2 — Creating a Custom Service

Create a class named **`tk.service.Remote<name>`**. The interface defines all of the methods that the class provides.

Our sample will create a custom service to add two numbers.

Sample Code

Our short sample (`RemoteTestService`, above) provides a single method, **sum**, which takes two integers as arguments.

The sample code provided in the Calypso release jar's `samples` directory contains methods to handle the EquityBasket class examples which are also in the `samples` directory. A similar set of methods can be added to the interface for any new custom object.

Sampled Code in `samples/cookbook/tk/service/`

- `RemoteCustomData.java`

Step 3 — Creating the Custom Service.

Create a class named **`tk.service.<service_name>`** that implements the custom remote service interface created in Step 2 — Creating a Custom Service. In our example, the service name is **TestService**.

The service class is responsible for:

- Performing the requested persistence task by calling the appropriate method in the object's SQL class.
- Publishing (and saving, if necessary) events triggered by the saving and removing of an object.

Verifying that Your Service is Running

Use the following short program to view the services operating on the system:

3.6 Read-Only Data Servers

A read-only Data Server provides a mechanism to off-load work from the active Data Server for read operations. High volume read operations, such as generating reports, or other similar tasks could have a negative impact on Data Server performance. Using a read-only Data Server allows access to data, while not impeding write operations. Read-only Data Servers do not operate as standalone servers, they require the presence of an Active (i.e., a read/write) Data Server.

Note: Read-only Data Servers do not update the cache. The cache is updated by listening for events sent by the active Data Server and then reloading the information from the database as needed. This requires database connectivity for the read-only Data Server.

The call to use events to update the cache for a read-only Data Server rather than RMI services, is controlled by *setUseCacheSubscriber()*. If the DS_READ_ONLY property is true, indicating that the Data Server is running in read-only mode, then *setUseCacheSubscriber()* also returns true. Note that the DS_READ_ONLY property must be set prior to starting the Data Server.

Refer to the *Calypso System Guide* for complete information on setting up the Data Server in read-only mode.

How to Customize SQL Query Testing

You can customize the testing of SQL Queries to see if they are allowed on a read-only Data Server.

Create a class named **`tk.core.sql.CustomReadOnlySQLTest`** that implements the interface **`com.calypso.tk.core.sql.ReadOnlySQLTest`**.

This class will be invoked from **`com.calypso.tk.core.sql.CalypsoStatement`**.

4 Event Services

Calypso leverages a standard JMS bus to propagate events throughout the platform to listeners. The implementation leverages a single JMS topic for all business related events. The ESStarter interface is the starting point for all access to the event bus.

4.1 Subscribing to and Publishing Events

Listeners and publishers must be created through the ESStarter class and must leverage the PSConnection and PSSubscriber interfaces to interact with the underlying JMS connection as publishers and subscribers. Refer to Section 2.4.1, "Connecting to Data Server and Event Server," on page 18 for a high-level overview of how to leverage the ESStarter interface and refer to the ESStarter java API for details. PSEvent (`com.calypso.tk.event.PSEvent`) is the base class for all event types. Each derived class represents a specific type of event. For example, PSEventTrade models the events published when a user performs an operation on a trade.

Note All events are named PSEvent<*event type*> and are located under `com.calypso.tk.event`.

4.1.1 Event Types and Filtering

Events published to the bus are differentiated by their type. This type is represented by different classes implementing each event container. All classes extend the PSEvent class, each derived class represents a different specific event type.

Subscribers built via the ESStarter can register for specific event types. This event list must be defined at the time of building the PSConnection and can cannot be altered once defined. Any changes to the list of events requires the PSConnection to be rebuilt.

4.1.2 Publishing Events

Once you have a PSConnection, you can:

- Publish non-persistent events using the *publish()* method on `com.calypso.tk.event.PSConnection`.
- Publish persistent events using the *saveAndPublish()* method on `com.calypso.tk.service.RemoteTrade`. This method saves and publishes an event, or a list of events, as a single transaction so that events are published only if the database save operation succeeds.

4.1.3 Subscribing to Events

Subscription to events requires the creation of a listener which implements the PSSubscriber interface. This class will implement the basic

event handling logic desired. An instance of this class will then be passed to the ESStarter.startConnection method.

The subscriber class can do the following:

- Handle events received from the real-time event bus.
- React to a disconnect from the bus. The disconnect call back is provided in order to allow implementors to react to this event and attempt to rebuild their PSConnection.

Note: Its important to note that management of the PSConnection should not be done within the event handling thread (in the PSSubscriber) and should be delegated to a separate thread allowing the subscriber to continue its lifecycle to either consume events or complete the tear down of the connection.

4.2 Handling Lost Connections to the Event Server

When writing a client program you may want to be notified and subsequently attempt a reconnection if the connection to the Event Server is lost. The following sample illustrates how to create a PSSubscriber that will automatically reconnect to the Event Server.

Sample Code in `samples/cookbook/`

`SmartSubscriber.java`

4.3 Creating a Custom Event

New event types can be added to the system to accommodate the addition of custom objects. For example, when we add the EquityBasket in the Data Server extension example (Section 3.5, "Extending the Data Server," on page 28), we must also add a new event type PSEventEquityBasket to notify the system when an EquityBasket object is created or modified.

Overview of Steps

- Step 1 — Create an event class that extends PSEvent
- Step 2 — Register the new event class

Step 1 — Creating a Custom PSEvent

Create a class named **`tk.event.PSEvent<event type>`** that extends **`com.calypso.tk.event.PSEvent`**.

Note: Since event objects are serialized for communication, each event class must have its own unique serialVersionUID, and must be included in the class definition. Obtain the ID by running `%JAVA_HOME%\bin\serialver.exe` on Windows platforms or `$JAVA_HOME/bin/serialver` on Unix platforms,

The following methods of PSEvent must be implemented:

- *toString()* — Returns a canonical string representing the event object. The result includes the class name and identification number. However, the actual event subclasses will often want to redefine this method to produce a more descriptive description of the event.
- *getEventType()* — Returns the event class name of the event. However, the actual event subclasses will often have an event type that is more specific than the class name. Many subclasses redefine this method to return a more precise event type designation.

Sample Code in `samples/cookbook/tk/event/`

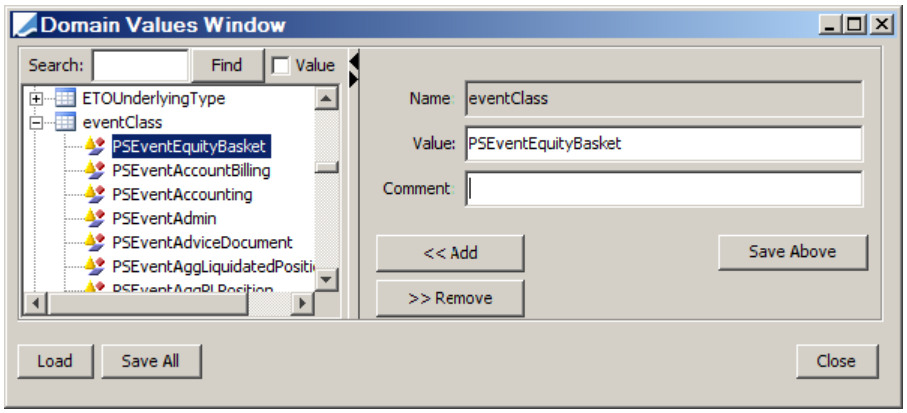
`PSEventEquityBasket.java`

Step 2 — Registering the new Event Class

Add the new event class to the eventClass domain.

1. Open the *Domain Values Window* (**Configuration-> System-> Domain Values**).
2. Expand the **eventClass** item.
3. Enter PSEventEquityBasket in **Value**.
4. Click **<<Add** to add the new class.
5. Click **Save All** to save your changes.

Figure 4-1: Registering a New Event Class



5 Core

5.1 Creating a Custom Daycount

Create a class named `tk.core.CustomDayCountCalculator` which implements the interface `com.calypso.tk.core.DayCountCalculator`.

This class is invoked from **com.calypso.tk.core.DayCount** so that once it is compiled, the new Daycount is made available in the system.

Note: The maximum length of a daycount is nine characters.

Sample Code in calypsox/tk/core/

CustomDayCountCalculator.java

5.2 How to Create a Custom Tenor

Create a class named **tk.core.CustomTenorCalculator** which implements the interface **com.calypso.tk.core.TenorCalculator**.

This class is invoked from **com.calypso.tk.core.Tenor** so that once it is compiled, the new Tenor is available in the system.

Sample Code in calypsox/tk/core/

CustomTenorCalculator.java

5.3 How to Create a Custom DateRule

Create a class named **tk.core.DateGenerator<*custom_name*>** that implements the interface **com.calypso.tk.core.DateGenerator**.

This class is invoked from **com.calypso.tk.core.DateRule** so that once it is compiled, the new DateRule is available in the system.

5.4 How to Create a Custom Frequency

Create a class named **tk.core.CustomFrequencyCalculator** that implements the interface **com.calypso.tk.core.FrequencyCalculator**.

This class is invoked from **com.calypso.tk.core.Frequency** so that once it is compiled, the new Frequency is available in the system.

Sample Code in calypsox/tk/core/

CustomFrequencyCalculator.java

5.5 How to Chain Exceptions

All calypso exceptions are derived from the class **com.calypso.tk.core.CalypsoException**. Exceptions can be chained using *setNext()* and *getNext()*, which allows the system to throw multiple exceptions.

In all SQL transactions under **tk**, the code catches all Throwable rather that just Exception.

5.6 How to Create a Comparator Class for Sorting Objects

Create a class named **tk.util.<name>Comparator** that implements **java.util.Comparator**.

You can access the comparator class using **ComparatorFactory.getCustomComparator("<name>")**.

You can create a custom comparator class to order aggregation groups in ScenarioAnalysisViewer.

6 Engines

Engines in Calypso are special type of event listeners. These specialized listeners manage the guaranteed delivery of persisted events, the event bus reconnect features as well as allow for multi-threaded processing of events.

6.1 Creating a Custom Engine

An engine is implemented by sub-classing the Engine abstract base class.

Overview of Steps

- Step 1 — Create a custom Engine class
- Step 2 — Register the new engine
- Step 3 — For subscribing to persistent events only
- Step 4 — Start the new engine

Step 1 — Creating a Custom Engine Class

Create a class that extends **com.calypso.engine.Engine**. There is no restriction regarding the name of the engine or the location of the class.

The constructor for the Engine class is as follows:

- `Engine(DSConnection dsCon, String hostname, int port)`

  Where:

  - **dsCon** — a live connection to the Data Server.
  - hostname/port — Beginning in V12.0, the management of the Event server connection is no longer the responsibility of each specific engine class and therefore the remaining parameters can be ignored and simply passed to the parent constructor.

The following methods of Engine must be implemented:

- *getEngineName()* — Returns the engine name. The engine name must be unique from other engines in the system. Engine names are specified in the **engineName** domain.

- *getClasses()* is now a final method. The method to implement is *getNonPersistentClasses()* which subscribes to non-persistent events. See "Step 3 — For Subscribing to Persistent Events Only" on page 41 for the procedure for setting persistent subscriptions.
- *process(PSEvent event)* — These event types are automatically subscribed to by the engine based on the configuration stored in the database. It is therefore not necessary to manually subscribe to these event types. Subscribing to persistent event types that are not configured in the event config can result in lost events and should *never* be done. Returns True if the event was successfully processed, or False, otherwise. The *process()* method must call *RemoteTrade.eventProcessed()* mark the event as processed.

Sample Code in `samples/`

`SampleEngine.java`

Note: The sample program works Transfer Events. This is in addition to registering the engine as described below.

Tips

- **Add publishing to an engine** — Engines often publish events, as well as subscribe to events. To publish within an engine you would simply add the following to the engine *getPSConnection().publish()*.
- **Suspend/Resume** — Engine that are subscribed to PSEventAdmin and operating in real time (listening for events), as opposed to Batch mode (not listening to events), will respond to suspend/resume events. MAX_QUEUE_SIZE controls the number of trades held in memory before an engine switches to Batch Mode.
- **Setting engine parameters** — You can define parameters in your engine that can be saved and viewed in the **Administrator Window -> Engine Thread** tab. Use the following methods on **`com.calypso.tk.service.RemoteAccess`** to load and save engine parameters:
  - saveEngineParams(EngineParamOptionsConfig)
  - getEngineParams()
  - getEngineParam(String engine, String param)

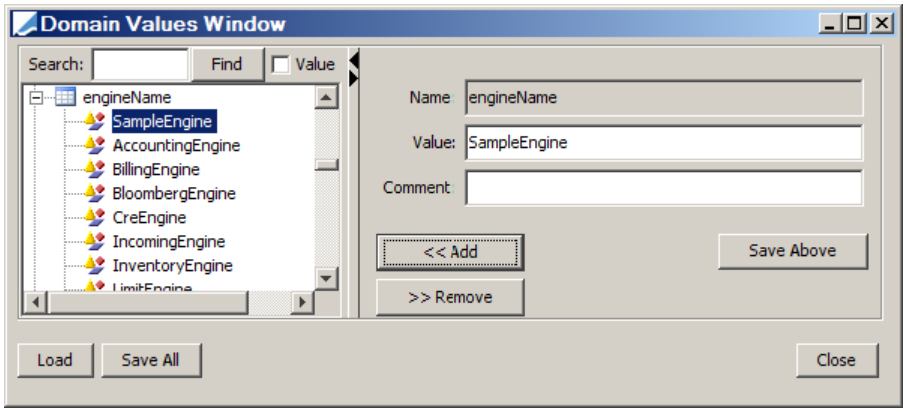
Step 2 — Registering the New Engine

Use the following procedure to register a new engine:

1. Add the engine name to the **engineName** domain.

a. Open the *Domain Values Window* (**Configuration-> System-> Domain Values**).

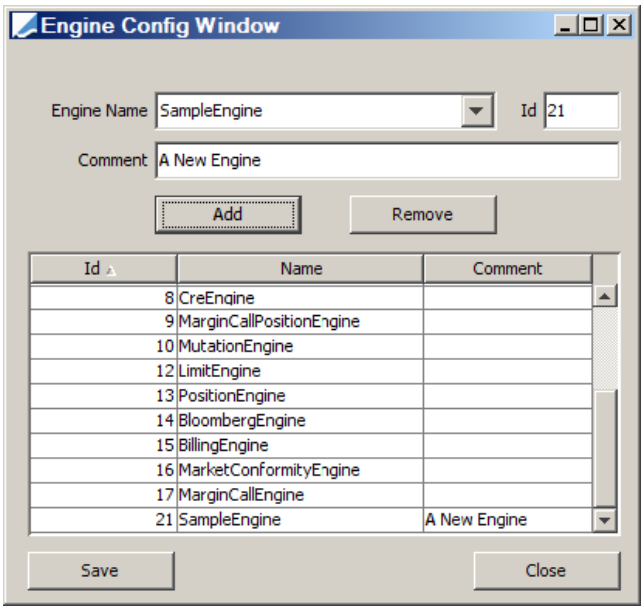
Figure 6-1: Registering a New Engine



b. Expand the **engineName** item.
c. Enter the engine name in **Value**.
d. Click **<<Add** to add the new engine.
e. Click **Save All** to save your changes.

2. Add a configuration for the new engine.
   a. Open the *Engine Config Window* (**Main Entry -> Configuration -> System -> Engine**).

Figure 6-2: Engine Configuration Window — Sample Engine



b. Select your new engine from **Engine Name**. Your engine name must first be entered in **engineName** (*Domain Values Window*). See Step 1 on page 39.

c. Assign an unused **ID**.

d. Enter a **Comment**, if desired.

e. Click **Save** to retain your changes

Note: If your engine ID conflicts with a Calypso engine ID in a subsequent release, you must ensure that all events for your engine are processed and then change the conflicting engine ID to the next available engine ID.

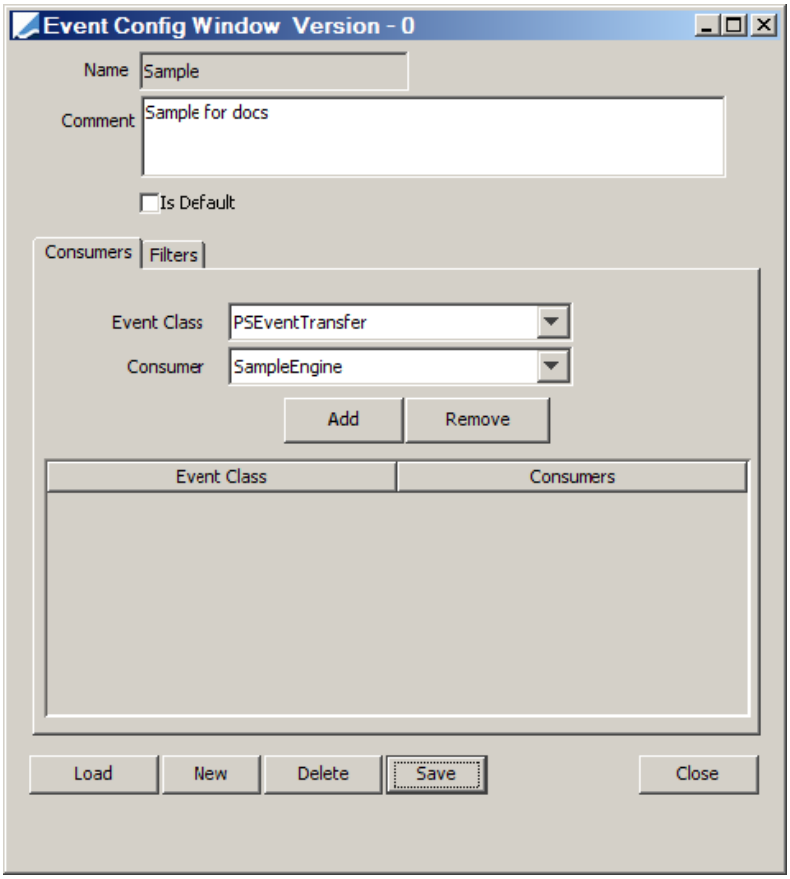
Step 3 — For Subscribing to Persistent Events Only

Use the *Event Config Window* (**Main Entry-> Configuration-> System-> Event**) to specify the persistent events to which the engine subscribes.

1. Click **New** to create a new persistent subscription.
2. Choose an **Event Class** for the subscription.
3. Select the appropriate **Consumer**. In this case, your new engine.
4. Click **Save** to retain the new subscription. The *Event Config Name Window* will open.
5. Either select an existing **Event Config Name**, or enter a new name, then click **OK**. The *Event Config Name Window* will close.

6. Click **Close** on the *Event Config Window*.

Figure 6-3: Event Config Window after Saving a New Subscription



Persistent events are returned by *Engine.getPersistentClasses()*.

Remember to specify any required event filters.

Step 4 — Starting the New Engine

In the client application, create a connection to the Data Server, instantiate the engine class, and call *start(Boolean doBatch)* on the engine. If doBatch is set to true, the engine will call *RemoteTrade.getEngineEvents()* in a separate thread to load and process any outstanding persistent events.

When *start()* is called on the engine, the engine starts a connection to the Event Server. The Event Server, based on the event types returned from the method *getPersistentClasses()*, sends the application any events that the persistent engine subscribes to and has missed since the last session. Upon receiving an event, the engine automatically creates a new thread and calls the method *process()*. If the event received is of a type that the engine subscribes to, the engine notifies the Event Server upon the completion of event processing by calling *RemoteTrade.eventProcessed()*.

You can also use the sample engine that subscribes to PSEventTransfer events to process payments. The subscription is established in the database. Refer to the script `samples/sql/sampleEngine.sql` for an example.

6.2 How to Customize the Transfer Engine

The Transfer engine uses a process based on a BOProductHandler for a given product, and the Transfer rules if settlement instructions have been set up, to generate transfers as follows.

1. The BOProductHandler creates all the transfers relating to a Trade using cash flows and applying the settlement instructions.

   If the Transfer Engine is processing a given Trade for the first time, all transfers are generated from the beginning of the trade. If transfers for the trade were generated previously, the Transfer Engine has filters the Transfers to the those having a settle date on or after the current day's date, and which do not have a CANCELED status. If the event is a PSEventProcessTrade, the limit date for back-value payment regeneration is specified. You can create a custom date filter, and you can also customize how to set the dates on the transfers.

2. The Transfer Engine then compares the generated transfers to the existing transfers and matches them based on the following criteria. If a transfer does not match, it is considered unmatched.

3. The NettingHandler will create netted transfers if netting is required by the transfers. The Netting Type field on the Transfer indicates if netting is required and what type of netting should occur. You can customize the netting selector. You can customize the netting handler by netting type.

6.2.1 Creating a Custom BOProductHandler

The base class for producing trade cashflows and transfers with settlement instructions is **`com.calypso.tk.bo.BOProductHandler`**. Each product will usually have its own BOProductHandler. A **`BOProduct<`*`product_type`*`>Handler`** can itself extend or override another **`BOProduct<`*`product_type`*`>Handler`**.

Create a class named **`tk.bo.BO<`*`product_type`*`>Handler`** or **`tk.bo.BO<`*`product_family`*`>Handler`** which extends **`com.calypso.tk.bo.BOProductHandler`**.

A BOProductHandler may define the following methods: *generateTransferRules*, *generateTransfers*, *exercise*, *UpdateTransfer*, and *addSecurityFlows*.

This class will be invoked from **`com.calypso.tk.bo.BOProductHandler`**.

Sample Code in `calypsox/tk/bo/`

- `BODEMO_P1Handler.java`
- `BODEMO_P2Handler.java`
- `BOIROptionHandler.java`
- `BOStraddleHandler.java`
- `BORepoHandler.java`

`BORepoHandler` is a custom handler that extends **`com.calypso.tk.bo.BORepoHandler`**. In the event that a coupon occurs during the life of a repo, no messages or postings are generated for the transfers related to this coupon.

6.2.2 BOSimpleRepoHandler

BOSimpleRepoHandler is a custom handler that extends **`com.calypso.tk.bo.BOSimpleRepoHandler`**. It provides, for example, the possiblity to add an attribute to a Transfer by using the *UpdateTransfer()* method.

6.2.3 Creating a Custom Interest Dispatch Process

It is possible to customize the behavior of the interest dispatch process for repos.

Create a class named **`tk.bo.BORepo<dispatch_method>DispatchInterestHandler`** that implements **`com.calypso.tk.bo.BORepoDispatchInterestHandler`**.

BORepoDispatchInterestHandler has two methods:

- *dispatchInterest()* called from *addSecurityFlow()*, allows creating as many interest flows as necessary.
- *updateInteretTransfer()* called from *preProcessDAPTransfers()*, allows linking transfers created because of the new interest flows to the security and principal transfers of each collateral.

Then register the dispatch method with the "Repo.DispatchInterestMethod" domain.

This class will be invoked from **`com.calypso.apps.tk.bo.BORepoHandler`**.

6.2.4 Creating a Custom Date Filter

The default behavior will generate the Product transfers up to the next event date if the XFER_NEXT_EVENT parameter is set to True.

Create a class named **`tk.product.<product_type>ProductNextEventDate`** that implements the interface **`com.calypso.tk.product.ProductNextEventDate`**.

<*product_type*>ProductNextEventDate is invoked by **`com.calypso.tk.product.ProductNextEventDateUtil`**, which is used by BOProductHandler.

6.2.5 Creating a Custom Date Selector

Create a class named **tk.bo.CustomBOTransferDateSelector** which implements the interface **com.calypso.tk.bo.BOTransferDateSelector**.

CustomBOTransferDateSelector is invoked by **com.calypso.tk.bo.BOProductHandler** to set the dates on the transfers.

Sample Code in calypsox/tk/bo/

SampleCustomBOTransferDateSelector.java

6.2.6 Creating a Custom Transfer Matching Mechanism

This interface determines whether a Transfer must be canceled, updated, or created. For instance, it allows the user to disable some criteria during the matching of two transfers.

Create a class named **engine.payment.<*product_type*>TransferMatching** or **engine.payment.DefaultTransferMatching**, which implements the interface **com.calypso.engine.payment.TransferMatching**.

The *TransferMatching class is invoked by **com.calypso.engine.payment.TransferMatchingUtil**, which is used by TransferEngine to select the transfer matching mechanism.

The default transfer matching mechanism is based on the CHECK_PAST_SDI_VERSION property. If CHECK_PAST_SDI_VERSION is set to False and the SDI version number has changed, transfers will not be regenerated. If CHECK_PAST_SDI_VERSION is set to True and the SDI version number has changed, transfers will be regenerated.

Sample Code in calypsox/engine/payment/

SampleDefaultTransferMatching.java

6.2.7 Creating a Custom Netting Handler

You can create a custom netting handler to modify how netting is done.

Create a class named **tk.bo.<*netting_handler*>NettingHandler** that implements **com.calypso.tk.bo.NettingHandler**. The default implementation is DefaultNettingHandler.

Register the *netting_handler* in the **nettingHandler** domain. You can associate a custom netting handler with a netting type in the Netting Config window.

6.2.8 Creating a Custom Netting Method Selector

A custom Netting Method Detector automatically sets the Netting Type on the Transfer Rules and lets you override the Netting Method default value.

Create a class named **tk.bo.<product_type>NettingSelector** which implements the interface **com.calypso.tk.bo.NettingSelector**.

*NettingSelector is invoked by **com.calypso.tk.bo.NettingSelectorUtil** to set the netting method.

Sample Code in calypsox/tk/bo/

- SimpleMMNettingSelector.java
- SampleCustomBondNettingSelector.java

6.2.9 Creating a Custom Persistence Routine for Transfer Attributes

Create a class named **tk.bo.sql.CustomTransferAttributeSQL** that implements the interface **com.calypso.tk.bo.sql.TransferAttributeSQL**.

CustomTransferAttributeSQL is invoked from **com.calypso.tk.bo.sql.BOTransferSQL** when saving transfers.

TransferAttributeSQL includes methods for archiving custom attributes that must be implemented. To fully support archiving and restoring of custom attributes, you must define new history tables for any custom attribute tables you have added to the database.

6.3 Customizing the Message Engine

The Message engine generates messages for various events as applicable (Trade, Transfer, etc), using the following process:

- The Message engine identifies the roles to which sending the message using TradeRoleFinder. Once, the roles are determined, receiver contact information can be retrieved. A message is generated for each receiver using a BOMessageHandler provided MessageSelector confirms the message must be generated. The behavior of TradeRoleFinder and MessageSelector can be customized. You can create a custom BOMessageHandler for a given product.
- BOMessageHandler builds the messages based on the Message setup rules. It uses FormatterUtil to select a type of Formatter (SWIFT, HTML, etc.) and a template. It uses MessageFormatterUtil to select a MessageFormatter for populating the template. The MessageFormatter is selected based on the type of Formatter and the product. You can create a custom type of Formatter, a custom template selector, a custom MessageFormatter selector, and a custom MessageFormatter.

6.3.1 Creating a Custom Update Transfer

A custom Update Transfer can modify the Transfer depending on the Transfer, Trade, and Cashflow.

Create a class named **CustomUpdateTransfer** that implements the interface **com.calypso.tk.bo.UpdateTransfer**.

6.3.2 Creating a Custom Role Finder

For example, you want to retrieve the possible receivers of a message based on the Legal Entity role. Write this class to retrieve the Legal Entities for a given Role defined in a Trade, a Product, or a Transfer.

Create a class named **tk.bo.<*product_type*>RoleFinder** or **tk.bo.<*product_family*>RoleFinder** which extends the class **com.calypso.tk.bo.TradeRoleFinder**.

This class will be invoked from **com.calypso.tk.bo.TradeRoleFinder**.

Sample Code

- com/calypso/tk/bo/TransferAgentRoleFinder.java
- calypsox/tk/bo/RepoRoleFinder.java

6.3.3 Creating a Custom Message Selector

For instance, the SampleCustomMessageSelector returns False when there is no SWIFT address for the Receiver Contact, causing a message to not be generated by the Message engine.

Create a class named **engine.advice.CustomMessageSelector** which implements the interface **com.calypso.engine.advice.MessageSelector**.

This class is invoked by **com.calypso.engine.advice.MessageEngine** to establish if a message must be generated.

Sample Code in calypsox/engine/advice/

- SampleCustomMessageSelector.java
- AnotherCustomMessageSelector.java
- YetAnotherCustomMessageSelector.java

6.3.4 Customizing Message Selection

A custom interface allows customers to implement a specific behavior of AdviceConfigSelection in the MessageEngine.

The customized implementation must be called **engine.advice.CustomAdviceConfigSelector** and must implement **com.calypso.engine.advice.AdviceConfigSelector**.

The Default backward compatible implementation is **com.calypso.engine.advice.DefaultAdviceConfigSelector**, which is instantiated by default.

A sample custom implementation is located in the calypsox directory. This custom implementation generates a message for a given event type for a specific product type, as well as a message for the product type ALL. Hence, it is no longer necessary to duplicate an advice config product type ALL for a given eventType if an advice config for particular productType and eventType already exists.

Sample Code in calypsox/engine/advice/

- CustomAdviceConfigSelector

6.3.5 Creating a Custom Message Handler

Create a class named **tk.bo.BO<*product_type*>MessageHandler** or **tk.bo.BO<*product_family*>MessageHandler** which extends **com.calypso.tk.bo.BOMessageHandler**.

Note: For message types without a product type, such as STATEMENT, you can create a class named **tk.bo.BOSTATEMENTMessageHandler**. In this case, the product-specific message handler is simply skipped and the application uses the more generic message-type-based handler.

You may be redefine the following BOMessageHandler methods:

- *generateBOMessages()* — Defines what messages should be produced. For example, for an FX Swap confirmation by Swift, it would create two messages, one for each leg.
- *setSpecialMessageEnvironment()* — The Message engine calls this function whenever an existing message has been found. It allows you to set a link between each message and decide what should be done on the previous existing message. Typically this function sets the keyword AMEND, CANCEL or NEW.
- *filterMessages()* — Returns the existing message matching exactly the new one which should be generated. For example, if you have already produced a Bond Confirmation, it will return the existing Bond Confirmation already generated for the Trade.
- *isMessageRequired()* — Allows you to decide if the new message should be created. This function provide you with access to the previous message generated. You can therefore perform any type of comparison required.
- *fillDefaultMessageInfo()* — Called to initialize each message created by the Message Engine.
- *getTradeFieldsNotAmendmentDomain()* — Selects the domain containing the trades fields that can be amended without triggering the generation of a new BO Message.
- *getFeeNotAmendmentDomain()* — Selects the domain containing the fees names that can be amended on a trade without triggering the generation of a new BO Message.

This class will be invoked from **com.calypso.tk.bo.BOMessageHandler**.

6.3.6 Creating a Custom Type of Formatter

Calypso provides formatter support for HTML, Text, SWIFT, and XML, standard. Howver, it might be necessary to support the FIX format and hence, create a FIX generator class.

Create a class named **tk.bo.<*format_type*>Formatter** which implements the interface **com.calypso.tk.bo.Formatter**. Implement the methods *generate()* to generates an advice document, and *display()* to displays the advice document in the Task Station.

<*format_type*>Formatter is invoked from **com.calypso.tk.bo.FormatterUtil**.

6.3.7 Creating a Custom Template Selector

Create a class named **tk.bo.<*product_type*>TemplateSelector** which implements the interface **com.calypso.tk.bo.TemplateSelector**.

Invoke TemplateSelector from **com.calypso.tk.bo.FormatterUtil** to choose a template selector.

Sample Code in calypsox/tk/bo/

SampleSwapTemplateSelector.java

6.3.8 Creating a Custom Message Formatter Selector

Create a class named **tk.bo.<message_type>MFSelector** which implements the interface **com.calypso.tk.bo.MFSelector**.

Invoke MFSelector from **com.calypso.tk.bo.MessageFormatterUtil** to choose a MessageFormatter selector.

6.3.9 Creating a Custom Message Formatter

A Message Formatter is responsible for extracting the information from a trade and matching the appropriate keywords in the template. See Section 8.1, "How to Create an HTML Template," on page 64 for information on how to create an HTML template.

Create a class named **tk.bo.<message_type><product_type>MessageFormatter**, **tk.bo.<product_type>MessageFormatter** or **tk.bo.CustomMessageFormatter** which extends **tk.bo.MessageFormatter**.

In your MessageFormatter, create a *parse<keyword_name>()* method for each keyword you wish to add. For example *parseRATE_INDEX()*.

Implement each parse method to take the following arguments and return the keyword value for a given situation, as defined by the passed arguments.

- `message` — the message which will use the returned keyword value;
- `trade` — the trade with which the advice is associated;
- `sender` — the contact person sending the advice (LEContact);
- `rec` — the contact person receiving the advice (LEContact);
- `transferRules` — a Vector of TradeTransferRule objects which provide the general definition of any payments associated with the advice;
- `transfer` — the payment, if any, associated with the advice;
- `dsCon` — a connection to the Data Server.

Note: In your parse method, you can check if a custom keyword is being evaluated within an IF statement using *FormatterParser.isConditionalEvaluation()*, which will return a Boolean.

A MessageFormatter class will be invoked from **com.calypso.tk.bo.MessageFormatterUtil**.

CustomMessageFormatter is invoked from **com.calypso.tk.bo.MessageFormatter**.

Sample Code in calypsox/tk/bo/

- RATE_RESETSwapMessageFormatter.java
- StructuredProductMessageFormatter.java
- SwapMessageFormatter.java
- XLSMessageFormatter.java

6.3.10 Creating a Custom Persistence Routine for Message Attributes

Create a class named **tk.bo.sql.CustomMessageAttributeSQL** that implements the interface **com.calypso.tk.bo.sql.MessageAttributeSQL**.

MessageAttributeSQL is invoked from **com.calypso.tk.bo.sql.BOMessageSQL** when saving messages.

MessageAttributeSQL includes methods for archiving custom attributes that must be implemented. To fully support archiving and restoring of custom attributes, you must define new history tables for any custom attribute tables you have added to the database.

6.3.11 Creating a Custom XML Generator

Create a class named **tk.bo.xml.<template_name>XMLGenerator** or **tk.bo.xml.DefaultXMLGenerator** that implements the interface **com.calypso.tk.bo.xml.XMLGenerator**.

XMLGenerator is invoked from **com.calypso.tk.bo.xml.XMLUtil**.

Sample Code in calypsox/apps/util/

- XMLGenerator.java

6.3.12 Creating Multiple BOMessages per Message Type

In a class named **tk.product.*<product_type>*** implement the interface **com.calypso.tk.product.MultiMessageProduct**.

For example, an FX Swap needs two trade confirmations when you verify the trade; one confirm for the near leg and a second confirm for the far leg.

6.3.13 How to Customize a Statement Message

The AccountStatementInterface interface is used by the Message engine to set the Message Reference on the Statement messages, It provides

flexibility in the population of the fields: *fillMessageReference(Message-Array message, AccountStatement statement, DSConnection dsCon).*

To customize the statements, create a class named **tk.bo.swift.AccountStatementCustomizer** that implements AccountStatementInterface.

6.4 How to Customize SWIFT Messages

SWIFT messages can be generated using a SwiftGenerator for each type of message, or from an XML template using SWIFTFormatter. Based on the template name, the message engine first tries to instantiate a SWIFTFormatter. If no SWIFTFormatter is available, the message engine then uses a SwiftGenerator.

6.4.1 Using SwiftGenerator

Creating a Custom SwiftGenerator

Create a class named **tk.bo.swift.<*template_name*>SwiftGenerator** which implements the **com.calypso.tk.bo.swift.SwiftGenerator** interface.

For example, if you are creating a SWIFT message for FX Swap payments then the name of the class might be FXSwapPaymentSwiftGenerator. You must add the template name in the domain values. Add the domain FXSwapPayment to the domain "SWIFT.Templates" so that the template is selectable in the message setup window.

FXSwapPaymentSwiftGenerator is invoked from **com.calypso.tk.bo.SWIFTFormatterUtil**.

Sample Code in `calypsox/tk/bo/swift/`

- `FRAConfirmSwiftGenerator.java`
- `FXPaymentSwiftGenerator.java`
- `FXReceiptSwiftGenerator.java`
- `FXSwapPaymentSwiftGenerator.java`
- `FXSwapReceiptSwiftGenerator.java`

6.4.2 Using SWIFTFormatter

Creating a Custom SWIFTFormatter

This allows creating a custom SWIFTFormatter that does not use SwiftGenerator. The XML templates should be placed in `templates/swift`. See the SWIFTFormatter Javadoc for details.

Create a class named **tk.bo.swift.<name>SWIFTFormatter** that extends **com.calypso.tk.bo.swift.SWIFTFormatter**.

SWIFTFormatter is invoked from **com.calypso.tk.bo.SWIFTFormatterUtil**.

Creating a Custom Iterator to Populate Repeated Information

Create a class named **tk.bo.swift.<name>Iterator**, **tk.bo.swift.<name>**, or **<name>** that implements the interface **java.util.Iterator**.

The SWIFTFormatter can access the iterator count and the iterator object, and behave as applicable. See the SWIFTFormatter Javadoc for details.

Sample Code in calypsox/tk/bo/swift/

TestIterator.java

Creating a Custom Header Block

Create a class named **tk.bo.swift.SwiftTextCustomizer** that implements the interface **com.calypso.tk.bo.swift.SwiftTextInterface**.

Sample Code in calypsox/tk/bo/swift/

SwiftTextCustomizer.java

6.4.3 Applying Custom Validation to SWIFT Messages

Create a class named **tk.bo.swift.CustomSwiftMessageValidator** that implements the interface **com.calypso.tk.bo.swift.SwiftMessageValidator**.

CustomSwiftMessageValidator is invoked by **com.calypso.tk.bo.swift.SwiftMessage** before the message is saved.

Sample Code in calypsox/tk/bo/swift/

SampleCustomSwiftMessageValidator.java

6.4.4 Customizing IsFinancial in SWIFT Messages

Create a class named **tk.bo.swift.CustomSwiftUtilInterface** that extends **com.calypso.tk.bo.swift.DefaultSwiftUtilInterface**.

Method getIsFinancial

This method determines whether or not the provided Legal Entity is an financial organization.

Input **LegalEntity le —** The Legal Entity

BOMessage message — Future Use

Returns **True —** The Legal Entity is a financial organization.

False — The Legal Entity is not a financial organization.

Usage Boolean getIsFinancial(BOMessage message, LegalEntity le)

This method is typically used to set the value of the isFinancial variable.

6.4.5 Custominzing EntityInfo for SWIFT Messages

Create a class named **tk.bo.swift.CustomEntityInfo** that implements **EntityInfo**.

CustomEntityInfo is invoked by
com.calypso.tk.bo.swift.SwiftUtil.

6.5 Customizing the Sender Engine

The Sender engine uses DocumentSender objects to physically transmit message documents to a given address method or gateway. Calypso provides EMAILDocumentSender to send documents via e-mail. Address methods are stored in the addressMethod domain.

Note: When sending Advice Messages, users must strip the message prior to calling the *send()* method.

6.5.1 Creating a Custom Document Sender

Create a class named
tk.bo.document.<*address_method*>DocumentSender,
tk.bo.document.Gateway<*gateway*>DocumentSender, or
tk.bo.document.<*address_method*>Gateway<*gateway*>DocumentSender,
that implements the **com.calypso.tk.bo.document.DocumentSender** interface.

The DocumentSender is invoked by
com.calypso.tk.bo.document.DocumentSenderUtil.

In your DocumentSender, create the *send()* method to send the message.

Note: When dealing with an Advice message, you must first call *SwiftMessage.stripExtraInfo(AdviceDocument.getDocument())* to strip the message prior to calling *send()*.

Generally the *send()* method initiates the physical production of the document via some output mechanism such as a printer or email utility. The *send()* method must return a Boolean True if successful, or if not, False. You must also define the *isOnline()* method. The Sender engine will query this isOnline to ensure that the sender gateway system is online.

DocumentSender has the ability to also process a PSEventMessage event, in addition to sending an Advice document. The following parameters of the *send()* method should be used as described below:

- **saved** — **saved[0]** should be true to indicate if the document was saved and the event processed
- **engineName** — should be null to indicate that the Sender engine need not process the event

Sample Code in calypsox/tk/bo/document/

- GatewayPRINTERDocumentSender.java
- SWIFTDocumentSender.java

6.6 Customizing the Accounting Engine

The Accounting engine using the following process to generate postings for various events (Trade, Valuation, Liquidation, etc.) as applicable:

- The Accounting engine selects which accounting rule to apply using a mapping mechanism between the events it subscribes to and the accounting rules configured in the system. It builds a list of requested accounting events based on the selected accounting rule. The mapping mechanism can be customized.
- For each accounting event, the Accounting engine calls a generic AccountingHandler to specify how to generate the corresponding posting. AccountingHandler can call a specific AccountingHandler for a given product type, or a specific AccountingEventHandler for a given type of accountzing event. The Accounting engine also allows adding custom attributes to the generated postings.

  Beginning with Rel1000P3, the AccountingHandler calls methods in either DefaultFeeAccountingHandler or CustomFeeAccountingHandler, if defined.
- Once all the postings have been created, a matching process occurs to compare a set of criteria on the new postings and the old postings, and to generate reverse postings as applicable. The matching mechanism can be customized.

6.6.1 Generating Fee-Related Postings

In Calypso Rel1000P3 and above, you can define a CustomFeeAccountingHandler that extends DefaultFeeAccountingHandler which implements FeeAccountingHandler to support new fee-related event types, or to redefine existing event types.

For a fee-related accounting event type such as PREMIUM, call the getFee() method. For an accounting event type such as PREMIUM_AM, the use the getFeeAM() method.

The FeeAccountingHandler interface contains methods for the Calypso-defined, fee-related events. Method invocation is by reflection, therefore, you can add your own methods to your CustomFeeAccountingHandler without changing the interface.

6.6.2 Creating a Custom Mapping Mechanism to an Accounting Rule

Create a class named **`engine.accounting.CustomAccountingRuleSelector`** that implements the interface **`com.calypso.engine.accounting.AccountingRuleSelector`**.

CustomAccountingRuleSelector is invoked from **`com.calypso.engine.accounting.AccountingEngine`** to select an accounting rule.

Sample Code in calypsox/engine/accounting/

- SampleCustomAccountingRuleSelector.java

6.6.3 Creating a Custom Accounting Handler

Create a class named **tk.bo.accounting.<product_type>AccountingHandler** or **tk.bo.accounting.<product_family>AccountingHandler** which extends **com.calypso.tk.bo.accounting.AccountingHandler**.

The AccountingHandler should have a *get<accounting event type>()* method for each accounting event type that it will calculate. Accounting event types are listed in the accEventType domain. The set of accounting event types that your system will calculate is established in the AccountingEventConfig objects for a given AccountingRule. For example, *getCOT()* calculates a COT accounting event.

Product AccountingHandler is invoked from **com.calypso.tk.bo.accounting.AccountingHandler** to generate a posting for a given product type of family type.

Sample Code in calypsox/tk/bo/accounting/

- FXForwardTakeUpAccountingHandler.java
- IROptionAccountingHandler.java

6.6.4 Creating a Custom Event Accounting Handler

Create a class named **tk.bo.accounting.<event_type>AccountingHandler** that implements the interface **com.calypso.tk.bo.accounting.EventAccountingHandler**.

Event AccountingHandler is invoked from **com.calypso.tk.bo.accounting.AccountingHandler** to generate a posting for a given type of accounting event.

Sample Code in calypsox/tk/bo/accounting/

- EXT_EVENT_TYPEAccountingHandler.java
- EXT_MTM_FULLAccountingHandler.java

6.6.5 Creating a Custom Posting Description

Create a class named **engine.accounting.CustomFillPostingDescription** that implements the interface **com.calypso.engine.accounting.FillPostingDescription**. You can implement *fillDescription()* for adding attributes and *fillPostingDates()* for customizing the dates set on the posting: booking date and effective date.

CustomFillPostingDescription is invoked from **com.calypso.engine.accounting.AccountingEngine** to customize the content of the postings.

Sample Code calypsox/engine/accounting/

- SampleCustomFillPostingDescription.java

  For example, this API is used to fulfill the following request: To freeze the image of the postings during the EOD process, the posting status is changed from NEW to EOD_PROCESSING in *fillDescription()*. Hence if any cancellation occurs during the EOD process, the EOD_PROCESSING posting will be reversed and a NEW posting will be created. Otherwise the original posting would just move to status DELETE.

6.6.6 Creating a Custom Accounting Matching Mechanism

Create a class named **engine.accounting.<product_type>AccountingMatching** or **engine.accounting.DefaultAccountingMatching** that implements the interface **com.calypso.engine.accounting.AccountingMatching**.

AccountingMatching is invoked from **com.calypso.engine.accounting.AccountingMatchingUtil** when matching old postings and new postings for generating reverse postings if applicable.

Sample Code in calypsox/engine/accounting/

- SampleSwapAccountingMatching.java

6.6.7 Creating a Custom Account Keyword for Automatic Accounts

Create a class named **tk.bo.accounting.keyword.<keyword>AccountKeyword** which implements the interface **com.calypso.tk.bo.accounting.keyword.AccountKeyword**.

AccountKeyword is invoked from **com.calypso.tk.bo.accounting.keyword.KeywordUtil**.

Then register the <keyword> in the **attributeType** domain. For example, for `calypsox/tk/bo/accounting/keyword/IBANAccountKeyword.java`, you must register IBAN in the **attributeType** domain.

Sample Code in calypsox/tk/bo/accounting/keyword/

- IBANAccountKeyword.java
- InitialMaturityAccountKeyword.java
- MatTenorAccountKeyword.java
- MethodAccountKeyword.java
- OnTimeAccountKeyword.java
- RIBAccountKeyword.java

6.6.8 Applying Custom Validation to Accounting Rules

Create a class named **apps.refdata.CustomAccRuleValidator** which implements the interface **com.calypso.apps.refdata.AccRuleValidator**.

CustomAccRuleValidator is invoked from **com.calypso.apps.refdata.AccountingRuleFrame** prior to saving an accounting rule.

6.6.9 Applying Custom Validation to an Account

Create a class named **apps.refdata.CustomAccountValidator** that implements **com.calypso.apps.refdata.AccountValidator**.

CustomAccountValidator is invoked from **com.calypso.apps.refdata.AccountFrame**.

Sample Code in calypsox/apps/refdata/

CustomAccountValidator.java

6.6.10 Creating a Custom Closing Account Name

Create a class named **tk.bo.accounting.CustomClosingAccountName** that implements the interface **com.calypso.tk.bo.accounting.ClosingAccountName**.

CustomClosingAccountName is invoked from **com.calypso.tk.bo.BalanceUtil** when assigning a closing account.

Sample Code in calypsox/tk/bo/accounting/

SampleCustomClosingAccountName.java

6.6.11 Creating a Custom External Name for Automatic Accounts

Create a class named **tk.bo.accounting.keyword.CustomAccountExternalName** that implements **com.calypso.tk.bo.accounting.keyword.AccountExternalName**.

CustomAccountExternalName is invoked from **com.calypso.tk.bo.accounting.keyword.KeywordUtil**.

Sample Code in calypsox/tk/bo/accounting/keyword/

CustomAccountExternalName.java

6.6.12 Add Custom Attributes to BOPosting.

Custom attributes are saved when the postings are created so that they are part of the matching.

Create a class to generate the attributes, named, **tk/bo/accounting/CustomFillPostingAttribute** that implements **com/calypso/tk/bo/accounting/FillPostingAttribute**.

The attributes are saved in the posting_attribute table. You can customize the save operation by creating a class named

tk/bo/sql/CustomPostingAttributeSQL that implements **com/calypso/tk/bo/sql/PostingAttributeSQL**.

6.7 How to Customize the Position Engine

6.7.1 Creating a Custom Liquidation Method

Create a class named **engine.position.Liquidation<liquidation_method>** that extends the class **com.calypso.engine.position.Liquidation**. By default, liquidation methods are stored in the **liquidationMethod** domain.

Liquidation<*liquidation_method*> is invoked from **com.calypso.engine.position.LiquidationUtil**.

6.7.2 Creating a Custom Sort Method

Create a class named **tk.mo.Comparator<sort_method>** that implements **java.util.Comparator**. By default, Calypso sort methods are stored in the **sortMethod** domain.

Comparator<*sort_method*> is invoked from **com.calypso.engine.position.LiquidationUtil** for sorting open positions.

6.7.3 Creating a Custom Routine for Computing the Liquidation Date

Create a class named **tk.mo.LiquidationDateCalculator** that implements the **com.calypso.tk.mo.LiquidationInfoCalculator** interface.

LiquidationDateCalculator is invoked from **com.calypso.tk.mo.LiquidationInfo**.

Sample Code in calypsox/tk/mo/

LiquidationDateCalculator.java

6.8 How to Customize the Inventory Engine

6.8.1 Creating a Custom Inventory Position Selector

For example, you want to customize the list of Positions classes handled by the Inventory Engine: INTERNAL, CLIENT, EXTERNAL.

Create a class named **engine.inventory.InventoryPositionSelector** that implements the **com.calypso.engine.inventory.PositionSelector** interface.

InventoryPositionSelector is invoked from **com.calypso.engine.inventory.InventoryEngine**.

Sample Code in calypsox/engine/inventory/

SampleInventoryPositionSelector.java

6.9 How to Customize the CRE Engine

The CRE engine generates CREs (accounting events). The CRE engine calls a generic CreHandler to specify how to generate a CRE. CreHandler can call a specific CreHandler for a given product type, or a specific CreEventHandler for a given accounting event. CreHandler also allows adding custom attributes to the generated CREs.

6.9.1 Creating a Custom CRE Handler

Create a class named **tk.bo.accounting.<product_type>CreHandler** or **tk.bo.accounting.<product_family>CreHandler** that extends **com.calypso.tk.bo.accounting.CreHandler**.

Implement a *get<accounting event type>()* method for each accounting event. For example, *getCOT()* for the COT accounting event.

The Product CreHandler is invoked from **com.calypso.tk.bo.accounting.CreHandler** to generate a CRE for a given product type of family type.

6.9.2 Creating a Custom Event CRE Handler

Create a class named **tk.bo.accounting.<accouting_event_type>CreHandler** that implements the **com.calypso.tk.bo.accounting.EventCreHandler** interface.

The Event CreHandler is invoked from **com.calypso.tk.bo.accounting.CreHandler** to generate a CRE for a given type of accounting event.

6.9.3 Creating a Custom CRE Description

Create a class named **com.calypso.tk.bo.accounting.CustomFillCreDescription** that implements **com.calypso.tk.bo.accounting.FillCreDescription**.

CustomFillCreDescription is invoked from **com.calypso.tk.bo.accounting.CreHandler** to add custom attributes to the generated CREs.

Sample Code in calypsox/tk/bo/accounting/

SampleCustomFillCreDescription.java

6.9.4 Creating a Custom Persistence Routine for CRE Attributes

Create a class named **tk.bo.sql.CustomCreAttributeSQL** that implements **tk.bo.sql.CreAttributeSQL**.

CreAttributeSQL includes methods for archiving custom attributes that must be implemented. To fully support archiving and restoring of custom attributes, you must define new history tables for any custom attribute tables you have added to the database.

CustomCreAttributeSQL is invoked from `com.calypso.tk.bo.sql.BOCreSQL` when saving CREs.

6.10 How to Customize the CRE Sender Engine

The CRE Sender engine sends the CREs generated by the CRE engine. During the send, the status of a CRE is updated (to SENT, RE_SENT or DELETED), and the CreSenderFormater API is called to produce the output of a CRE. You must implement CreSenderFormater. Note that the scheduled task CRE_SENDER also calls CreSenderFormater for formatting CREs.

6.10.1 Creating a Custom CRE Formatter

Create a class named `engine.accounting.CreSenderFormaterImpl` that implements `com.calypso.engine.accounting.CreSenderFormater`.

CreSenderFormaterImpl is invoked from `com.calypso.tk.bo.sql.BOCreSQL` when saving CREs.

6.11 Customizing the Hedge Relationship Engine

The Hedge Relationship engine uses the following process to automatically generate Hedge Relationships using a saved Hedge relationship Configuration as a template and for events (Trade, etc.) as applicable:

- The Hedge Relationship engine selects the hedge relationship configuration to apply using a mapping mechanism between the events it subscribes to and the hedge relationship configuration configured in the system. It builds a list of new hedge relationships based on the selected configuration and on existing relationships also. The mapping mechanism can be customized.
- For each hedge relationship configuration, the Hedge Relationship engine calls a generic **HedgeRelationshipConfigHandler** to specify how to generate the corresponding hedge relationships. The **HedgeRelationshipConfigHandler** can call a specific **HedgeRelationshipConfigHandler** for a given Trade and a specific **HedgeRelationshipConfig**. The Hedge Relationship engine also allows adding custom attributes to generate the Hedge Relationships.
- Once all the new relationships have been built, a matching process occurs to compare a set of criteria (ratio, start date, end date, etc..) on the new hedge relationships and the old relationships. The matching mechanism cannot be customized.

6.12 Creating a Custom Hedge Relationship Config Handler

Create a class named **tk.bo.HedgeRelationshipConfigHandler**.

The **HedgeRelationshipConfigHandler** should have a **buildHedgingTradeRelationships()** method to build hedge relationships for a hedging trade based on the configuration set up and a **buildHedgedTradeRelationships()** method to build hedge relationships for a hedged trade based on the configuration set up.

6.13 Customizing the Diary Engine

The Diary Engine listens for trade creation and modification, and generates trade diary objects which stand for all the events related to the trade and to its lifecycle.

The Diary Engine calls a generic trade diary handler to specify how to generate the list of diary objects, but it can call a specific handler for a given product. This handler implements the interface **TradeDiaryHandler** (package **com.calypso.tk.product**).

6.13.1 Creating a Custom Trade Diary Handler

Create a class named **tk.product.*product_type*TradeDiaryHandler** that implements the interface **com.calypso.tk.product.TradeDiaryHandler** and create a **getDiary()** method exposed by the interface.

If you wish to add a diary object to the generic list, your **tk.product.*product_type*TradeDiaryHandler** custom class must extend the **com.calypso.tk.product.DefaultTradeDiaryHandler** generic class and override the **getDiary()** method.

6.14 Customizing the Billing Engine

The purpose of the Billing Engine is to generate Billing Fees that must be charged to or delivered to a CounterParty, an Agent, or any Legal Entity involved in a Trade. These Fees are aggregated in a Billing Trade with this Legal Entity as a Counterparty.

The Billing Engine is able to react on PSEVentTrade(s), PSEventTransfer(s), PSEventMessage(s), PSEventAccountBilling(s), PSEventTask(s) and PSEventMaintenanceTrade(s).

When processing an incoming event, the Billing Engine first creates, and then processes a BillingEvent.

The BillingEngine is also able to translate any PSEvent, including Custom events, into a Custom Billing Event.

6.14.1 Defining a Custom Billing Event

A Billing Event is a Java Class that must implement a Java Interface called BillingEvent. The BillingEvent Interface is defined as follows:

```
public interface BillingEvent extends Externalizable {
static final long serialVersionUID = -2533362755239752335L;
public void init(int accountId,
            int tradeId,
            int tradeVersion,
            int transferId,
            int transferVersion,
            int messageId,
            int messageVersion,
            int poId,
            int bookId,
            int leId,
            String leRole,
            JDate valueDate);

   public int getBookId();
   public Book getBook() throws Exception;
   public int getProcessingOrgId();
   public LegalEntity getProcessingOrg() throws Exception;
   public int getLegalEntityId();
   public LegalEntity getLegalEntity() throws Exception;
   public String getLegalEntityRole();
   public int getAccountId();
   public List<Integer> getAccountIds();
   public Account getAccount() throws Exception;
   public int getTradeId();
   public Trade getTrade() throws Exception;
   public int getTradeVersion();
   public int getTransferId();
   public BOTransfer getTransfer() throws Exception;
   public int getTransferVersion();
   public int getMessageId();
   public BOMessage getMessage() throws Exception;
   public int getMessageVersion();
   public long getTaskId();
   public Task getTask() throws Exception;
   public int getTaskVersion();
   public boolean matches(BillingEvent event);
   public String getObjectType();
   public int getObjectId();
   public Object getObject() throws Exception;
   public int getObjectVersion();
   public JDate getDefaultValueDate();
   public JDate getValueDate(String dateType) throws Exception;
   public boolean getUnderlyingObjectAmendedB();
   public boolean getCancelB();
   public void setPricingEnv(PricingEnv pricingEnv);
   public PricingEnv getPricingEnv();
   public BillingEvent cloneEvent();
   public Map<String, String> getTradeAttributes();
   public long getOccurrenceTime();
}
```

This CustomBillingEvent is created by a Custom BillingHandler.

6.14.2 Defining a Custom BillingHandler

The class below must be defined in the calypsox package. The custom code placed in buildCustomEvent(...) must create an instance of a Custom implementation of the BillingEvent Interface.

```
calypsox.engine.billing.BillingHandler extends com.calypso.engine.billing.BiilingHandle {
  public BillingEvent buildCustomEvent(PSEvent event) {
    // Custom code creating the BillingEvent
    BillingEvent billingEvent = .....
    return billingEvent;
    }
}
```

}Any Billing Event generated out of

7 Limits

7.1 Creating Custom Limit Types

Create a class named **tk.limit.<LimitType>Limit** that extends the abstract base class **com.calypso.tk.limit.BaseLimit**.

Register custom limit types in the **limitType** domain.

7.2 Excluding Trades from Limit Checking

You can now specify a rule that excludes trades from the limit check rather than using a trade filter or the limit exclude keyword.

The exclusion rule is implemented using the EXCLUSION limit type. It does not compute limit usages, but rather determines whether a given trade should be included or not. Using the EXCLUSION limit type does not replace either the trade filter or the limit exclude keyword, but instead provides a third, more customizable way to exclude a trade.

To specify an EXCLUSION type of limit, implement a class called **tk.limit.<product_type>EXCLUSIONLimit**, or **tk.limitEXCLUSIONLimit** that implements **com.calypso.tk.limit.BaseLimit**.

If no such EXCLUSIONLimit class is implemented, trades are only excluded by the trade filter or by using the limit exclude keyword.

8 Message Documents

Messages in Calypso are converted into documents prior to being physically sent out of the system. These documents may also be edited and stored in the database prior to being sent. By default, Calypso supports the following document formats: HTML, text, XML and SWIFT.

8.1 How to Create an HTML Template

Templates are supported for the HTML format. Calypso provides a standard set of document templates. You may wish to modify them or to create your own. Templates are associated with messages using **Main Entry -> Configuration -> Messages & Matching -> Message Set-up Configuration**.

HTML templates contain the text and the format of message documents such as confirmations, payment, or receipt advices or any other message document generated, based upon the Message Setup (refer to the *Calypso Messages User Guide* for details). Any information that is required from the trade, the message, or the transfer is marked as a keyword in the template. The MessageFormatter extracts the information from the trade and populates the template keywords. Conditional processing is also supported to allow for more flexibility in structuring templates. For example, common header and footer information may be kept in their own files and included in other templates. Sub-documents may also be included based upon conditions.

Calypso templates are located under `resources.com.calypso.templates`. Custom templates should be located under `resources.<custom package name>.templates`. A list of the keywords available for building your own message templates can be seen in **Main Entry -> Help -> Message Template Keywords**.

Keywords have the format `|keyword name|`. This section focuses on conditional keywords with the format: `<calypso>conditional keyword name</calypso>`.

Code Delimiters

Any code that is between the tags `<!--calypso></calypso-->` or `<calypso></calypso>` is interpreted. Note that there can be multiple sets of <calypso> tag pairs within a document.

All text outside these tags is ignored by the document parser and is treated as regular HTML. All text within these tags, however, must be syntactically correct and cannot include HTML tags. The text within the tag pair is parsed for special directives. HTML tags included in the tag pair will raise exception(s) unless they are included in an inline directive.

Logical Expressions

The following keywords are available in the language: if, else, include, set, inline, and iterator. They are described below.

Note that they are case sensitive.

The following syntax is used in this section:

- Whenever you see a definition that uses `<word>`, it means that the word expression is defined elsewhere.
- A `<statement>` can be any of the following: `<if statement>`, `<set statement>`, `<include statement>`, or `<inline statement>`. Hereafter if you see the text `<statement>`, you can substitute any of these expressions instead.
- `<statements>` is a succession of `<statement>`, typically separated with a semicolon, very much like in modern programming languages.

<if statement>

```
if ( <conditions> ) <statement>
```

— or —

```
if ( <conditions> ) { <statements> }
```

The start and end brackets are optional, but they are necessary if you wish to have multiple statements.

`<conditions>` enables you to chain various `<condition>` statements together using logical operators && (AND), || (OR), and ! (NOT). Hence, the following would be a valid set of conditions:

```
<condition> && ( <condition> || <condition> || ! <condition>)
```

It is also possible to parse a `|KEYWORD|` inside an "if" statement. For example, `if (|MASTERAG_NAME| == "ISDA" && |MASTERAG_SIGN| != "SIGNED")`, where `|MASTERAG_NAME|` and `|MASTERAG_SIGN|` are defined as keywords available from MessageFormatter.

Nested "if" statements are supported, and the "else" keyword can be added to provide a catch-all clause at the end.

<condition>

A Condition checks the value of an object attribute or the result of a method and compares it against a fixed, literal value. Certain values are directly returned from predefined objects. For more customized operations, an interface can be implemented so that a call can be made to a custom class.

The following objects are predefined: Message, Transfer, Trade, Product, Sender, and Receiver. So, for example, all the following would constitute valid condition statements:

```
Trade.quantity > 100000
Message.status = "CANCELED"
Transfer.isPayment() = true
Sender.lastName = "Johnson"
Product.getRateIndex() like "%LIBOR%"
```

As you can see, it is possible to directly query the field (quantity, status) or to make a method call on the related object (*isPayment()*, *getRateIndex()*). Examine the API reference for the related objects (**com.calypso.tk.core.Trade**, **com.calypso.tk.bo.BOTransfer**, **com.calypso.tk.bo.BOMessage**, **com.calypso.tk.core.Product**, and **com.calypso.tk.refdata.LEContact**) to determine the available methods.

You can also call custom functions as described in Section 8.1.1, "How to Create Custom Functions," on page 68, with the following syntax:

```
MyFunction("arg1", "arg2") > 0.75
```

As far as available comparisons, here are all the valid operators: **<, >, <=, >=, == (equals), != (not equal), like, in,** and **notin**. The like operator works identically to that found in SQL.

<set statement>

```
set KEYWORD = "value";
```

All the values for identifiers used in Set statements can be used as default values. If a keyword is undefined or its value cannot be extracted otherwise by MessageFormatter, the 'default' value could be retrieved from the 'set' directive.

Take this code snippet for example:

```
<calypso>
set HELLO="Bonjour";
</calypso>
...
<center>|HELLO|</center>
...
```

In this case, the HTML output for keyword HELLO will default to Bonjour unless it has been overridden elsewhere. For example, there could be a *parseHELLO()* method that does the job. In any event, this provides a convenient method to set default values for keywords inside the document. Note that set statements can also be used in Conditions:

```
<>
```

```
calypso>
if ( Message.language == "English" )
    set HELLO="Hello";
if ( Message.language == "French" )
    set HELLO="Bonjour";
</calypso
```

Also, you can use the set statement to store function results, as shown in the example below:

```
<!--calypso>
set TRADE_ID = Trade.getId();
set PRODUCT_TYPE = Trade.getProductType();
set CUSTOM_VALUE = MyCustomFunction("One", "Two", "Three");
```

```
</calypso-->
...
We are sending you this |PRODUCT_TYPE| Trade Confirmation for Trade ID |TRADE_ID|. Here is
the custom value:  |CUSTOM_VALUE|.
...
```

Note that to set CUSTOM_VALUE, the parser expects the class **`tk.bo.formatter.MyCustomFunction`** to exist. See Section 8.1.1, "How to Create Custom Functions," on page 68 for details.

<include statement>

An Include statement reads and inserts a text specified in the URL string into the generated document.

```
include "<url>";
```

<url> can be a filename "myfile.html" located in the template directory, or any valid URL (for example,http://www.mysite.com/myfile.html).

<inline statement>

An Inline statement inserts the text within quotation marks directly into the generated document.

```
inline "HTML text";
```

For example:

```
if ( Trade.quantity == 0 )
     inline "<b>Trade quantity is 0.</b>";
```

Note: You cannot escape the double quote character (") in an inline statement. For example, the following comment causes a parsing error:

```
Inline "This is a \"String\"";
```

The inline statement is only intended for several lines of text in any case.

<iterator statement>

You can define Iterators as in the example shown below.

```
<!--calypso>
iterator ( "CashFlow" )
   inline "
<tr>
   <td>|CASHFLOW_START_DATE|</td>
   <td>|CASHFLOW_END_DATE|</td>
   <td>|CASHFLOW_RATE|</td>
</tr>
   ";
</calypso-->
```

The following Iterators are already provided:

- BondCashFlow
- BondCallSchedule
- CashFlow

- CompoundPeriod
- Fee
- MessageGroup
- PayFee and ReceiveFee
- PayLegCompoundPeriod and ReceiveLegCompoundPeriod
- PayLegFlow and ReceiveLegFlow
- StructuredProduct

8.1.1 How to Create Custom Functions

Although Conditions can retrieve properties for the most commonly used objects (Product, Trade, Transfer, etc.), it is sometimes necessary to use a custom function derive the value.

Custom classes can be called directly from the FormatterParser if they are placed in the **calypsox.tk.bo.formatter** or the **com.calypso.tk.bo.formatter** packages and implementing the FormatterFunction interface. This interface defines one method:

```
public Object call(DSConnection dsCon,
  BOMessage message,
  BOTransfer transfer,
  Trade trade,
  LEContact sender,
  LEContact receiver,
  Vector args);
```

For example, a custom class named MyFunction is placed in the **calypsox.tk.bo.formatter** package, and this new class implements FormatterFunction.

The following code is inserted in the template file:

```
<calypso>
if ( MyFunction("one", 2) == true )
    include "subdocument";
</calypso>
```

FormatterParser will locate the class MyFunction and make a call to its call method as defined above. The args parameter will be a vector with 2 elements representing the arguments "one" and 2.

In this case, the semantics suggest that the *call()* method should return a Boolean object since we're comparing against a true value. Of course, there's no real way to check usage so this cannot be enforced.

Note: In the event of an error when checking a Condition, the returned result is False. For example, if your condition is performing a Boolean comparison and the returns a String, the Condition will return a false. Therefore, best practice suggests that you should only perform branching based on True evaluaton results. Using the NOT operator (!) can result in incorrect branching if an error is encountered during the comparison.

8.1.2 Creating a Custom Display in Document Manager

To provide Document Security, a Document must setup regions. If a document declares no regions, then it defaults to read-only, meaning that it cannot be modified. Regions are defined using the following tags in the HTML template:

```
<!--region:NAME--> ... <!--/region-->
```

This would define a region named NAME.

Once regions are defined, permissions can be set on the documents based on the region, using **Main Entry -> Configuration -> Messages & Matching -> Document Manager**. Refer to the *Calypso Messages User Guide* for details.

Calypso provides an API to customize HTML document display in the Document Manager.

Create a class named **tk.bo.document.CustomDocumentFilter** that implements **com.calypso.tk.bo.document.DocumentFilter**.

Sample Code

calypsox/tk/bo/document/CustomDocumentFilter.java

This sample display non-editable regions in gray.

8.2 How to Create SWIFT Messages

A standard set of SWIFT messages are provided by the Calypso system. SWIFT messages can be generated by product type and message type, and are selected using template names, or can be generated from XML templates.

8.3 How to Create Custom Import of Message Documents

Create a class named **tk.bo.document.CustomDocumentImporter** that implements **com.calypso.tk.bo.document.DocumentImporter**.

CustomDocumentImporter is invoked from **com.calypso.apps.reporting.MessageDocumentWindow**.

The default implementation is in **com.calypso.tk.bo.document.DefaultDocumentImporter**.

9 Market Data

9.1 Balance Engine

While the Balance Engine itself cannot be customized, two items are available to customize:

- Finding the closing account: Implement a tk.bo.accounting.CustomClosingAccountName implementing tk.bo.accounting.ClosingAccountName.

  The default behavior is to use the name of the account being closed, plus the year.
- The behavior when Balance Positions have been archived.

When balance positions are archived and a posting that falls into the archived range is detected, the default behavior is to *not* update the archived entries. An error will be logged. "Live" position total values should be correct but will not correspond to the sum of daily changes from archived positions.

If a **`tk.bo.CustomBalanceHistoryHandler`** implementing a **BalanceHistoryHandler** is found, it will be called and may update archived balance positions. If that handler returns true, no error is logged.

9.2 Quotes

9.2.1 How to use Quotes

QuoteSet is a repository for quote values that are used for pricing and curve generation. Typically, you would obtain a QuoteSet object from a given PricingEnv.

The following example illustrates how to use QuoteSet. Specifically, it shows how to obtain quote values for a given product and for the curve underlying instruments used by a given curve. Furthermore, it illustrates how to manipulate quote values (bumping the quotes by 1bp) and use the bumped quotes for curve generation.

Note: There are other methods in the QuoteSet class that are not used in the example. For example, there are methods specifically for getting FX quotes, rate index quotes, etc.

Sample Code

`samples/cookbook/UseQuoteSet.java`

9.2.2 How to Subscribe to real-time Quotes

The following example illustrates how to subscribe to real-time quotes. The sample program regenerates a given zero curve every few seconds using the latest real-time quotes.

Create a class that implements the **`com.calypso.tk.marketdata.FeedListener`** interface. The key method is the *newQuote()* method which is invoked whenever a real-time quote is available.

Tip

ⓘ Before running the program, ensure that you have the proper real-time feed configuration in the system. Refer to the *Calypso Market Data User Guide* for information on setting up a real-time feed.

Sample Code in `calypsox/apps/reporting/`

`FXPositionWindow.java`

9.2.3 How to Connect to a Custom Feed Source

Calypso provides an API to extend the system to support any real-time feed source. The following example demonstrates how to connect to a real-time feed source.

Overview of Steps

- Step 1 — Create a FeedHandler
- Step 2 — Register the new FeedHandler

Step 1 — Creating a FeedHandler

Create a class named **`tk.marketdata.<feed_name>FeedHandler`** that extends **`com.calypso.tk.marketdata.FeedHandler`**.

The FeedHandler is responsible for creating a physical connection to the real-time feed and providing a mapping between Calypso quote names and feed quote names based on the Feed Address Config.

<*feed_name*>FeedHandler is invoked from **`com.calypso.tk.marketdata.FeedHandler`**.

Sample Code in `calypsox/tk/marketdata/`

`SampleFeedHandler.java`

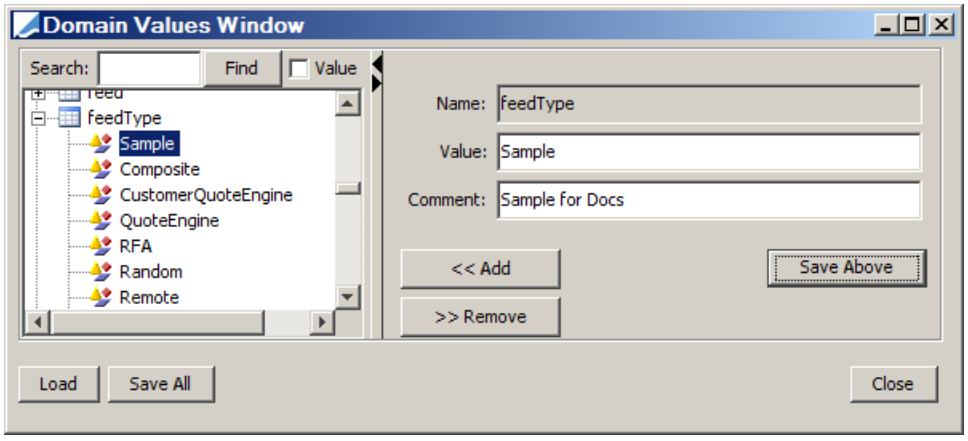
Sample code in `com/calypso/tk/marketdata/`

`RandomFeedHandler.java`

Step 2 — Registering the new FeedHandler

The feed must first be registered with the system before it can be used. To do so, add the feed name to the feedType domain.

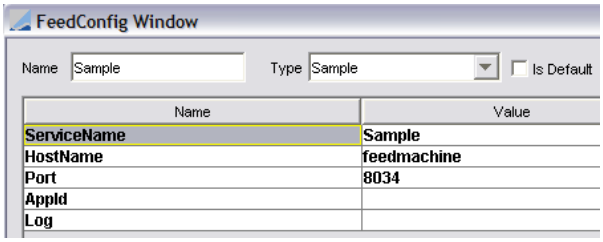
Figure 9-1: Domain Values Window — Registering a new FeedHandler



Then, you must configure the feed:

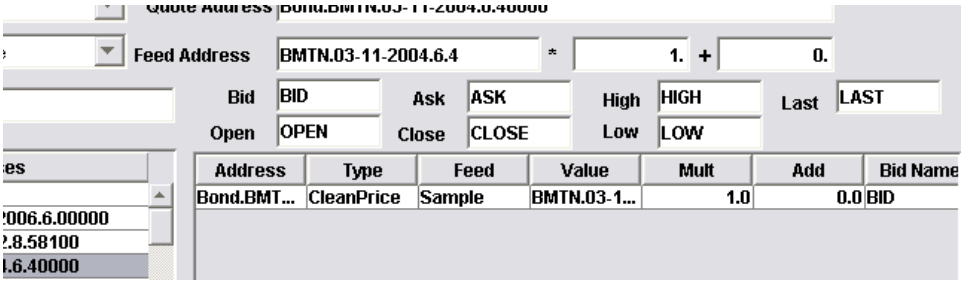
- Specify connection parameters using **Main Entry -> Configuration -> Market Data -> Feed Configuration** as shown in the example below. Check **Is Default** if you wish to use this feed as the default feed.

Figure 9-2: Feed Configuration



- Map Calypso quote names to the feed's quote names using **Main Entry -> Configuration -> Market Data -> Feed Address Mapping** as shown in the example below.

Figure 9-3: Feed Address Mapping



Refer to the *Calypso Market Data User Guide* for information on setting up a real-time feed.

You can inspect real-time quotes using **Main Entry -> Market Data -> Market Quotes -> Feed Quotes**.

9.3 Market Data Items

9.3.1 Creating a Custom Curve

Do the following to create a custom curve:

1. Create a class named **`tk.marketdata.<curve_type>`** that extends the abstract base class **`com.calypso.tk.marketdata.Curve`**.

   The <*curve_type*> class is invoked from **`com.calypso.tk.marketdata.Curve`**.

2. To make the curve persistent, create a class named **tk.marketdata.sql.<*curve_type*>SQL** that extends the abstract base class **com.calypso.tk.marketdata.sql.MarketDataItemSQL**.

   <*curve_type*>SQL is invoked from: **com.calypso.tk.marketdata.sql.MarketDataItemSQL**.

3. If the curve is persistent, create a database table and a corresponding stored procedure for storing the curve's instances.

4. Register the curve in the **marketDataType** domain.

9.3.2 How to Populate a Curve with Quotes

A sample program shows how to populate a curve with quotes: `samples/ImportCurveProbability.java`.

9.3.3 How to Create a Custom Volatility Surface

Do the following to create a custom volatility surface:

1. Create a class named **tk.marketdata.<*vol_surface_type*>** that extends the class, **com.calypso.tk.marketdata.VolatilitySurface3D**.

   The <*vol_surface_type*> class is invoked from **com.calypso.tk.marketdata.VolatilitySurface3D**.

2. To make the volatility surface persistent, create a class named **tk.marketdata.sql.<*vol_surface_type*>SQL** that extends the abstract base class **com.calypso.tk.marketdata.sql.MarketDataItemSQL**.

   <*vol_surface_type*>SQL is invoked from **com.calypso.tk.marketdata.sql.MarketDataItemSQL**.

3. If the volatility surface is persistent, create a database table and a corresponding stored procedure for storing the volatility surface's instances.

4. Register the volatility surface into the **marketDataType** domain.

9.3.4 How to make a Custom Market Data Item Available for Selection

You can make your market data item available in the market data selector for loading, saving and deleting.

Create a class named **apps.marketdata.<*market_data_type*>Selector** that implements the **com.calypso.apps.marketdata.MarketDataItemSelector** interface.

<*market_data_type*>Selector is invoked by **com.calypso.apps.marketdata.MarketDataUtil**.

9.3.5 How to Display a Custom Market Data Item

Create a class named **apps.marketdata.<*market_data_type*>Window** that implements the interface **com.calypso.apps.marketdata.MarketDataItemViewer**.

<*market_data_type*>Window is invoked from **com.calypso.apps.marketdata.MarketDataUtil**.

9.3.6 How to add a Custom Menu Item to a Curve Window

Create a class named **apps.marketdata.CustomCurveMenu<*name*>** that implements the interface **com.calypso.apps.marketdata.CustomCurveMenu**.

CustomCurveMenu<*name*> is invoked from the Curve windows.

9.3.7 How to add a Custom Menu Item to the VolatilitySurface3D Window

Create a class named **apps.marketdata.CustomVolSurfaceMenu<*generator_name*>** that implements the interface **com.calypso.apps.marketdata.CustomVolSurfaceMenu**.

CustomVolSurfaceMenu<*generator_name*> is invoked from **com.calypso.apps.marketdata.VolatilitySurface3DWindow**.

9.3.8 Creating a Custom Volatilty Surface Selector

Create a class named **apps.marketdata.VolatilitySurface3DSelector** that implements **com.calypso.apps.marketdata.VolatilitySurface3DSelector**. It is invoked from **com.calypso.apps.marketdata.VolatilitySurface3DWindow**.

9.4 Curve Generation

9.4.1 Creating a Custom Curve Interpolator

Create a class named **tk.core.<*name*>** that extends the abstract base class **com.calypso.tk.core.Interpolator**.

The <*name*> class is invoked from **com.calypso.tk.marketdata.Curve**.

Sample Code

calypsox/tk/core/InterpZeroDEMO.java

9.4.2 Creating a Custom Curve Generation Algorithm

The Calypso framework allows users to add new curve generation algorithms to the system. To allow for maximum flexibility, the framework allows additional input and output values to be added to the curve for curve generation. The system has built-in capabilities to save, retrieve, and display any new input and output value required. Specifically, the

system can accommodate the following extensions for input and output values:

- Input parameters
- Adjustments to quote values
- Adjustments to curve points

There are various ways in which adjustments to curve points can be used. For example, a pricer can use the adjustment values in pricing or a new curve point interpolator can use the adjustment values when interpolating curve points.

In this example we will create a generation algorithm for a zero curve which will make use of the input/output extensions. The curve generation algorithm will require alpha and beta for input parameters, correlation for quote value adjustment, and coefficient for curve point adjustment. The algorithm will create a curve point for each underlying instrument with value equals (1+alpha) multiplied by (1+beta) multipled by the quote value of underlying instrument. The coefficient for curve point is calculated by multiplying the quote value of underlying instrument by the correlation value for the quote.

Overview of Steps

- Step 1 — Create a CurveGenerator
- Step 2 — Register the new CurveGenerator

Step 1 — Creating a CurveGenerator

Create a class named **`tk.marketdata.CurveGenerator<name>`** which extends the abstract base class **`com.calypso.tk.marketdata.CurveGenerator`**.

Note: For zero curves, the CurveGenerator class should extend **`com.calypso.tk.marketdata.CurveGeneratorZero`**.

The *usesQuoteAdjustment()* method allows a curve generator to specify whether or not to display adjustment columns. The curve window does not display adjustment columns if the method returns false. The method returns true by default.

The *notifyChange()* method will regenerate the curve if the underlying has changed.

CurveGenerator<*name*> is invoked from **`com.calypso.tk.marketdata.Curve`**.

Sample Code in `calypsox/tk/marketdata/`

- `CurveGeneratorDEMO.java`
- `CurveGeneratorDEMO_2.java`
- `CurveGeneratorDEMO_3.java`

Sample Code in `calypsox/tk/marketdata/`

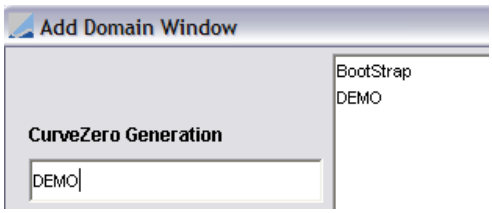
The following samples show how to generate a curve on the server side:

- CurveGeneratorRemote.java
- RemoteGenerate.java
- CurveGeneratorDispRemote.java

Step 2 — Registering the new CurveGenerator

To register the new CurveGenerator, click the ... button next to the **Generation Alg** field in the Curve window where you want this CurveGenerator, then add the CurveGenerator in the *Add Domain* window, as shown below.

Figure 9-4: Add Domain — Registering a New CurveGenerator



The new CurveGenerator will be available for selection, and the input parameters will appear under the Quotes panel of the Curve Window as applicable.

When the curve is generated, the output contains the point adjustment coefficient under the Points panel of the Curve Window as applicable.

9.4.3 Making Generator Parameters Persistent

Create a class named **tk.marketdata.sql.CurveGenerator<*name*>SQL** that implements **com.calypso.tk.marketdata.sql.CurveGeneratorSQL**.

CurveGenerator<*name*>SQL is invoked from **com.calypso.tk.marketdata.sql.CurveSQL**.

Sample Code in calypsox/tk/marketdata/

- CurveGeneratorDerivedTstData.java
- sql/CurveGeneratorDerivedTstSQL.java

9.4.4 Displaying Generator Parameters in a Popup Window

Note: This also applies to generator parameters for volatility surface.

The user can launch a GeneratorParameter window by double-clicking in the Parameters table of the Curve window or Volatility Surface window under the Quotes panel.

Create a class named **apps.marketdata.GeneratorParameter<parameter_name>** that implements the interface **com.calypso.apps.marketdata.GeneratorParameter**.

GeneratorParameter<*parameter_name*> is invoked from **com.calypso.apps.marketdata.GeneratorParameterUtil**.

9.4.5 Creating a Custom Curve Underlying Instrument

Do the following to create a custom curve underlying instrument:

1. Create a class named **tk.marketdata.CurveUnderlying<*instrument_type*>** that extends the abstract base class **com.calypso.tk.marketdata.CurveUnderlying**.

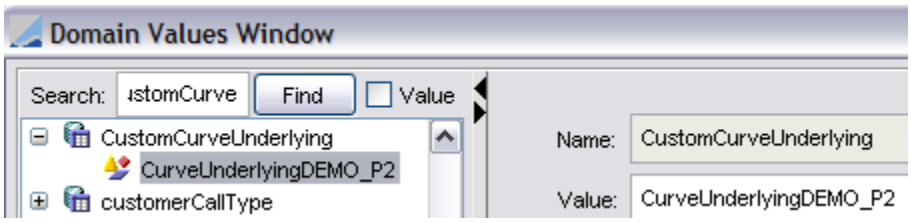
   CurveUnderlying<*instrument_type*> is invoked from **com.calypso.tk.marketdata.CurveUnderlying**.

2. To make the curve underlying instrument persistent, create a class named **tk.marketdata.sql.CurveUnderlying<*instrument_type*>SQL** that extends the class **com.calypso.tk.marketdata.CUSQL**.

   CurveUnderlying<*instrument_type*>SQL will be invoked from **com.calypso.tk.marketdata.sql.CurveUnderlyingSQL**.

3. Register the new curve underlying instrument in the CustomCurveUnderlying domain.

Figure 9-5: Domain Values Window — Registering a New Curve Underlying Instrument



Sample Code in `calypsox/tk/marketdata/`

CurveUnderlying samples:

- `CurveUnderlyingDEMO_P1.java`
- `CurveUnderlyingDEMO_P2.java`

CUSQL samples:

- `sql/CurveUnderlyingDEMO_P1SQL.java`
- `CurveUnderlyingDEMO_P2SQL.java`

9.4.6 Displaying a Custom Curve Underlying Instrument

Create a class named **apps.marketdata.CU<*instrument_type*>Panel** which implements the interface **com.calypso.apps.marketdata.CUPanel**.

CU<*instrument_type*>Panel is invoked from **com.calypso.apps.marketdata.CUWindow**, which displays a panel for the new curve underlying instrument.

Sample Code in calypsox/apps/marketdata/

- CUDEMO_P1Panel.java

Sample Code in calypsox/tk/marketdata/

- CUDEMO_P2Panel.java

9.4.7 Using a Custom Curve Underlying Instrument for Curve Generation

Sample Code in calypsox/tk/marketdata/

- CurveGeneratorDEMO-2.java

DEMO-2 is the a curve generation algorithm that supports the curve underlying instrument DEMO_P2. CurveGeneratorDEMO-2 extends CurveGeneratorDEMO, created above, and redefines the method *getCurveUnderlyingTypes()*. When the CurveGenerator DEMO-2 is selected, DEMO_P2 will be available as underlying instrument.

9.5 Volatility Surface Generation

9.5.1 Creating a Custom Volatility Surface Interpolator

Create a class named **tk.core.<*interpolator_name*>** which extends the abstract base class **com.calypso.tk.core.Interpolator3D**.

The **<*interpolator_name*>** class is invoked from **com.calypso.tk.marketdata.MarketDataSurface** and **com.calypso.tk.marketdata.VolatilitySurface3D**.

9.5.2 Creating a Custom Volatility Surface Generation Algorithm

Overview of Steps

- Step 1 — Create a VolSurfaceGenerator
- Step 2 — Register the new VolSurfaceGenerator

Step 1 — Creating a VolSurfaceGenerator

Create a class named **tk.marketdata.VolSurfaceGen<*name*>** that extends the abstract base class **com.calypso.tk.marketdata.VolSurfaceGenerator**.

VolSurfaceGen<*name*> is invoked from **com.calypso.tk.marketdata.MarketDataSurface** and **com.calypso.tk.marketdata.VolatilitySurface3D**.

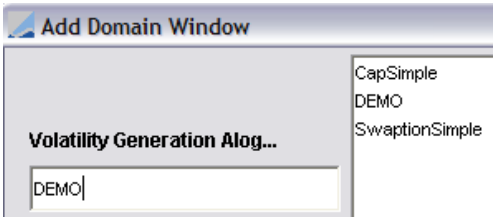
Sample Code in calypsox/tk/marketdata/

- VolSurfaceGenDEMO.java
- VolSurfaceGenDEMO_2.java
- VolSurfaceGenVega.java

Step 2 — Registering the New VolSurfaceGenerator

To register the new VolSurfaceGenerator, click the ... button next to the Generation field in the Volatility Surface window where you want this VolSurfaceGenerator, and add the VolSurfaceGenerator in the Add Domain window as shown below.

Figure 9-6: Add Domain Window — Registering the VolSurfaceGenerator



The new VolSurfaceGenerator will be available for selection.

Note that for FX volatility surface generators, derived generators are registered in the domain "FXVolSurfaceGenerator," and simple generators are registered in the domain "FXVolSurface.gensimple."

9.5.3 Making Generator Parameters Persistent

Create a class named **`tk.marketdata.sql.VolSurfaceGenerator<name>SQL`** that implements **`com.calypso.tk.marketdata.sql.VolSurfaceGeneratorSQL`**.

VolSurfaceGenerator<*name*>SQL is invoked from **`com.calypso.tk.marketdata.sql.VolatilitySurface3DSQL`**.

Sample Code in `calypsox/tk/marketdata/`

- `sql/VolSurfaceGeneratorCapCurveSQL.java`
- `VSGenCapCurveData.java`

9.5.4 Displaying Generator Parameters in a Popup Window

See Section 9.5.4, "Displaying Generator Parameters in a Popup Window," on page 79.

9.5.5 Creating a Custom Volatility Surface Underlying Instrument

Do the following for creating a custom volatility surface underlying instrument:

1. Create a class named **`tk.marketdata.VolSurfaceUnderlying<instrument_type>`** that extends the abstract base class **`com.calypso.tk.marketdata.VolSurfaceUnderlying`**.

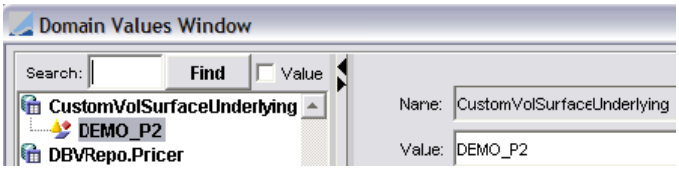
   VolSurfaceUnderlying<*instrument_type*> is invoked from **`com.calypso.tk.marketdata.VolSurfaceUnderlying`**.

2. To make the volatility surface underlying instrument persistent, create a class named **`tk.marketdata.sql.VolSurfaceUnderlying<instrument_type>SQL`** that extends the class **`com.calypso.tk.marketdata.sql.VolUSQL`**.

   VolSurfaceUnderlying<*instrument_type*>SQL is invoked from **`com.calypso.tk.marketdata.sql.VolSurfaceUnderlyingSQL`**.

3. Register the new volatility surface underlying instrument in the **CustomVolSurfaceUnderlying** domain.

Figure 9-7: Domain Values Window — Registering a New Volitilty Surface Underlying



Sample Code in `calypsox/tk/marketdata/`

VolSurfaceUnderlying sample:

`VolSurfaceUnderlyingDEMO_P2.java`

VolSurfaceUnderlyingSQL sample:

`sql/VolSurfaceUnderlyingDEMO_P2SQL.java`

9.5.6 Displaying a Custom Volatility Surface Underlying Instrument

Create a class named **`apps.marketdata.VolUnderlying<instrument_type>Panel`** that implements the interface **`com.calypso.apps.marketdata.VolUnderlyingPanel`**.

VolUnderlying<*instrument_type*>Panel is invoked from **`com.calypso.apps.marketdata.VolUnderlyingWindow`**, which displays a panel for the new volatility surface underlying instrument.

Sample Code in `calypsox/apps/marketdata/`

`VolUnderlyingDEMO_P2Panel.java`

9.5.7 Creating a Custom Volatility Type

Create a class named **`tk.marketdata.VolatilityType<name>`** that implements **`com.calypso.tk.marketdata.VolatilityType`**.

VolatilityType<*name*> is invoked from **`com.calypso.tk.marketdata.VolatilityType`**.

9.5.8 Creating a Custom Correlation Type

Create a class named **`tk.marketdata.CorrelationType<name>`** that extends the abstract base class **`com.calypso.tk.marketdata.CorrelationType`**.

CorrelationType<*name*> is invoked from
com.calypso.tk.marketdata.CorrelationType.

9.5.9 Creating Custom Selection Criteria for Filter Sets

Create a class named **tk.marketdata.CustomFilter** that implements the interface **com.calypso.tk.marketdata.CustomFilterInterface**.

This class will be invoked from
com.calypso.tk.marketdata.FilterElement.

Sample Code in calypsox/tk/marketdata/

CustomFilter.java

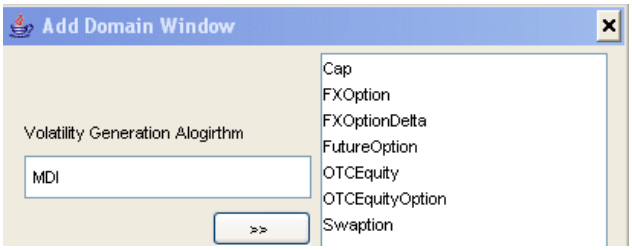
This sample adds trade maturity date as a range between two tenors.

9.5.10 Storing Underlying Market Data with a Volatility Surface

Write a custom generator in calypsox which implements the methods *getMDIParameterNames* and *getMDIParameterType(String s)*.

Restart Main Entry and, in the Volatility Surface window, register the custom generator by clicking the ... button next to the Generation field in the Volatility Surface window where you want this Generator, and add MDI in the Add Domain window:

Figure 9-8: Add Domain Window — Selecting a Volitility Generation Algorithm



Sample Code in calypsox/tk/marketdata/

VolSurfaceGenMDI.java

9.6 Pricer Configuration

9.6.1 Extending the Pricer Config Custom Panel

Create a class named **tk.product.<*product_type*>MDataSelector** that implements the interface
com.calypso.tk.product.ProductMDataSelector.

<*product_type*>MDataSelector is invoked from
com.calypso.apps.marketdata.PCProductSpecificMDPanel2.

Sample Code in calypsox/tk/product/

BondMDataSelector.java

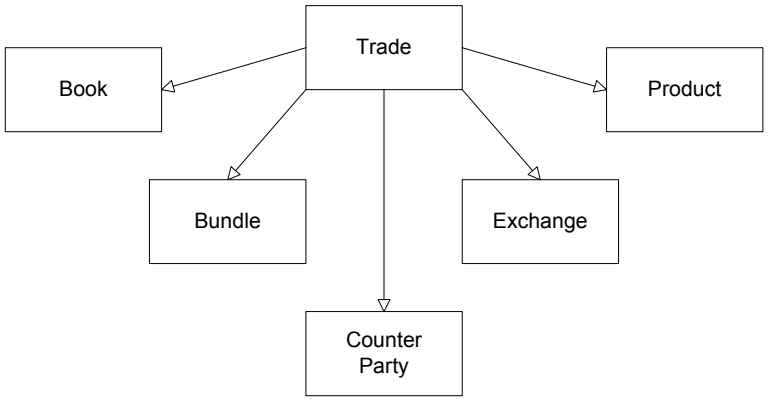
SwapMDataSelector.java

10 Product and Trade

10.1 Trade Object

The trade object references five other objects: the trading book, the bundle the trade belongs to (if any), the counterparty to the trade, the exchange the product is traded on (if any), and the product being traded. There are some aspects to every trade and are common across all products, these details are left on the trade object itself.

Figure 10-1: Tables Referenced by the Trade Object



Details common to most trades, and which are also included on the trade object itself, are: **ID**, **Status**, **Trade Date**, **Settle Date**, **Quantity**, **Negotiated Price**, **Negotiated Price Type**, **Trade Currency**, **Settle Currency**, **Trader Name**, **Salesperson Name**, **Comment**, **Keywords**, and **Fees**.

These common trade details make up the common components of a trade screen shared across products. The example below is of a CDS Index trade screen:

Figure 10-2: Trade Window with CDS Index

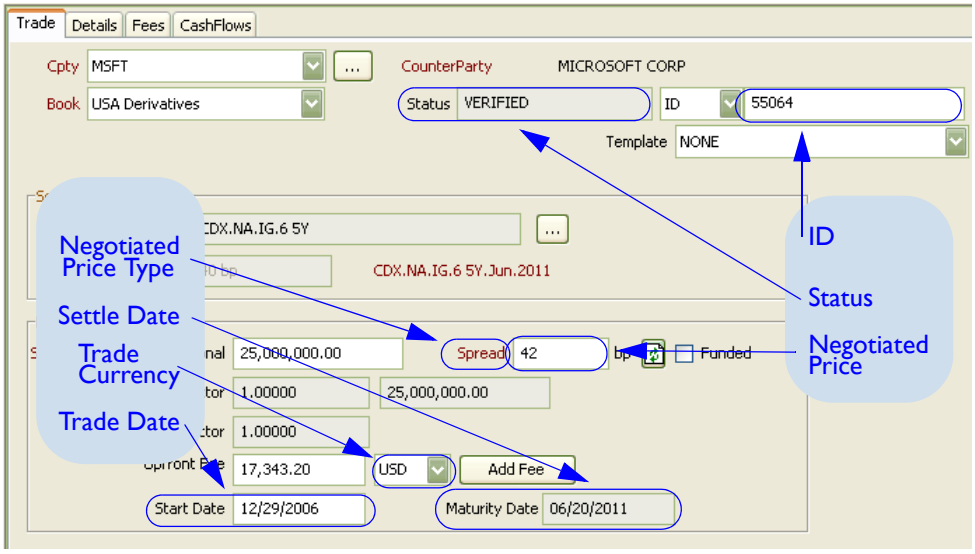
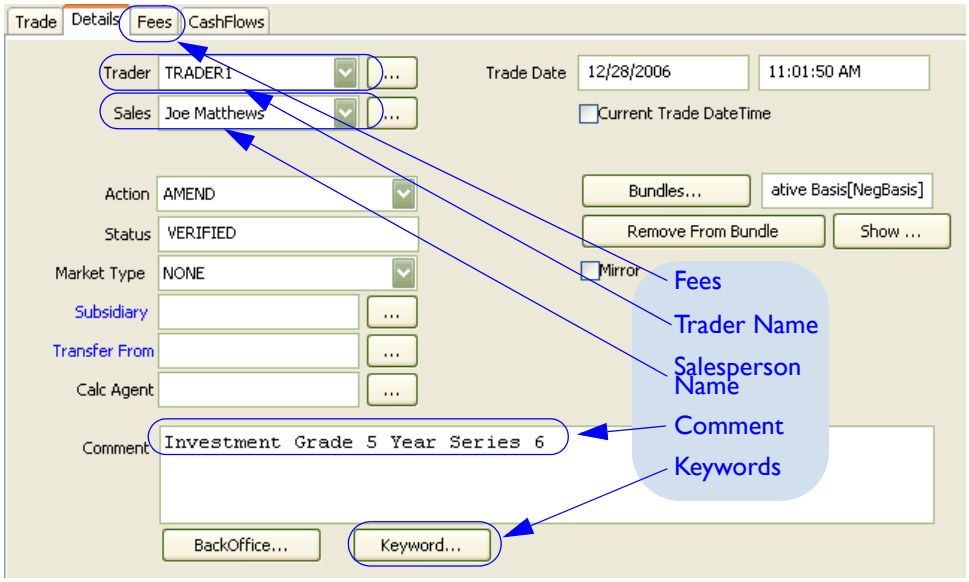


Figure 10-3: Trade Window Details Tab



10.2 Product Object

A financial product is any instrument that can be traded. This includes contracts that are traded multiple times, such as bonds, futures con-

tracts, and stocks, and it also includes one-off deals that are structured to meet client requirements and traded only once, like an interest rate swap or cap.

Every product must extend the base abstract Product class and any sub-class of the Product class must override its abstract methods.

10.2.1 Abstract Methods

The product object has four abstract methods to be overwritten: *getPrincipal()*, *getProductClass()*, *getProductFamily()*, and *hasSecondaryMarket()*. The first method returns the principal or notional of the product being traded. The second two methods *getProductClass()* and *getProductFamily()* simply return Strings of the type of product and product family this product belongs to. The last method returns a Boolean indicating whether or not this product is multiply traded or not. A return value of false would indicate that this instrument is a one-off deal like an interest rate swap or cap.

10.2.2 Public Methods

There are several methods already available to any product extending the Product class. Some should be overridden if such a method makes sense for your product and a few methods shouldn't be overridden as it may interfere with the functionality of the system. In addition to these methods, there are several interfaces which when implemented will add functionality to certain categories of products. These interfaces are covered in the next section.

The methods that shouldn't be overridden in the Product class are the *getName()* and *getType()* methods. These methods are used internally for referring to this object. The g*etName()* method will call the *getDescription()* method (which should be overridden) and the *getType()* returns the class name.

The other methods in the Product class should be overridden only if it makes sense for your particular product. One method that should always be overridden is the *getDescription()* method. This returns a String that describes the product being traded along with its relevant details. A good description would contain the more salient details of the product.

More methods that might make sense to override for a given product are: *getCurrency()*, *getPutCall()*, *getStrike()*, *getMaturityDate()*, *getRateIndex()*, *getSubType()*, and the cash flow generation methods *generateFlows()* and *calculate()*.

10.2.3 Cashflows

Some products can be realized as a series of cash flows between parties, such as bond coupon payments or simple interest payments. For those products that have cash flows, each product is responsible for generating its own cash flows. This is accomplished through the *generateFlows(JDate)* method and once the cash flow dates have been

created all known data such as saved quotes will be filled in by the *calculate(CashFlowSet, PricingEnv, JDate)* method on the product.

10.3 Product Interfaces

There are many public interfaces in the package `com.calypso.tk.product` available for use in the product objects. Each interface models a specific characteristic in a financial derivative. A sample of available interfaces is listed below:

Table 10-1: Product Interfaces

Product	Description
CallablePuttable	All products having optionality should implement this interface.
CashFlowGenerationBased	All products that require cash flow generation must implement this interface. This interface is used by the CashFlowGenerator for cash flow generation.
CollateralBased	All products that have collaterals must implement this interface. This interface is used by the reports.
CreditEventBased	All product classes that can be impacted by credit events, such as credit default swaps, should implement this interface.
CreditRisky	Credit risky products should implement this interface.
EventTypeActionBased	Any product that will maintain a schedule of EventTypeActions should implement this interface. Each EventTypeAction represents a change to the product's parameters that is effective at a given time.
Exercisable	All options must implement this interface.
FIRollOver	This class defines how a product should implement RollOver.
ForexRollOver	This class defines how an FX product should implement RollOver.
Option	All option products must implement this interface.
RateResetBased	All products that must handle special actions during a change in the rate reset must implement this interface.

10.4 Writing a New Product

Writing a new product amounts to extending the Product class, implementing any required interfaces, and defining any necessary variables specific to the product. Choose which methods to override from the Product class and create getters and setters for the new variables.

10.4.1 Example: Weather Derivatives

Weather derivatives are financial instruments that can be used by organizations or individuals as part of a risk management strategy to reduce risk associated with adverse or unexpected weather conditions. The dif-

ference from other derivatives is that the underlying asset (rain/temperature/snow) has no direct value to price the weather derivative. Farmers can use weather derivatives to hedge against poor harvests caused by drought or frost; theme parks may want to insure against rainy weekends during peak summer seasons; and gas and power companies may use heating degree days (HDD) or cooling degree days (CDD) contracts to smooth earnings.

Heating Degree Days/Cooling Degree Days are one of the most common types of weather derivative. Typical terms for an HDD contract would be: for the November to March period, for each day where the temperature falls below 18 degrees Celsius (or 65 degrees Fahrenheit) keep a cumulative count. Depending upon whether the option is a put option or a call option, pay out a set amount per heating degree day that the actual count differs from the strike (usually $20 per unit).

As an example, suppose we have purchased a thirty day HDD contract at 65 degrees Fahrenheit. During those thirty days each day average temperature is 50 degrees Fahrenheit. Then at the end of the thirty day term we are entitled to a payout of (15 * 30) $20 = $9,000.

There is no Weather Derivative product currently in Calypso, so if we were to make one we would need the following variables: a Start Date, End Date, the strike Temperature, a toggle whether this temperature is in Celsius or Fahrenheit, also a toggle to indicate whether or not this is an HDD or CDD contract, and the location the temperature is to be measured from. In addition we will make the payout equal to the notional times the number heating or cooling units so we will need a variable for the notional and the currency of the notional.

The following section contains an outline to create this product in Calypso.

10.4.2 Exercise: Write a New Product

Goal — You are going to create a new financial product to Calypso

Prerequisite: Completion of the CashFlowLookBack code.

Refer to the sample code **`api.examples.tk.product.HDDCDD`**.

Task 1: Create the necessary variables and methods.

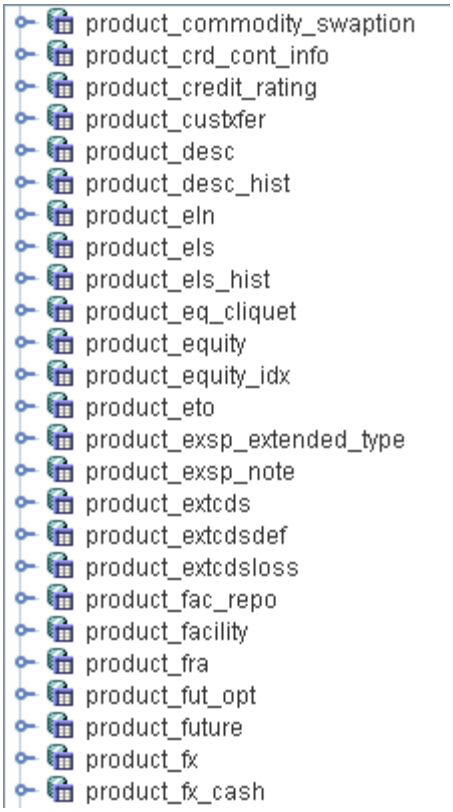
Table 10-2: Variables and Methods for the Weather Derivative Example

Step	Description
1	Extend the Product Class and create the following variables of the appropriate types: • startDate • maturityDate • notional • currency • temperature • fahrenheit_b (boolean) • hdd_b (boolean) • location
2	Create public getters and setters for all of these variables. If you are using an IDE such as Eclipse, there are tools to automatically generate these methods.
3	Override the abstract methods from the Product class: **getPrincipal**(), **getProductClass**(), **getProductFamily**(), and **hasSecondaryMarket**(). In the **getPrincipal**() method, return the notional variable. In the second two methods, return the static string *HDDCDD* for the product class and *WEATHER_DERIVATIVE* for the product family. Return the boolean false for the last method.
4	Override the **getDescription**() method from the Product class. Create a custom String to return that is a descriptive combination of the variables on the product. One example: CDD.50 F.San Francisco.12/19/2006-01/08/2007

10.5 Persistence

10.5.1 Extending the Data Model

All products, whether they are multiply-traded or singly-traded, have their own table in the database with the prefix "**`product_.`**"



A list of database tables corresponding to a few products

You must created a new table to store the variables on the new product. Every product table must have a **product_id** column that is set to the primary key for this table.

10.5.2 Example: Heating Degree Days/Cooling Degree Days

In the case of our Heating Degree Days/Cooling Degree Days we created create a new product table and named it **product_hdd_cdd**. The new table will persist the variables we created in our product, along with the Product ID to identify this product. Those variables are Product ID (always), Start Date, End Date, Notional, Currency, Temperature, Fahrenheit boolean, HDD boolean, and Location.

These columns should have the appropriate type: int, datetime, double precision, char, bit, etc. and all should be NOT NULL. The following exercise is an SQL sample of adding this table to a Sybase database.

10.5.3 Exercise: Add a Product Table to the Database

Goal — You are going to add a table to the Database.

Refer to the sample code WeatherDer.sql.

Table 10-3: Use ExecuteSQL to Add a Table to Your Database

Step	Description

2	Click on the ExecuteSQL window and paste the following text from the WeatherDer.sql file into the Query window.
	`CREATE TABLE product_hdd_cdd`
	`(product_id int NOT NULL,`
	`start_date datetime NOT NULL,`
	`maturity_date datetime NOT NULL,`
	`notional double precision NOT NULL,`
	`currency char(3) NOT NULL,`
	`temperature double precision NOT NULL,`
	`fahrenheit_b bit NOT NULL,`
	`hdd_b bit NOT NULL,`
	`location varchar(32) NOT NULL,`
	`CONSTRAINT ct_primarykey PRIMARY KEY CLUSTERED`
	`(product_id))`

3	Click Execute.
4	Verify that the table has been created by executing the following SQL statement: SELECT * FROM product_hdd_cdd

10.5.4 Writing a Persistence Class

Each new product needs a matching SQL class to persist it into the database. The SQL class follows the naming convention of **<*product_name*>SQL** and extends the **ProductSQL** class. Any class

that extends the SQL class must implement the *insert()*, *remove()*, *save()*, and *getAll()* methods in the **ProductSQL** class.

Figure 10-4: Extending ProductSQL

```
public class SampleSQL extends ProductSQL {

    protected boolean insert(Product prod, Connection con)
            throws PersistenceException, DeadLockException {
        return false;
    }

    protected boolean remove(Product prod, Connection con)
            throws PersistenceException, DeadLockException {
        return false;
    }

    protected boolean save(Product prod, Connection con)
            throws PersistenceException, DeadLockException {
        return false;
    }

    public Collection getAll(String from, String whereClause, Connection con,
            boolean joinWithProductDesc, boolean isDistinct) throws PersistenceException {
        return null;

    }
```

When saving the product details to the product table you must also save the product description in the **product_desc** table. There are methods built in to the **ProductSQL** class for performing this operation, ***updateDescription***() and ***saveDescription***(), but you must call the appropriate method on an update or insert.

SQL Object Persistor

To safely handle the task of persisting objects in the database, utility methods have been written to handle these tasks properly. This is achieved by creating a private class within the <*product*>SQL class that extends from the SQLObjectPersistor class. There are two methods which perform complementary functions: one creates a Calypso product from the database and the other sets database parameters from the Calypso product. The order of these parameters is dictated by a prepared SQL statement defined as a static String.

See the example below:

```
private class SampleLoader extends SQLObjectPersistor {

    private static final String SELECT = "SELECT product_wea
    private static final String INSERT = "INSERT INTO produc
    private static final String UPDATE = "UPDATE product_wea
    private static final String DELETE = "DELETE FROM produc

    /**
     * handle one row, and return the object created from th
     */
    public Object buildObjectFromResultSet(JResultSet rs) th
        Swap product = new Swap();
        return product;

    }

    public void setParametersFromObject(Object obj, Preparec
        WeatherDerivative ref = (WeatherDerivative)obj;

        int j = 1;
        setParameter(st, ref.getStartDate(), j++);
        setParameter(st, ref.getMaturityDate(), j++);

    }
}
```

The **SQLObjectPersistor** has its own useful methods for reading, updating, and saving to the database: ***listFromDB***(), ***saveTODB***(), and ***updateDB***() which all take a connection to the database and an SQL statement as parameters.

Example: Heating Degree Days/Cooling Degree Days

In the running example of the Heating Degree Days/Cooling Degree Days, we must write SQL statements to read from and write to the **product_hdd_cdd** database table. Making use of the SQLObjectPersistor class, we must complete the prepared statements SELECT, INSERT, UPDATE, and DELETE. Then, describe how to build the object from a result set or create a result set from the product object.

Because we must also modify the **product_desc** table, we use the methods save*Description()* and *updateDescription()* of the ProductSQL class. After constructing the appropriate prepared SQL statement, call either *listFromDB()*, *saveToDB()*, or *updateDB()* methods.

Exercise: Write a Persistence Class

Goal — You are going to create a persistence class for a financial product in Calypso

» Prerequisite: Completion of the HDDCDD code.

» Refer to the sample code:
`api.examples.tk.product.sql.HDDCDDSQL`.

Table 10-4: Create the Necessary Prepared Statements in the HDDCDDLoader Class

Step	Description
1	Create the SQL SELECT statement which selects each column in the table but where the product_id is the last column selected. For example: SELECT product_hdd_cdd.start_date, … etc.… , product_hdd_cdd.product_id FROM product_hdd_cdd
2	Create the prepared SQL INSERT statement using the *exact* same order of items as in the SELECT statement above. For example: INSERT INTO product_hdd_cdd (start_date, … etc. …, product_id) VALUES (?,?,?,?,?,?,?,?,?,?)
3	Create the prepared SQL UPDATE statement using the *exact* same order of items as in the SELECT statement above. For example: UPDATE product_hdd_cdd SET start_date=?, … etc. … WHERE product_id=?

Table 10-5: Complete the Methods in the HDDCDDLoader Class

Step	Description

4	The `buildObjectFromResultSet(JResultSet)` method constructs the HDDCDD product from the result set. The items in the result set will be returned in the *exact* order of the SELECT statement in Step 1.
	The ResultSet has get methods for each Calypso data type to return the necessary data type for the product.
	For example:
	int j = 1;
	ref.setStartDate(rs.getJDate(j++));
	ref.setMaturityDate(rs.getJDate(j++));
	ref.setNotional(rs.getDouble(j++));

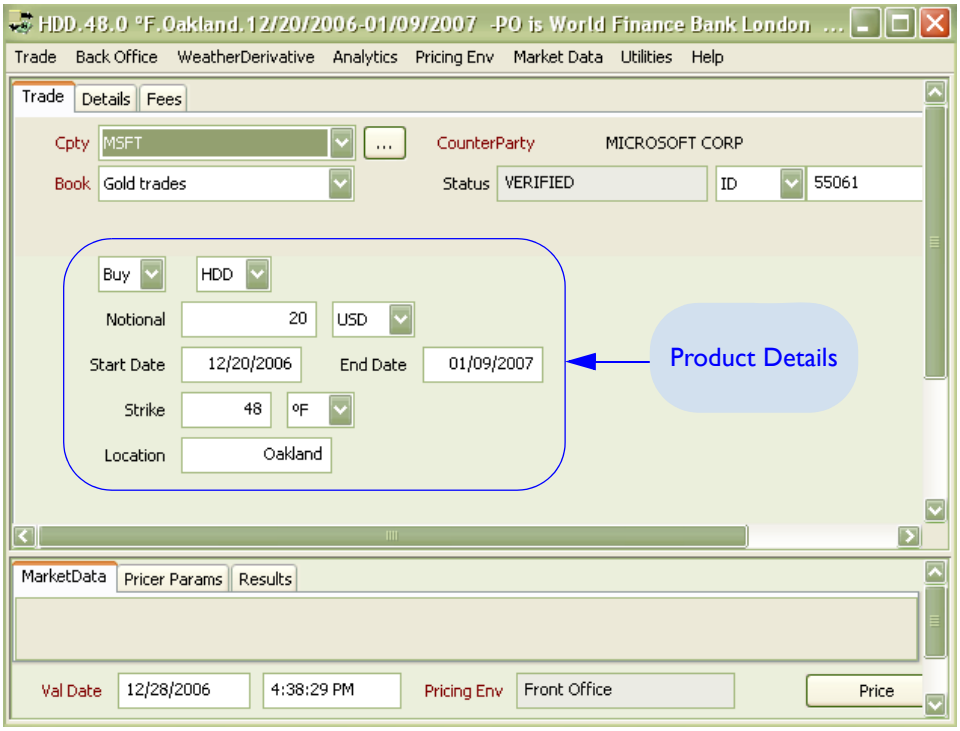
5	The `setParametersFromObject(Object, PreparedStatement)` method set the parameters for the prepared statement from the variables of the product object. The setParameter() method is used to set each parameter one by one. It is overloaded to take multiple Java data types.
	For example:
	int j = 1;
	setParameter(st, ref.getStartDate(), j++);
	setParameter(st, ref.getMaturityDate(), j++);
	setParameter(st, ref.getNotional(), j++);

10.6 Trade Window

10.6.1 Product Panel

The final step in adding a new product is to create a window to display the data. In the Calypso trade window, only one panel contains the product details and this component is the only class that must be written.

Figure 10-5: Trade Window



Trade Product Panel, which is is invoked from the Trade Window, is for displays all the product details in the Trade Window. There are three steps to adding a new window:

1. First , create a class named `apps.trading.Trade<product_type>Window` that extends the TradeWindowBase class. This new class calls the super-class's constructor from its public constructor.
2. Second, add this class to the MainEntry window: From MainEntry choose **Utilities -> MainEntry Configurator** and add the action **trading.Trade<*product_type*>Window** to the Trade sub-menu.
3. Finally, create a class named `apps.trading.<product_type>TradeProductPanel` that implements the `TradeProductPanel` interface.

10.6.2 Public Methods

The main methods in the **TradeProductPanel** are *buildTrade()*, *newTrade()*, *setDefaults()*, and *showTrade()*. These methods are responsible for creating new trade display, setting the default fields, synchronizing the trade object with what is displayed, and conversely synchronizing the display with a loaded trade and product object.

The other responsibility of this class is to paint the necessary components displayed to the user.

10.6.3 Example: Heating Degree Days/Cooling Degree Days

In the Trade Window for the Heating Degree Days/Cooling Degree Days, we must write the GUI components as they are seen Figure 10-5, "Trade Window" on page 93 . Ensure that all of the variables created on our product can be set and displayed through the GUI.

Once all the necessary components have been created: Buy/Sell, HDD/CDD, Notional, Start Date, End Date, Strike, Fahrenheit/Celsius, and Location, the *buildTrade()* add *showTrade()* methods can then be used to synch the display with the product object.

10.6.4 Exercise: Add a Product Container to the Trade Window

Goal — You are going to write the components in the product container GUI of the Trade Window.

Prerequisite:

- Completion of the HDDCDD code.

Refer to the sample code:
`api.examples.apps.product.HDDCDDTradeProductPanel`.

Table 10-6: Task 1: Create the Necessary Combo Boxes, Labels, and Text boxes to Display the Data

Step	Description
1	Create a **CalypsoComboBox** for Buy/Sell, HDD/CDD, temperature degree units, and the currency selection.
2	Create a **JLabel** for the Notional, Start Date, End Date, Strike, and Location.
3	Create a **JTextField** for the Notional, Start Date, End Date, Strike, and Location.

Figure 10-6: Example Product Container GUI Layput Specificiations

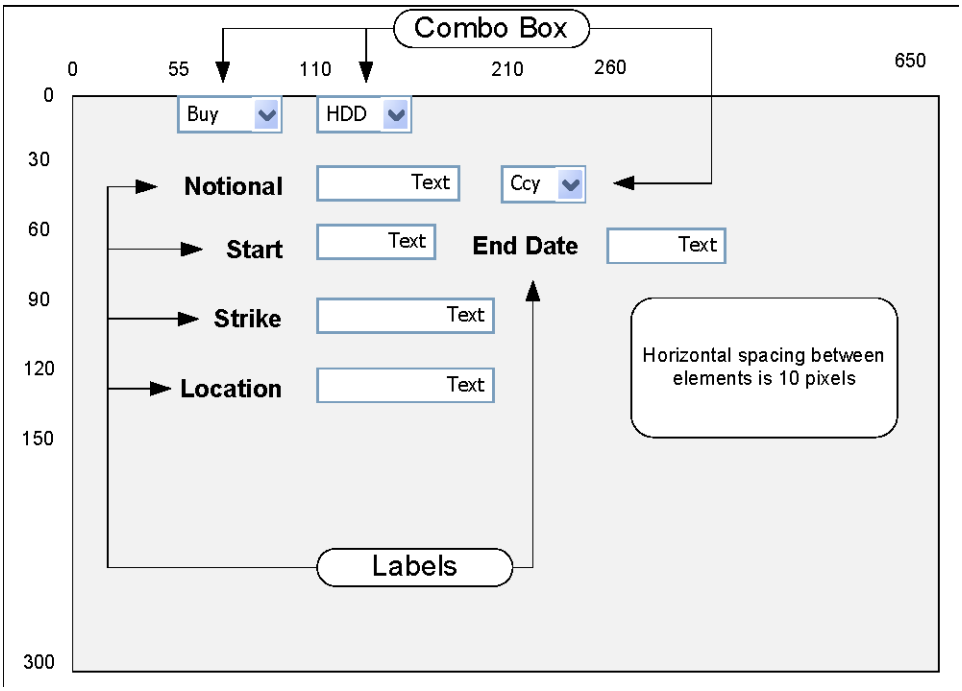


Table 10-7: Task 2: Define the Layout

Step	Description
4	Using the diagram above as a guide, make the components with the appropriate labels and bounds. Note that all of the **JLabels** and **JTextFields** should be right-justified**.(JTextField.*RIGHT*, SwingConstants.*RIGHT*)**. Complete the private methods in the sample code to make the components.
5	In the case of the Combo Boxes, one can use the utility methods in AppUtil to set the lists of the Combo Boxes. For example: AppUtil.*set*(CalypsoComboBox, Vector) AppUtil.*setToDomain*(CalypsoComboBox, DomainName)

Table 10-8: Task 3: Complete the Constructor and initDomains()

Step	Description
6	Add to this product container each of the components defined in the previous steps.
7	Next, call **initDomains**() from within the Constructor.
8	In the **initDomains**() method: Use the **AppUtil** to add Choice, Date, and Number listeners. For the End Date use the method ***addDateListener*(JTextField, JTextField startDate)** – this allows such keyboard shortcuts for the end date as "10d" or "3m" which will compute 10 days or 3 months from the start date, respectively.

Table 10-9: Task 4: newTrade() and serDefaults()

Step	Description
9	The **newTrade**() method is called when the window is opened for the first time and is used to create defaults for the screen. Once the defaults have been set we call the **buildTrade**() method.
10	In the **setDefaults**() method, set the following defaults for the screen: **Buy/Sell Drop-Down**: Buy **Heating/Cooling Days Drop-Down**: HDD **Notional Text Box:** 20. **Temperature Text Box:** 65 **Temperature Degree Drop-Down**: F **Currency**: The user's preferred currency *Hint for the Currency selection*: Obtain the **UserDefaults** from the Data Server connection, get the preferred currency, and use **AppUtil.*showFavoriteCcy*()** to set the currency choice.

Table 10-10: Task 5: showTrade() and buildTrade()

Step	Description
11	**showTrade(Trade)** and **buildTrade(Trade)** perform opposite actions. The **showTrade**() method synchronizes the **Trade** object with what is displayed in the window and the **buildTrade**() method synchronizes the display in the trade window with the **Trade** object.
12	**showTrade(trade)** method: from the passed trade object we set the necessary components. Using the variables of the trade and product objects, set all of the combo boxes and text fields in the window. Hint: Buy/Sell is a trade detail and the rest are product details. Also, you can you the utility methods on the **Util** object to convert a Number or Date to a String. For example, **Util.*numberToString*().**
13	**buildTrade(trade)** method: from the selected items in the trade window, set the relevant trade and product details on the passed trade object. On the **trade** object: • Set the Settlement Date to the Start Date. • Set the Settlement Currency to the selected currency. • Set the Trade Currency to the selected currency. • Set the Quantity to be 1 if this derivative is purchased and -1 if it is sold. • On the **product** object (HDDCDD): • Set all the variables on the HDDCDD object. Hint: Again, you may make use of utility methods in the **Util** class to convert Strings to Numbers and JDates.

10.6.5 Exercise: Add the Trade Window to Main Entry

Goal — Write the components in the product container GUI of the Trade Window.

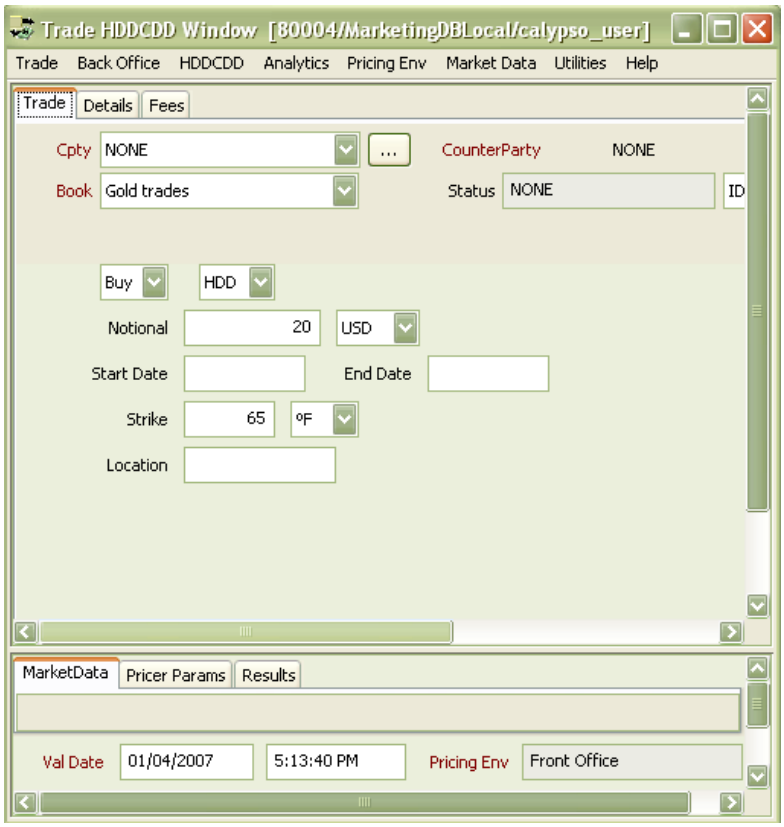
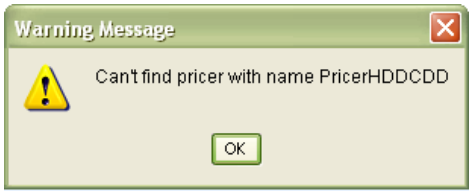
Prerequisite: Completion of the HDDCDDTradeProductPanel code.

Refer to the sample code
`api.examples.apps.product.TradeHDDCDDWindow`.

Table 10-11: Add the new Trade screen to Main Entry

Step	Description
1	Make sure that all the code HDDCDD.java, HDDCDDSQL.java, HDDCDDTradeProductPanel.java, Trade-HDDCDDWindow.java have all been compiled into the correct directory and the product table has been added to the database.
2	From Main Entry, choose **Utilities > MainEntry Configurator** to bring up the customizer window. Under the Trade menu create a new menu called Weather Derivatives. Then under the sub-menu Weather Derivatives create a new item called HDD / CDD with the action **trading.TradeHDDCDDWindow**. This will launch the Heating Degree Days/Cooling Degree Days screen. 

Table 10-11: Add the new Trade screen to Main Entry

Step	Description
3	Click Save, then re-start MainEntry. Select the new item from the menu and you should see something similar to the following. 
3	You should see a warning message similar to the one below as the window tries to find a default pricer for this product of named PricerHDDCDD and can't find one. This warning will go away once a pricer is assigned. 

10.7 Validating Security Codes for Custom Products

Call *Product.checkSecConstraints()* on the server side or *RemoteProduct.checkSecConstraints()* on the client side, to validate the security codes prior to saving. This will return a vector of error messages if any.

10.8 Customizing Structured Products

10.8.1 Creating a Custom Structured Product

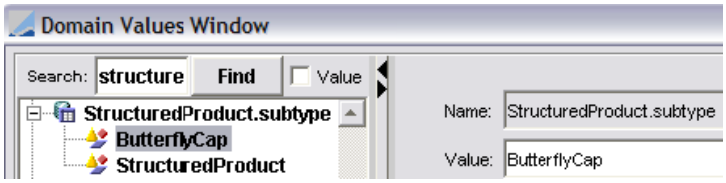
Structured Product allows new product types that are composed of individual products to be added to the system in a timely manner. A Structured Product is treated as a single trade by the system. For instance, a risk report will show a structured product trade as a single trade and the back office will produce a single confirmation.

In this sample, we will add a butterfly cap, which is composed of one long cap at strike x, two short caps at strike y and one long cap at strike z with x < y < z. The notional of all the caps should be of the same amount. The trade entry validation sample code will check that the notional amount in all the component caps are equal and will issue a warning if the validation fails. The entry validation is performed before a structured product trade is saved.

Do the following for creating a custom structured product:

1. Register the new Structured Product type in the **StructuredProduct.subtype** domain.

Figure 10-7: Domain Values Window — Registering a New Structured Product Type



2. Structure the product using **Main Entry -> Trade -> Structured Product**. Select **ButterflyCap** from the **Type** field. Create a butterfly cap structure by adding two long caps and two short caps.

10.8.2 Customizing Validation by Product Subtype

You can customize the structured product validation by product subtype. Create a class named **`tk.product.StructuredProduct<product_subtype>Constraint`** that implements **`com.calypso.tk.product.StructuredProductConstraint`**.

Sample Code in `calypsox/tk/product/`

`StructuredProductCorridorCapConstraint.java`

10.8.3 Customizing Report Style by Product Subtype

In the reports, you can customize the display by product subtype. The system will first look if there is a customization by product subtype, for example StructuredProductCorridorCapReportStyle, then by product type StrcturedProductReportStyle.

Sample CodeRegister the new Pricer

calypsox/tk/report/StructuredProductCorridorCapReportStyle.java

10.9 Adding Custom Exotic Functions to the Formula Editor

To create a custom exotic function fro use in the Formula Editor, create a class named **tk.product.eXSP.functionI.XFX<*function_name*>** (where <*function_name*> is the name of your choice) that implements **com.calypso.tk.product.eXSP.function.ExoticFunction** and extends **com.calypso.tk.product.eXSP.functionI.ExoticFunctionI**.

At a minimum, your class should implement the following methods:

```
performCompute(eXSPComputeEnvironment, Vector, CashFlowPeriod, CashFlowPeriod)
parse(eXSPComputeEnvironment, String)
```

Sample Code in calypsox/tk/product/eXSP/

functionI/XFXtest.java.

The name to be registered in the domain ExoticFunction is the name defined for the variable XFUNC_NAME. In the sample, it is "xtest".

```
private static String XFUNC_NAME = "xtest";
```

Refer to the javadoc for the ExoticFunction interface for further information.

10.10 Customizing Existing Products

10.10.1 Creating Custom Attributes

Do the following for creating custom attributes for a product:

1. Create a class named **tk.product.<*custom_data_class_name*>** which contains all of your additional attributes and which implements the interface **com.calypso.tk.core.ProductCustomData**.
2. To make the custom attributes persistent, create a class named **tk.product.sql.<*custom_data_class_name*>SQL** which extends the abstract base class **com.calypso.tk.core.sql.ProductCustomDataSQL**.

<*custom_data_class_name*>SQL is invoked from **com.calypso.tk.core.sql.ProductSQL**.

Sample Code in calypsox/tk/product/

ProductCustomData sample:

RepoProductExtension.java

ProductCusomtDataSQL sample:

sql/RepoProductExtensionSQL.java

10.10.2 Using Product-Related Interfaces

SpecificResetBased

To handle specific resets, a product implements the **`SpecificResetBased`** interface. This interface requires the implementation of the method *getSpecificResets()*, which returns a vector of ProductReset.

The pricers use the vector of ProductReset to calculate cashflows known interest amounts.

10.10.3 Creating a Custom Trade Decomposition Routine

For example, you want to decompose a complex trade into more than one basic trade so that risk, positions, etc can be computed on the basic trades.

Create a class named **`tk.mo.<product_type>Explode`** which implements the interface **`com.calypso.tk.mo.TradeExplode`**. Implement the *explode()* method.

<*product_type*>Explode is invoked from **`com.calypso.tk.mo.TradeExplode`**.

Sample Code in `calypsox/tk/mo/`

- `FXForwardTakeUpExplode.java`
- `IROptionExplode.java`

10.10.4 Creating a Custom Retrieval Routine for a Product

For example, you may wish to import a bond from another system when a user enters its CUSIP in a trade.

Create a class named **`tk.product.sql.CustomProductFinder`** which extends **`com.calypso.tk.core.sql.ProductFinder`**.

This class will be invoked from **`com.calypso.tk.core.sql.ProductSQL`** when retrieving a product.

Sample Code in `calypsox/tk/product/sql/`

- `CustomProductFinder.java`

10.10.5 Applying Custom Validation to a Product

Create a class using one of the following names, **`tk.product.<product_type><product_subtype>ProductValidator`**, **`tk.product.<product_type>ProductValidator`**, **`tk.product.<product_family>ProductValidator`**, or **`tk.product.DefaultProductValidator`**, which extends the class **`com.calypso.tk.product.ProductValidator`**.

The following methods should be implemented:

- *isValidInput()* — Returns true or false depending upon whether validation succeeds or fails.

- *applyDefaults()* — Sets default values for the product if not already set, for instance holidays. Method called before a Trade is saved if validation succeeds.

product*.ProductValidator is invoked from **com.calypso.tk.product.ProductValidatorUtil** to validate product information and to apply default values as applicable.

Sample Code in calypsox/tk/product/

- BondAssetBackedProductValidator.java
- OTCEquityOptionVanillaProductValidator.java

10.10.6 Creating a Custom Product Description

Typically used in reports, the task station and blotters for product sub-type(s) and quote names.

Create a class named **tk.product.<*product_type*>ProductDescriptionGenerator** or **tk.product.<*product_family*>ProductDescriptionGenerator** which implements the interface **com.calypso.tk.core.ProductDescriptionGenerator**.

product*ProductDescriptionGenerator is invoked from **com.calypso.tk.core.ProductDescriptionGeneratorUtil**.

Sample Code in calypsox/tk/product/

- BondProductDescriptionGenerator.java

10.10.7 Creating a Custom Spot Date Calculation

A custom spot date calculation is called from Trade windows by double-clicking the Settle Date and from pricers where the computation of a spot date is required.

Create a class named **tk.product.<*product_type*>SpotDateCalculator** that extends the class **com.calypso.tk.product.SpotDateCalculator**.

<*product_type*>SpotDateCalculator is invoked from **com.calypso.tk.product.SpotDateCalculatorUtil**.

Sample Code in calypsox/tk/product/

- SwapSpotDateCalculator.java

10.10.8 Creating a Custom Basket Calculation

Create a class named **tk.product.<*basket_function_name*>** that implements the interface **com.calypso.tk.product.BasketFunction**.

The <*basket_function_name*> class is invoked from **com.calypso.tk.product.sql.SecurityBasketSQL**.

10.10.9 Creating a Custom ObservedData

Do the following for creating a custom ObservedData:

1. Create a class named **tk.product.<*custom_data_class_name*>** that implements the interface **com.calypso.tk.product.CustomObservedData**.
2. To make the ObservedData persistent, create a class named **tk.product.sql.<*custom_data_class_name*>SQL** that extends **com.calypso.tk.product.sql.CustomObservedDataSQL**.

   Invoke this class from: **com.calypso.tk.product.sql.ObservedDataSQL**.

Sample Code in `calypsox/tk/product/`

CustomObservedData sample:

`TestCustomObservedData.java`

CustomObservedDataSQL sample:

`sql/TestCustomObservedDataSQL.java`

10.10.10 Creating a Custom Payout Formula

Create a class named **tk.product.util.PayOutFormula<*name*>** that extends **com.calypso.tk.product.util.PayOutFormula**.

This class will be invoked from **com.calypso.tk.product.util.PayOutFormula**.

10.10.11 Customizing a Bond

Creating a Custom Bond

Create a class named **tk.product.<*bond_type*>** that extends **com.calypso.tk.product.Bond**.

Handling Bond Prices

Bond prices are handled in the BondPrice class (in the **com.calypso.tk.core** package). This class does for bond prices what the **tk.core.Amount** class does for currency amounts. That is, it allows the user to store complete information about the price, including the tick size. Such information could not be represented in a primitive data type like a double. To support the use of BondPrice, the **tk.core.Util** class now has two methods, *bondPrice2String()* and *string2BondPrice()*, to convert from and to a BondPrice object.

When creating a new BondPrice object, you must set its tick size. Use the new method *getQuoteBase()* in the Bond class to find out the tick size, and set the BondPrice's tick size accordingly. The tick size is the integer denominator in the fractional portion of the bond's price (for example, 32, 64, or 100).

Creating a Custom Dialog for the Bond Product Window

A Custom Data button is available on the Bond product window. It will invoke a class that implements CustomDataWindow.

Create a class named **apps.product.BondCustomDataWindow** that implements **com.calypso.apps.product.CustomDataWindow**.

Invoke this class from **com.calypso.apps.product.Bond**.

10.10.12 Customizing an ETOContract

Creating a Custom ETOContract

Create a class named **tk.product.ETO<*ETO_underlying_type*>** that extends **com.calypso.tk.product.ETO**.

10.10.13 Customizing a FutureContract

Creating a Custom FutureContract

Create a class named **tk.product.Future<*future_type*>** that extends **com.calypso.tk.product.Future**.

Creating a Custom DateGenerator for a FutureContract

Create a class named **tk.product.ContractDateGenerator<*custom_name*>** that implements the interface **com.calypso.tk.product.ContractDateGenerator**.

This class will be invoked from **com.calypso.tk.product.FutureContract**.

Sample Code in calypsox/tk/product/

ContractDateGeneratorTest.java

10.10.14 Customizing a FutureOptionContract

Creating a Custom FutureOptionContract

Create a class named **tk.product.FutureOption<*future_option_type*>** that extends **com.calypso.tk.product.FutureOption**.

Create a Custom DateGenerator for a FutureOptionContract

Create a class named **tk.product.OptionContractDateGenerator<custom_name>** that implements the interface **com.calypso.tk.product.OptionContractDateGenerator**.

This class will be invoked from **com.calypso.tk.product.FutureOptionContract**.

Sample Code in calypsox/tk/product/

OptionContractDateGeneratorTest.java

10.10.15 Credit Derivatives

Some useful interfaces.

- **com.calypso.tk.pricer.PricerReferenceEntity** — Pricers which can calculate pricer measures per reference entities (issuer id and seniority). Used by credit derivatives reports.
- **com.calypso.tk.product.CreditRisky** — Any product that may have credit risk associated with it (such as CreditDefaultSwap, TotalReturnSwap, AssetSwap, Bond, etc.). Used by credit derivatives reports.
- **com.calypso.tk.product.CreditEventBased** — Any product that can be affected by credit events (such as CreditDefaultSwap). Used by the credit event application.

10.11 ProductChooser Window Customization

10.11.1 Creating a Custom Panel in the ProductChooser Window

Create a class named **tk.product.<*product_type*>ProductChooserHandler** which extends **com.calypso.tk.product.ProductChooserHandler**.

This class will be invoked from **com.calypso.tk.product.ProductChooserHandler**.

Sample Code in calypsox/tk/product/

BondProductChooserHandler.java

10.11.2 Creating a Custom ProductChooser

Create a class named **apps.product.<*product_family*>ProductChooser** which implements **com.calypso.apps.product.ProductChooser**.

This class will be invoked from **com.calypso.apps.product.ProductUtil** to load a list of products for display in the ProductChooser window.

Note: The Product Specific panel in the Pricer Config call *ProductUtil.getChooser()*, so if a custom ProductChooser class exists, it will be invoked from the Pricer Config.

10.12 Printing a Product

Create a class named **tk.product.<*product_type*>ProductPrint** that implements **tk.product.ProductPrint**.

<*product_type*>ProductPrint is invoked from **com.calypso.tk.core.ProductPrintUtil**.

Sample Code in calypsox/tk/product/

FXProductPrint.java

10.13 How to Use Cashflows

When using out-of-the-box products, cashflows will be automatically generated (provided they implement the CashFlowGeneratorBased interface).

10.13.1 Cashflows Generation

The Calypso system assumes that cashflows and all of their "known" attributes are generated from contractual data, i.e. Trade and Product classes. In other words, attributes such as flow dates and known flow amounts (for example a fixed flow, or a floating flow where the reset date is in the past) should not depend on the pricing algorithm being used. Hence, the cashflows are generated and the known amounts are calculated by the Product class (*generateFlows()* and *calculate()* methods). These flows are used by the entire Calypso system including back office components, for example to generate payments – again the assumption here being that the known payments should only depend on contractual data and not the pricing algorithm being used. If a flow attribute which is assumed to be part of the contractual data (such as the payment date of a swap flow) must be customized, this again is done and saved as part of trade/product using the customized checkbox on the cashflow tab.

The Pricer's responsibility is to calculate the values of the requested pricer measures. It can also project valuation attributes on the cashflows, such as the projected amount of an unknown flow (for example a floating flow where the reset date is in the future) or the survival probability used for a credit default swap flow. In addition to all of the existing attributes on the cashflows which can be set by the pricer (such as the projected amount, discount factor, etc.), the pricers can also add their own attributes by implementing the *getCashFlowColumnNames()* and *getCashFlowColumn()* methods. All of these attributes are displayed on the cashflows tab of the trade windows.

In short, the Calypso system separates the cashflow generation into two parts:

- The first part generates and sets the contractually defined attributes of the cashflows, and
- The second part sets the attributes necessary for valuation.

Creating a Custom Cashflow Calculator for a Reference Index

Create a class named **`tk.product.flow.<currency><index_name>Calculator`** or **`tk.product.flow.<index_name>Calculator`** which implements the interface **`com.calypso.tk.product.flow.IndexCalculator`**.

Note that <*index_name*> can be the value of the rate index attribute IndexCalculator.

The <*index_name*>.Calculator class is invoked fromThe **`com.calypso.tk.product.flow.IndexCalculatorUtil`**.

Creating a Custom Cashflow Generator for a Product

Create a class named **tk.product.flow.CashFlow<*name*>** that implements **com.calypso.tk.product.flow.CashFlowSimple**.

CashFlow<*name*> is invoked from **com.calypso.tk.product.sql.CashFlowSQL**.

Creating a Custom Coupon Period

Create a class named **tk.product.util.<*product_type*>PeriodGenerator**, **tk.product.util.<*product_family*>PeriodGenerator**, or **tk.product.util.CustomDefaultPeriodGenerator** that implements **com.calypso.tk.product.util.CustomPeriodGenerator**.

The *PeriodGenerator class is invoked from **com.calypso.tk.product.util.PeriodGenerator** to calculate coupon period dates when generating the cashflows.

Creating a Compound Period

Create a class named **tk.product.flow.CashFlowCompound<name>** that implements **com.calypso.tk.product.flow.CashFlowCompound**.

This class will be invoked from **com.calypso.tk.product.sql.CashFlowCompoundSQL**.

10.13.2 Cashflows Display

The functionality involving columns (accessible through the GUI) is distinct from the actual generation of cashflows by products. It is made possible via the CashFlowLayout class or possibly one of its product-specific subclasses, for example SwapCashFlowLayout. Such a class converts a CashFlowSet returned by the product's *getFlows()* method to a table for GUI display, and allows users to edit cashflows and lock cashflows to prevent changes.

All flows will initially be generated and calculated independent of the column configuration in CashFlowLayout or its subclasses. However, if the values in that column have subsequently been edited and locked, the product should contain a method that uses the appropriate CashFlowLayout class using the static method *CashFlowLayout.createCashFlowLayout(Product)* to take into account the locked flows. For example, see *SwapLeg.generateAndKeepLocksFlows()* below.

```
public CashFlowSet generateAndKeepLocksFlows(boolean paySideB, JDate asOfDate)
  throws FlowGenerationException
  {
    CashFlowSet flows = getFlows(); //just to make sure it is uncompressed
    if (flows == null || flows.size() == 0) {
      generateFlows(paySideB);
      return getFlows();
  }

  CashFlowLayout cfParser = CashFlowLayout.createCashFlowLayout("Swap");
```

```
  cfParser.processCashFlows(flows,
              getCouponPaymentAtEndB(),
              getPrincipalActualB(),
              asOfDate,
              this);
  long idLock = getCfGenerationLocks();
  Vector colLocks = cfParser.ids2VectorNames(idLock);
  cfParser.setColumnLocks(colLocks);
  generateFlows(paySideB);
  CashFlowSet newFlows = getFlows();
  cfParser.checkBeforeApplyingLocks(newFlows);
  cfParser.applyLockedValuesToCashFlows(newFlows,
              getCouponPaymentAtEndB(),
              getPrincipalActualB(),
              this);
  setFlows(newFlows);
  return newFlows;
}
```

On the other hand, if only the cashflow generation is different for a product, but not the actual display of the cashflows, it may be better to subclass an existing CashFlowGeneratorBase implementation, and override the flow generation methods.

Creating a Custom Cashflow Panel

Create a class named **tk.product.util.<product_type>CashFlowLayout**, **tk.product.util.<product_family>CashFlowLayout**, or **tk.product.util.CustomCashFlowLayout** which extends the class **com.calypso.tk.product.util.CashFlowLayout**.

This class will be invoked from **com.calypso.tk.product.util.CashFlowLayout**.

You can add custom columns to a custom CashFlowLayout in the following manner:

```
final static public int XYZ = 999;        // column id
final static public String S_XYZ = 'xyz'; // column name
```

For editable columns you can use the following ids: 51, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, and 64. Editable columns ids are stored with a product to indicate which columns are locked and modified.

Thus, you have the opportunity to edit the values in the columns, provided they are editable. The custom CashFlowLayout should implement *isColumnEditable(String colName)* for editable columns.

For non editable columns it is recommended to use ids 500 and above to avoid any conflict with the base class.

Note that while a subclass and its parent cannot have overlapping column IDs, overlapping ids are allowed across subclasses.

The *parseCustomFlow()* method in a custom CashFlowLayout should only parse those custom flow fields that are not known by standard cash-

flow types, since the *parseFlow()* method in CashFlowLayout handles parsing of the standard cashflow types.

10.13.3 Principal Schedule

Creating a Custom Principal Schedule Generator

This could be used in conjunction with ProductCustomData to store the attributes for the generator.

Create a class named **tk.product.util.PrincipalGenerator<structure_name>** which implements the interface **com.calypso.tk.product.util.CustomPrincipalGenerator**.

This class will be invoked from **com.calypso.tk.product.util.PrincipalScheduleGenerator**.

Sample Code in calypsox/tk/product/util/

PrincipalGeneratorSample.java

Creating a Custom Principal Schedule Window

Currently available in the swap, cap/floor, swaption and cash trade windows.

Create a class named **apps.product.<structure_name>PrincipalStructureDialog** which implements the interface **com.calypso.apps.product.PrincipalStructureDialog**.

Creating Set and Get Amortization Parameters

To set and get amortization parameters, the following must be done.

1. Get the parameters Hashtable from the underlying product (we currently support SwapLeg and CapFloor): cast the product appropriately and call *getParams()*.

   The returned Hashtable will be used to store and retrieve the amortization parameters.

2. To set the Start Date for example, do the following:

```
PrincipalScheduleGenerator.setStartDate(params, date);
```

   where *params* is the previously retrieved Hashtable, and *date* is a JDate object representing the start date to be stored.

3. To get the Start Date for example, do the following:

```
JDate date = PrincipalScheduleGenerator.getStartDate(params);
```

   where *params* is the previously retrieved Hashtable.

   The amortization schedule, amount, rate, frequency and daycount can be obtained from the SwapLeg or CapFloor directly by calling get and set methods like getAmortAmount()/setAmortAmount(amount).

Refer to the Javadoc for PrincipalScheduleGenerator, SwapLeg, and CapFloor for details.

11 Pricing

11.1 Pricing Environment

11.1.1 Using a Pricing Environment

A Pricing Environment tells the system what pricers (pricing models) and market data (interest rate curves, volatility surfaces, quotes etc.) to use to value each product. A Pricing Environment contains a PricerConfig and a QuoteSet object. The PricerConfig specifies what Pricer to use for a product and which market data items to use with the Pricer. The QuoteSet is a repository of quote values that are needed for valuation and market data generation.

Sample Code in `samples/`

`PricingEnvSample.java`

Demonstrates how to price a trade given a PricingEnv.

11.1.2 Creating a Custom Panel for the PricerConfig Window

Create a class named **`apps.marketdata.PCProductSpecificMDPanel`** that implements the interface **`com.calypso.apps.marketdata.PCProductSpecificMDPanel`**.

This class will be invoked from **`com.calypso.apps.marketdata.PricerConfigWindow`**.

Sample Code in `calypsox/apps/marketdata/`

`PCProductSpecificMDPanel.java`

11.2 Pricer

11.2.1 Creating a Custom Pricer

Overview of Steps

- Step 1 — Create a Pricer
- Step 2 — Register the new Pricer

Step 1 — Create a Pricer

Create a class named **`tk.pricer.<pricer_name>`** which extends the abstract base class **`com.calypso.tk.core.Pricer`**.

The Pricer calculates a number of pricer measures: NPV, DELTA, etc. Note that the Calypso framework does not limit the type or number of pricer measures that a pricing model can support.

Note: If the custom Pricer uses market data specified in the Product Specific, Custom or Credit panels of the Pricer Configuration, it must be Lazy Refresh compatible (see below for details).

This class will be invoked from **`com.calypso.tk.core.Pricer`**.

Sample Code in `calypsox/tk/pricer/`

- `PricerCapFlrDEMO.java`
- `PricerDEMO_P2.java`

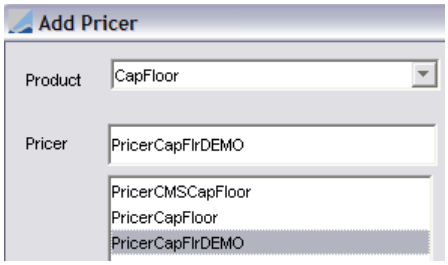
PricerCapFlrDEMO is a pricing model for Cap/Floor. The pricing model supports the calculation of the following pricer measures:

- NPV returns (zero rate * volatility) at the maturity date of the cap
- DELTA, GAMMA, THETA, VEGA, and CASH return 1, 2, 3, 4, and, 0 respectively

Step 2 — Register the new Pricer

Register the new pricer using **Main Entry -> Configuration -> System -> Add Pricer**.

Figure 11-1: Add Pricer Window



11.2.2 Making a Pricer Lazy Refresh Compatible

Lazy Refresh is a mode on the Pricer Configuration which improves the system performance by not loading the market data items, only their ids. The actual market data items are loaded when requested by an application, such as a Pricer. This is achieved by giving the list of MarketDataItem ids which are not yet loaded to the Pricer Configuration, and refreshing the Pricer Configuration.

The Lazy Refresh mode only applies to market data specified in the Product Specific, Custom and Credit panels of the Pricer Configuration. If a custom Pricer uses market data from these panels it MUST be made Lazy Refresh compatible, otherwise the market data will not be loaded in Lazy Refresh mode.

To make a Pricer compatible with the Lazy Refresh mode, you must override the following methods:

- getMarketDataItemIds(Trade trade, PricingEnv env, JDate valDate, Hashtable itemIds)

  In the itemIds Hashtable provide the list of lazy refresh enabled MarketDataItem ids required by the Pricer that are not yet loaded as: Key=Integer (MarketDataItem id), Value=Integer (MarketDataItem id).
- getMarketDataItemsWithoutRefresh(Trade trade, PricingEnv env, JDate valDate, Hashtable items)

  In the items Hashtable provide the list of MarketDataItems that are already loaded from the Pricer Configuration and can be used without refreshing the Pricer Configuration as: Key=MarketDataItem, Value=MarketDataItem.

Sample Code for Retrieving a Probability Curve

Probability curves are defined in the Credit panel of the Pricer Configuration.

```
public void getMarketDataItemIds(Trade trade, PricingEnv env,
   JDate valDate, Hashtable itemIds) {
      CreditDefaultSwap cds = (CreditDefaultSwap)trade.getProduct();
      ReferenceEntitySingle refEntity =
         (ReferenceEntitySingle)cds.getReferenceEntity();
      if (refEntity == null) return;
      PricerConfig config = env.getPricerConfig();
      Integer probCurveId = getCreditMarketDataItemId(config, cds, refEntity,
         PricerConfig.PROBABILITY);

      if (probCurveId != null) {
         itemIds.put(probCurveId,probCurveId);
      }
}

private Integer getCreditMarketDataItemId(PricerConfig config, CreditDefaultSwap cds,
   ReferenceEntitySingle refEntity, String usage) {
      Integer itemId = null;
      int tickerId = refEntity.getTickerId();
      if (tickerId > 0) {
         itemId = config.getCreditMarketDataItemId(usage, tickerId);
      }
      if (itemId != null) return itemId;
      int issuerId = refEntity.getLegalEntityId();
      String seniority = refEntity.getSeniority();
      itemId = config.getCreditMarketDataItemId(usage, cds.getCurrency(), issuerId,
         seniority, cds.getRestructuringType());
        return itemId;
}
```

Sample Code for Retrieving a Dividend Curve

Dividend curves are defined in the Product Specific panel of the Pricer Configuration.

```
public void getMarketDataItemIds(Trade trade, PricingEnv env,
   JDate valDate, Hashtable itemIds) {
      Product equityOption = trade.getProduct();
      Product underlying = getOptionUnderlying(equityOption);
      PricerConfig config = env.getPricerConfig();
      Integer divId =getDividendCurveId(equityOption, underlying, config);
      if (divId != null) {
         itemIds.put(divId,divId);
      }
}

protected Integer getDividendCurveId(Product option, Product underlying,
   PricerConfig config) {
      if (underlying == null) return null;
      return (Integer) config.getProductSpecificMDId(PricerConfig.DIVIDEND,
         underlying);
}
```

Note: In Lazy Refresh mode, *getProductSpecificMD()* **will not return** the MarketDataItem, you must use *getProductSpecificMDID()* instead, as shown above.

Custom Programs

If you have a custom program that must retrieve market data in Lazy Refresh mode, you must refresh the pricer configuration to load the MarketDataItems for the MarketDataItem ids.

```
mdataItem = config.getProductSpecificMD(usage, pe);
if (mdataItem == null) {
   Integer mdataItemId = config.getProductSpecificMDID(usage, pe);
   if (mdataItemId == null) {
      // the market data is really not there
      // throw an exception that market data is missing
   }
   else {
      Vector ids = new Vector();
      ids.addElement(mdataItemId);
      // refresh the pricer configuration
      config.refresh(ids, env);
      mdataItem = config.getMarketDataItem(mdataItemId);
   }
}
```

If you want to load the full pricing environment, use *PricerConfig.getAllMarketDataItems()*. It will load all MarketDataItems, including market data specified in the Product Specific, Custom and Credit panels of the Pricer Configuration.

11.2.3 Creating a Custom Pricing Parameter Entry Panel

Create a class which implements the interface **com.calypso.apps.trading.PricerInputViewer**. The name of the class should be returned by your pricer's *getInputViewerClassName()* method. By default **apps.trading.PricingJPanel** will be used.

This class will be invoked from **com.calypso.apps.trading.TradeViewerJFrame**.

11.2.4 Performing a Custom Action after Pricing

For example, you want to change the color or font of your pricing results after pricing.

Create a class named **apps.trading.PricerOutputViewer<pricer_name>** which implements the interface **com.calypso.apps.trading.PricerOutputViewer**.

This class will be invoked from **com.calypso.apps.trading.TradeWindow**.

11.2.5 Creating a Custom Solver

Create a class named **tk.pricer.<product_family>SolveFor** or **tk.pricer.<product_type>SolveFor** that implements **com.calypso.tk.pricer.SolveFor**.

This class will be invoked from **com.calypso.tk.pricer.SolveForUtil**.

11.2.6 Creating a Custom Inflation Forecasting Method

Create a class named **tk.pricer.<currency><index_name>InflationCalculator** or **tk.pricer.<index_name>InflationCalculator** that implements **com.calypso.tk.pricer.InflationCalculator**.

It will be invoked from **com.calypso.tk.pricer.InflationUtil**.

11.3 Pricer Measure

11.3.1 Creating a Custom Pricer Measure

A new pricer can calculate pricer measures that are not supported by the PricerMeasure class. In this case you must create a new PricerMeasure class.

Overview of Steps

- Step 1 — Create a PricerMeasure
- Step 2 — Register the new PricerMeasure types

Step 1 — Create a Pricer Measure

Create a class named **<pricer_measure_class_name>** which extends the class **com.calypso.tk.core.PricerMeasure**. It is recommended to place the PricerMeasure class in the **tk.pricer** package but it is not mandatory.

Overwrite the following methods:

- getName()
- toString()
- toInt()
- getDisplayClass()
- isAdditive()
- *calculate()* — This method should be implemented if a given pricer measure type is applicable for all pricers across all product types. All existing pricers will invoke this method when a new pricer measure type is encountered.
- isImplementedByPricer()

```
public boolean isImplementedByPricer(Pricer pricer) {
   return true;
}
```

This class will be invoked from **com.calypso.tk.util.PricerMeasureUtility**.

Specify a name and an id for each PricerMeasure type as shown below.

```
final static public int ZV_SPREAD=138;
final static public int ZV_YIELD=139;
final static public String S_ZV_SPREAD="ZV_SPREAD";
final static public String S_ZV_YIELD="ZV_YIELD";
```

Tips

- Use an id that starts with 1000 or higher to avoid any conflict with Calypso pricer measure types.
- Create a single PricerMeasure class to hold all pricer measure types that are required by your pricing models in order to centralize the information.

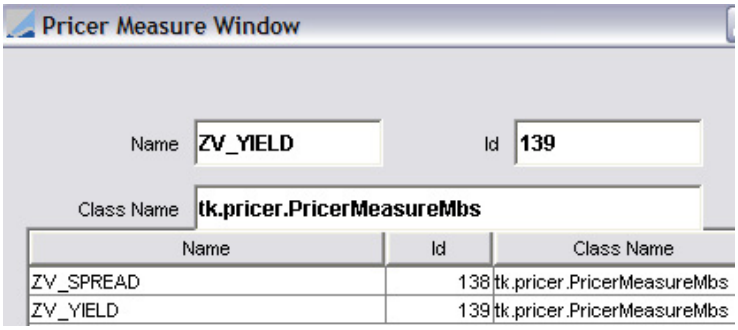
Sample Code in calypsox/tk/pricer/

- PricerMeasureTst.java
- PricerMeasureMbs.java

Step 2 — Register New Pricer Measure Types

Register the PricerMeasure types using **Main Entry > Configuration > System > Add Pricer Measure**.

Figure 11-2: Pricer Measure Window — Regitering a New Pricer Measure Type



For each PricerMeasure type, enter the name, its associated id and the fully qualified class name of the PricerMeasure class.

11.3.2 Creating Client Data for a Pricer Measure

Create a class that implements the interface **`com.calypso.tk.core.PricerMeasureClientData`** to specify a custom viewer for example. Your pricer creates an object of this type, and attaches it to the PricerMeasure by calling *PricerMeasure.setClientData()*.

11.3.3 Creating a Custom Display for a Pricer Measure

Create a class named **`apps.trading.PricerMeasure<viewer_name>Viewer`** or **`apps.trading.PricerMeasure<pricer_measure_name>Viewer`** that extends Jdialog and implements **`com.calypso.apps.trading.PricerMeasureViewer`**. The viewer name can be set through client data.

This class will be invoked from **`com.calypso.apps.trading.PricerMeasureUtil`** and will display a popup window when you double-click on the PricerMeasure results. If client data has been set on this pricer measure, it will retrieve the viewer name from the client data, otherwise, it will invoke **`PricerMeasure<pricer_measure_name>Viewer`**.

Sample Code in `calypsox/apps/trading/`

`PricerMeasureNPVViewer.java`

12 Trade

12.1 Trade

12.1.1 Creating Custom Trade Attributes

Do the following to create custom Trade attributes:

1. Create a class named **tk.core.<custom_data_class_name>** that contains all of your additional attributes and which implements the interface **com.calypso.tk.core.TradeCustomData**.
2. To make the custom data persistent, create a class named **tk.core.sql.<custom_data_class_name>SQL** which extends the abstract base class **com.calypso.tk.core.sql.TradeCustomDataSQL**.

This class will be invoked from **com.calypso.tk.core.sql.TradeSQL**.

Sample Code in calypsox/tk/core/

TradeCustomData sample:

RepoTradeExtension.java

TradeCustomDataSQL sample:

sql/RepoTradeExtensionSQL.java

12.1.2 Applying Custom Validation to a Trade

Create a class named **apps.trading.<ProductTypeCode>TradeValidator** or **apps.trading.CustomTradeValidator** which implements the interface **com.calypso.apps.trading.TradeValidator** and extends **Frame** or **javax.swing.Jframe**.

The custom TradeValidator will apply to all product types for which a product-specific TradeValidator is not found. Define the following methods in your TradeValidator:

- *inputInfo()* — Displays a Dialog that lets the user input extra data.
- *isValidInput()* — Checks a trade to make sure it is ready to be saved.

This class will be invoked from **com.calypso.apps.trading.TradeViewerJFrame** to create a popup window for users to input extra data as applicable, and to validate the trade prior to saving.

Sample Code in calypsox/apps/trading/

FXTradeValidator.java

StraddleTradeValidator.java

StructuredProductButterflyCapTradeValidator.java

12.1.3 Creating a Custom Copy and Paste Function

You can create copy and paste functions between two trades.

Create a class **apps.trading.CopyTrade<from_product_type>2<to_product_type>** which implements the interface **com.calypso.apps.trading.CopyTrade**.

This class will be invoked from **com.calypso.apps.trading.TransferableTrade**.

Sample Code in calypsox/apps/trading/

CopyTradeSwap2FRA.java

12.1.4 Creating a Custom Save As New Function

Create a class named **apps.trading.CustomSaveAsNewTrade** which implements the interface **com.calypso.apps.trading.SaveAsNewTrade**.

This class will be invoked from **com.calypso.apps.trading.TradeUtil**.

Sample Code in calypsox/apps/trading/

CustomSaveAsNewTrade.java

12.1.5 Creating a Custom Keyword Validator

Create a class named **apps.trading.CustomKeywordValidator** which implements the interface **com.calypso.apps.trading.KeywordValidator**.

This class will be invoked from **com.calypso.apps.trading.TradeViewerJFrame**.

Sample Code in calypsox/apps/trading/

CustomKeywordValidator.java

12.1.6 Creating a Custom Mirror Trade

Create a class named **tk.product.<product_type>MirrorHandler** or **tk.product.DefaultMirrorHandler**, which implements the interface **com.calypso.tk.product.MirrorHandler**.

This class will be invoked from **com.calypso.tk.product.MirrorHandlerUtil**.

Sample Code in calypsox/tk/product/

SwapMirrorHandler.java

12.2 Trade Window

12.2.1 Creating a Custom Trade Window Title

Create a class named **apps.trading.<product_type>TradeWindowTitleGenerator** that implements **com.calypso.apps.trading.TradeWindowTitleGenerator**.

This class will be invoked from **com.calypso.apps.trading.TradeWindowTitleGeneratorUtil**.

12.2.2 Creating Custom Default Values

Create a class named **apps.trading.Trade<product_type>DefaultValues** that implements **com.calypso.apps.trading.TradeDefaultValues**.

This class is invoked from **com.calypso.apps.trading.TradeWindowBase**, and will override existing product default values with custom values. **TradeDefaultValues** is not available to trade windows based on TradeWindow.

Sample Code

```
package calypsox.apps.trading;

import com.calypso.tk.core.*;
import com.calypso.tk.product.*;
import com.calypso.apps.trading.*;

public class TradeCommoditySwapDefaultValues implements TradeDefaultValues {
    public void setDefaultValues(Trade trade,ShowTrade w){
        CommoditySwap swap = (CommoditySwap)trade.getProduct();
        swap.setStartDate(JDate.getNow().addTenor(new Tenor("1Y")));
    }
}
```

12.2.3 Adding a Custom Panel to a Trade Window

Create a class named **apps.trading.CustomTabTrade<product_type>Window** that implements **apps.trading.CustomTabTradeWindow**.

Note that _tradeDetailsPanel will call the methods *buildTrade()*, *newTrade()*, and *showTrade()* of the CustomTabTradeWindow instance.

This class will be invoked from **com.calypso.apps.trading.TradeWindow**.

Sample Code in calypsox/apps/trading/

- CustomTabTradeSwapWindow.java
- CustomTabTradeRepoWindow.java

12.2.4 Creating a Custom Trade Dialog

A custom trade dialog can be implemented to enter additional details for a trade. Those details can be saved as trade keywords, therefore simplifying the trade customization.

This custom trade dialog can currently only be implemented for the Bond Front trade window and Repo Front trade window. An Info button will appear in the Trade panel that will invoke the custom dialog.

Create a class name **apps.trading.<product_type>TradeCustomDetailsDialog** or **apps.trading.TradeCustomDetailsDialog** that implements **com.calypso.apps.trading.CustomDetailsDialog**.

Sample Code in calypsox/apps/trading/

- TradeCustomDetailsDialog.java

12.2.5 Creating a Custom Trade Display

Only those trade windows that inherit from the **TradeWindowBase** class can use the CustomViewTrade.

Create a class named **apps.trading.CustomViewTrade** which implements the interface **com.calypso.apps.trading.ViewTrade**. The **CustomViewTrade** *show()* method is called from the *customView()* method of TradeWindow.java. The *customView()* method is only called from the *showTrade()* method of the **TradeWindowBase** class.

This class is invoked only by the *showTrade()* method of the **TradeWindowBase** class. Refer to the sample code.

Sample Code in calypsox/apps/trading/

- CustomViewTrade.java

  Red-flags a trade if it has the keyword CustomViewTrade.

12.2.6 Adding a Custom Menu Item to a Trade Window

For a specific pricer or a specific product.

Create a class named **apps.trading.CustomTradeMenu<pricer_name>** or **apps.trading.CustomTradeMenuProduct<product_type>** which implements the interface **com.calypso.apps.trading.CustomTradeMenu**.

This class will be invoked from **com.calypso.apps.trading.TradeWindow**.

Sample Code in calypsox/apps/trading/

- CustomTradeMenuProductCancellableSwap.java

12.2.7 Adding Custom Callbacks to a Trade Window

You can add callbacks before and/or after a trade is saved, removed, created, or priced, and before the trade window is closed.

Create a class named **apps.trading.CustomTradeWindowListener** which implements the interface **com.calypso.apps.trading.TradeWindowListener**.

This class will be invoked from **com.calypso.apps.trading.TradeWindow**.

Sample Code in calypsox/apps/trading/

CustomTradeWindowListener.java

12.2.8 Creating a Custom Warning Window

Create a class named **apps.trading.CustomTradeUpdate** that implements the interface **com.calypso.apps.trading.TradeUpdate**.

This class will be invoked from **com.calypso.apps.trading.TradeWindow** when a trade is modified.

12.2.9 Applying Custom Validation to a Trade Template

Create a class named **apps.trading.CustomTradeTemplateChecker** that implements the interface **com.calypso.apps.trading.TradeTemplateChecker**.

This class will be invoked from **com.calypso.apps.trading.TradeWindow**.

Sample Code in calypsox/apps/trading/

CustomTradeTemplateChecker.java

12.2.10 Creating a Custom FundingTradeHandler for AssetSwap

Create a class named **apps.util.CustomFundingTradeHandler** which implements the interface **com.calypso.apps.util.FundingTradeHandler**.

This class will be invoked from **com.calypso.apps.trading.AssetSwapTradeProductPanel**.

Sample Code in calypsox/apps/util/

CustomFundingTradeHandler.java

12.2.11 Creating a Custom CFD Execution Portfolio

Create a class named **apps.trading.CustomCFDExecutionPortfolio** that implements **com.calypso.apps.trading.CFDExecutionPortfolio**.

This class will be invoked from **com.calypso.apps.trading.CFDTerminationWindow**.

Sample Code in calypsox/apps/trading/

SampleCustomCFDExecutionPortfolio.java

12.2.12 Creating a Custom ETO Contract Selector Window

Create a class named **apps.trading.CustomContractSelector** that implements **com.calypso.apps.trading.ContractSelectorInterface**.

This class will be invoked from **com.calypso.apps.trading.ContractSelectorWindow**.

Sample Code in calypsox/apps/trading/

CustomContractSelector.java

12.3 Applying Custom Validation to CashSettleEntryWindow

Create a class named **apps.trading.CustomCashSettleEntryValidator** that implements the interface **com.calypso.apps.trading.CashSettleEntryValidator**.

This class will be invoked from **com.calypso.apps.trading.CashSettleEntryWindow**.

Sample Code in calypsox/apps/trading/

CustomCashSettleEntryValidator.java

12.4 Applying Custom Validation to a Bundle

Create a class named **apps.trading.CustomBundleValidator** which implements the interface **com.calypso.apps.trading.CustomBundleValidator**.

This class will be invoked from **com.calypso.apps.trading.TradeBundleWindow**.

12.5 Creating a Custom Blotter Trade Selector

Create a class named **apps.trading.CustomBlotterTradeSelector** which implements the interface **com.calypso.apps.trading.BlotterTradeSelector**.

This class will be invoked from **com.calypso.apps.trading.TradeBlotterPanel** to extend the list of tags available in the blotter under the "Add Trades..." button for loading a single trade. Currently, it contains Trade Id, Internal Ref, External Ref.

Sample Code in calypsox/apps/trading/

CustomBlotterTradeSelector.java

Illustrates adding a "Counterparty".

12.6 Adding Custom Menu Items to the Trade Blotter

You can customize the popup menu that appears when you right-click a trade.

Create a class named **apps.trading.CustomBlotterMenu** that implements **com.calypso.apps.trading.BlotterMenu**.

This class will be invoked from **com.calypso.apps.trading.TradeBlotterPanel**.

Sample Code in calypsox/apps/trading/

CustomBlotterMenu.java

12.7 Applying Custom Validation to a ManualLiquidation

Create a class named **apps.trading.CustomManualLiquidationValidator** that implements the interface **com.calypso.apps.trading.CustomManualLiquidationValidator**.

This class will be invoked from **com.calypso.apps.trading.ManualLiquidationJDialog**.

Sample Code in calypsox/apps/trading/

CustomManualLiquidationValidator.java

This sample also illustrates how to add access permissions on the manual liquidation process in addition to the existing business rules.

12.8 Creating a Custom Reference Entity Selection Window

Calypso's selection dialog lets users specify reference entities based on existing issuers in the system, for credit derivatives.

Create a class named **apps.trading.CustomRefEntityChooser** which extends the class **com.calypso.apps.trading.RefEntityChooserInterface**.

12.9 Creating a Custom BO Trade Display

Create a class named **apps.trading.CustomBOTradeDisplay** which implements the interface **com.calypso.apps.trading.BOTradeDisplay**.

This class will be invoked from **com.calypso.apps.trading.BOTradeFrame**.

Tip

- To override the PO SDI selection whenever the counterparty SDI is manually selected, implement the methods *getReceiverSDISelection()* and *getPayerSDISelection()*.
- You can also use the BOTradeDisplay interface to override the assignment of netting methods for standards SDIs as well as manual SDIs by implementing the *getNettingType()* method.

Note: These customizations will also appear in the Netting Manager, Assign, and Split windows of the Task Station.

- The method *modifyTransferRule()* allows modifying the behavior of a transfer rule when it is manually modified.

Sample Code in `calypsox/apps/trading/`

`CustomBOTradeDisplay.java`

12.10 Creating a Custom Fee Calculator

Do the following to create a custom fee calculator:

1. Create a class named **`tk.bo.<fee_method>FeeCalculator`** which implements the interface **`com.calypso.tk.bo.FeeCalculator`**. Implement the following methods:
   - *calculate()* — calculates the fee amount.
   - *calculateInverse()* — reverts the amount computed and stored on the Fee. For example, if you have a Fee Amount of 1,000 expressed as a percentage of the notional, its purpose is to display the initial percentage amount, 2%.
   - *getDescription()* – returns a fee description.

   This class will be invoked from **`com.calypso.tk.bo.FeeDefinition`**.
2. Register the new fee calculator in the feeCalculator domain.

Sample Code in `calypsox/tk/bo/`

`CustomerTransferFeeCalculator.java`

12.11 Three-Party Trades

12.11.1 Installing

1. Edit the file: **`calypsox/apps/trading/SampleThreeTradeValidator.java`**.
2. Replace "SampleThreeTrade" by "CustomTrade" and save the file as: **`calypsox/apps/trading/CustomTradeValidator.java`**.
3. Compile:

   `calypsox/apps/trading/CustomTradeValidator.java`

```
calypsox/apps/trading/ThreePartyJPanel.java
calypsox/tk/bo/workflow/rule/ThreePartyTradeRule.java
```

12.11.2 Configuring

1. Add the rule ThreeParty to the available trade rules using the Workflow Config.
2. Add the necessary trade keywords (if some required keywords are not set, the system will give you an error messages when trying to save the Three Party trade).
   - 3PartyType
   - NumTrades
   - Book, Book2, Book3 (also Book*n* if you intend to use *n* trades)
   - Location, Location2, Location3 (*same remark as book*)
   - Direction, Direction2, Direction3 (*same remark as book*)
   - Cpty, Cpty2, Cpty3 (*same remark as book*)
   - Role, Role2, Role3 (*same remark as book*)
   - SeqNo

12.11.3 Customizing the Three Party Trade

If you wish to add more automatic schemes, you must customize **`ThreePartyJPanel.java`** and **`ThreePartyTradeRule.java`**.

13 Trade Lifecycle

13.1 How to Create a Custom Allocation Process

Create a class named **`tk.product.<product_type>ProductAllocator`** which implements the interface **`com.calypso.tk.product.ProductAllocator`**.

This class will be invoked from **`com.calypso.tk.product.ProductAllocatorUtil`**.

13.2 Creating a Custom Corporate Actions Handler

Create a class named **`tk.product.<product_type>CorporateActionHandler`**, **`tk.product.<product_family>CorporateActionHandler`** or **`tk.product.DefaultCorporateActionHandler`** that implements **`com.calypso.tk.product.CorporateActionHandler`**.

This class will be invoked from **`com.calypso.tk.product.CorporateActionHandlerUtil`**.

13.2.1 Customizing Actions for Corporate Action

Create a class named **tk.product.CustomCATradeActionLookup** that implements **com.calypso.tk.product.CATradeActionLookup**.

The method *getTradeAction()* can change the action to be selected when a corporate action is applied or amended. The input parameters are the original action, the trade, and the CorporateAction.

It will be invoked from **com.calypso.tk.product.CorporateActionHandlerUtil**. It applies to both the manual Corporate Action process, and the CORPORATE_ACTION scheduled task.

Sample Code in calypsox/tk/product/

CustomCATradeActionLookup.java

Sample to change AMEND to UPDATE.

13.2.2 Custom Application of Corporate Actions

Create a class named **tk.product.<*product_type*>CAOptionManager** that implements **com.calypso.tk.product.CAOptionManager**.

Sample Code

```
package calypsox.tk.product;
import com.calypso.tk.core.Defaults;
import com.calypso.tk.core.JDate;
import com.calypso.tk.core.JDatetime;
import com.calypso.tk.core.Product;
import com.calypso.tk.product.CAOption;
import com.calypso.tk.product.CAOptionManager;
import com.calypso.tk.product.Warrant;
import com.calypso.tk.service.DSConnection;

public class WarrantCAOptionManager implements CAOptionManager {

  public CAOption instanciateCAOption(Product product, JDate applicationDate, JDatetime
processDateTime, boolean manageIssuance){
    CAOption result = null;
    if (product instanceof Warrant) {
      Warrant warrant = (Warrant) product;
      result = new CAOption();
      result.setUnderlying(warrant);
      result.setDeliveryType(warrant.getDeliveryType());
      result.setParityDenominator(warrant.getParityDenominator());
      result.setParityNumerator(warrant.getParityNumerator());
      result.setExDate(applicationDate);
      result.setNotificationDate(applicationDate);
      result.setRecordDate(applicationDate);
      result.setValueDate(warrant.getDeliveryDate(applicationDate));
      result.setEnteredDatetime(processDateTime);
      result.setCurrency(warrant.getCurrency());
      result.setComment("CA Generated by CAOptionEntryManager");
      result.setEnteredUser(DSConnection.getDefault().getUser());
      if (Defaults.isAutoFeedDeliveryQuote()) {
```

```
        result.setDeliveryQuote(warrant.getStrike());
        }
      result.setHandleIssuance(manageIssuance);
      }
    return result;
  }

}
```

13.3 Exercise and Expiration

13.3.1 Creating a Custom Exercise Process

Create a class named **tk.product.<product_type>Exercisable** which implements the interface **com.calypso.tk.product.Exercisable**.

This class will be invoked from **com.calypso.tk.product.OptionExerciseUtil**.

13.3.2 Applying Custom Validation to the Exercise Process

Create a class named **tk.product.<product_type>ExerciseValidator**, **tk.product.<product_family>ExerciseValidator**, or **tk.product.DefaultExerciseValidator** that implements **com.calypso.tk.product.ExerciseValidator**.

This class is invoked from **com.calypso.tk.product.OptionExerciseUtil**.

Sample Code in calypsox/tk/product/

DefaultExerciseValidator.java

13.3.3 Applying Custom Validation to the ETOExcerciseWindow

Create a class named **apps.reporting.CustomFutureOptionExerciseExpiryValidator** that implements the interface **com.calypso.apps.reporting.FutureOptionExerciseExpiryValidator**.

This class will be invoked from **com.calypso.apps.reporting.ETOExerciseWindow**.

Sample Code in calypsox/apps/reporting/

CustomFutureOptionExerciseExpiryValidator.

13.3.4 Applying Custom Validation to the FutureExpiryWindow

Create a class named **apps.reporting.CustomFutureExpiryValidator** that implements the interface **com.calypso.apps.reporting.FutureExpiryValidator**.

This class will be invoked from **com.calypso.apps.reporting.FutureExpiryWindow**.

Sample Code in calypsox/apps/reporting/

CustomFutureExpiryValidator.

13.4 Creating a Custom Price Fixing Handler

To perform any additional processing during price fixing — invoked from the Price Fixing window when the user publishes the price fixings.

Create a class **apps.reporting.CustomPriceFixingHandler** which implements the interface **com.calypso.apps.reporting.CustomPriceFixingHandlerInterface**.

This class will be invoked from the **com.calypso.apps.reporting.PriceFixingFrame** to perform any additional processing during price fixing.

Sample Code in calypsox/apps/reporting/

CustomPriceFixingHandler.java

13.5 Creating a Custom Rollover Process

Currently, Calypso implements rollover for treasury products such as Loans and Deposits, Repos, and FX products. The rollover process for Calypso Fixed Income products implements FIRollOver, and for Calypso FX products ForexRollOver.

Create a class named **tk.product.<product_type>RollOver** which implements the interface **com.calypso.tk.product.FIRollOver** or **com.calypso.tk.product.ForexRollOver**.

This class will be invoked from **com.calypso.tk.product.RollOverUtil**.

13.6 Termination

13.6.1 Creating a Custom Termination Process

Create a class named **tk.product.<product_type>Termination** or **tk.product.DefaultTermination** which implements the interface **com.calypso.tk.product.Termination**.

This class can also be used for transferring trades, provided the following methods are implemented:

- *transferTrade()* — Transfers a Trade to a new book or a new counter-party.
- *filterFlows()* — Filters flows for a Trade terminated or transferred. This function is called from the getFlows method attached to the Product when the flag addTradeFlows is set to true. The purpose of

this function is to remove or add any flows based on the Termination/ Transfer information. It can be redefined for each Product.

- *isTradeTransferrable()* — Returns true if the Trade can be transferred.

This class is invoked from:
`com.calypso.tk.product.TerminationUtil`.

13.6.2 Creating a Custom Termination Dialog

Create a class named `apps.product.<product_type>TerminationDialog` that implements the interface `com.calypso.apps.product.TerminationDialog`.

This class will be invoked from `com.calypso.apps.product.TerminationDialog`.

13.7 Adding Custom Menu Items to the Process Trade Window

Create a class named `apps.util.CustomProcessTradeMenu` which implements the interface `com.calypso.apps.util.CustomProcessTradeMenu`.

This class will be invoked from `com.calypso.apps.util.ProcessTradeUtil`.

13.8 Creating a Custom Attribute Matching Mechanism

Create a class named `tk.bo.matching.<matching name>MatchingAttributes` that implements `tk.bo.matching.MatchingAttributes`.

This class will be invoked from `com.calypso.tk.bo.matching.DefaultMatchingUtil` when performing attributes matching in the Financial Matching Window.

14 Reporting

14.1 Report Framework Overview

The Calypso Report Framework has been designed to clearly separate the various elements of a report:

- Report Template — The input parameters for the report.
- Report — The database query to retrieve the data.
- Report Output — Data model of the data retrieved by the report.

- Report Style — Utility to extract atomic values (columns) from the Report Output.
- Report Viewer — An interface to "render" or display the report output to the end-user.
- Report Window — A single GUI window drives all the reports. In so doing, a new report can easily be implemented and plugged into the system, immediately inheriting of all the services available in the report framework: Export to Excel, HTML, PDF, Aggregation functions, sorting, etc.

14.1.1 Defining Report Templates

The ReportTemplate and its subclasses provide the means to define search query parameters and what data the report should contain. It allows this information to be stored in the database for use in Report windows and/or in running Report Scheduled Tasks.

For example, a user wants to create a custom report with the following parameters:

- Currency
- Min and Max Amount
- Display Derivatives
- Start Date and End Date

The class would be named **tk.report.<CustomObject>ReportTemplate** and should extend ReportTemplate. It will look like:

```
public class CustomObjectReportTemplate extends ReportTemplate {
public static final String CURRENCY="Currency";
public static final String MIN_AMOUNT="MinAmount";
public static final String MAX_AMOUNT="MaxAmount";
public static final String DERIVATIVES_FLAG="DerivativesFlag";
  public void setDefaults() {
    Hashtable contents = getContents();
    contents.put(DERIVATIVES_FLAG, new Boolean(false));
}
}
```

The *setDefaults()* method allows setting default query parameters for the report.

The base class "ReportTemplate" handles the following parameters:

Table 14-1: ReportTemplate Parameters

Parameter	Description
Description	A free-form String description of this template
Start Date and End Date	Start Date cutoff for the report defined as an absolute date or as a relative tenor, and End Date cutoff for the report defined as an absolute date or as a relative tenor. Start and end dates are used in most reports, so it makes sense to provide the parameters as defaults in the base ReportTemplate class. However, it is to the discretion of the Report as to whether these are used or not. Most of these "input" parameters useful will only be if the Report class makes use of it; in other words, if it uses these parameter values in generating the query.
Holidays	Vector of Holidays to use when calculating dates.
Business Days	Boolean flag to differentiate between Business and Calendar Days.
Columns	Columns to be displayed in the Report. The Columns parameter enables you to select which columns to display in the output report. The selection of columns depend on the report and the selection is retrieved from the ReportStyle class and for each column that is defined, the ReportStyle class must code the logic on how that value is to be extracted from the Report row.
Sort Columns	Columns by which the report data should be sorted. The SortColumns parameter allows you to dictate how the data rows are to be sorted. The user can specify one or more sorting columns. For example, some reports must be sorted by Book, then by trade ID, while others may simply require sorting by date. The set of Sort Columns available is taken from the set of Columns mentioned above and it is not possible to sort on a column that is not included in the Columns parameter.
Subheadings	Columns to be displayed as "subheadings". Some reports must show subheadings. This only applies when sorting is being used and the columns to be shown as subheadings are chosen from those selected as sort columns. For example, to sort by Settlement Date, select the "Settlement Date" column as a subheading, this will result in the value being shown on its own row, with the rows that match its value being demarcated underneath.
Subtotals	Columns for which subtotals should be displayed. It is possible to tag columns for which subtotals should be computed. This only applies to columns that represent numeric values such as amounts. Attempting to generate a subtotal for a Legal Entity Name, say, will result in a subtotal of 0.0. Subtotals are shown whenever a break occurs in the sorting order. If a Transfer report is being sorted by Settlement Date and subtotaled by Transfer Amount the subtotal for all transfers matching the specific settlement date will be displayed. Subtotals are reset to 0 after each break in the sorting order.

Table 14-1: ReportTemplate Parameters (Continued)

Parameter	Description
Totals	Columns for which totals should be displayed. Totals are cumulative subtotals. The only difference between a Total and a Subtotal is that the total is not reset at each break in the sorting order but a tally is kept for each row in the report.
Subtotal Functions and Total Functions	Aggregation functions to use in calculating subtotals and totals. For each column defined as a subtotal or total, you can use an aggregation function to calculate that subtotal. The default function is Sum but other functions are available out-of-the-box including Maximum, Minimum, and Average.
AggregationFlag	There are times when an aggregated view of the report is required without displaying all row details. If the flag is set to true (default being false) then only an aggregated view will be displayed in the report. Only rows with subheadings, subtotals, and totals will be displayed while the specific row details will be hidden from view. Of course, the subtotals and totals will include these hidden rows; they simply will not be visible in the report. Columns, Sort Columns, Subheadings, Totals, Subtotals, and the associated Function parameters are used to control what data is displayed, how it is sorted, and if and how it is aggregated into groups (with subheadings and subtotals.) Although how the data is displayed may differ depending upon the ReportViewer (HTML will look different from a GUI table), the actual data should remain the same across all viewers. As far as data persistence is concerned, this is managed by the **tk.report.sql.ReportTemplateSQL** class and, since it implements the **tk.core.Attributable** interface, most of the attributes are stored in the entity_attributes table.

14.1.2 Defining Reports

The Report class is where a query is built based on the input parameters in the ReportTemplate. The query is built; a call is made to the Calypso Data server (via the remote API) and the returned data is a ReportOutput with a set of ReportRows. A ReportRow identifies the objects retrieved from the system on a per-row basis.

The reason for having the ReportRow objects (as opposed to simply passing the objects Vector itself) is because there are times when one must associate several different objects to one row. For example, a Message Report will sometimes have a Message, a Transfer, and a trade all attached to the same ReportRow.

Continuing with the earlier example, <CustomObject>Report would be as follows:

```
class CustomOjectReport extends Report {
  public ReportOutput load() {
    initDates();
    DefaultReportOutput output = new DefaultReportOutput(this);
    Hashtable from = new Hashtable ();
    String where = buildQuery(from);
```

```
    Vector objects = null;
    try {
      …
      RemoteCustom rc = ds.getRemoteCustom();

      objects = rc.getObjects(where, from);
      …
    }
    catch (Exception e) { … }

ReportRow[] rows = new ReportRow[objects.size()];
for (int i=0; i < rows.length; i++) {
rows[i] = new ReportRow(objects.get(i));
}
output.setRows(rows);

return output;
}

…
public String buildQuery(Hashtable from) {
    // go through the CustomObjectReportTemplate and build
    // the where query based on the values set.
    // Add the associated tables to from Hashtable
    …
  }
}
```

14.1.3 Defining Report Outputs

The core class that "handles" the report output is **com.calypso.tk.report.DefaultReportOutput**. Its purpose is not to render or display the data. This task is left to the report viewers. Its purpose is to arrange the report data which was set by the report with the *setRows()* method in a way compatible with the parameters given in the ReportTemplate (Columns, SortColumns, Subtotals, etc.)

Hence, the DefaultReportOutput class will sort the rows based on the SortColumns. It will parse through each row and calculate the subtotals and totals, and pass that information along to the ReportViewer.

14.1.4 Defining Report Styles

ReportStyles are helper classes which provide the functionality to extract column values from ReportRow objects. A ReportRow encapsulates one or more Calypso objects. It is necessary to extract a column value from the object because the column might be a direct mapping to an object field value (e.g. TRADE_ID column maps to *Trade.getId()* method). Other column values might be computed and/or generated when it is queried.

ReportStyle classes are extensible. For example, the SettlementReportStyle extends TransferReportStyle, which extends TradeReportStyle, which extends ReportStyle. The behavior to extract a column value is inherited by extending from a superclass.

The ReportStyle classes provide one core method which provides all of the functionality for the class, for example:

```
  protected Object getColumnValue (ReportRow row, int columnId) {
   …
   // Extract column value (columned) from the ReportRow.
// If there is no match, you can delegate the class to have it extract the
// value.
return super.getColumnValue(row, columnId);
   }
// The following is a more specific example of the TransferReportStyle and
// provides a short excerpt of how it is implemented:
  protected Object getColumnValue (ReportRow, row, int columnId) {
   BOTransfer transfer = (BOTransfer)row.getProperty(ReportRow.TRANSFER);

   if (columnId == ID_AVAILABLE_DATE) {
   return transfer.getAvailableDate();
   }
   else if (columnId == ID_DELIVERY_TYPE) {
   return transfer.getDeliveryType();
   }
   …
   return super.getColumnValue(row, columnId);
}
```

An additional note is required regarding TradeReportStyle. In order to place Product-specific columns in their own style classes and permit clients to extend the framework easily, TradeReportStyle proceeds to search for matching column name(s) if it does not find it in its own set of columns. First, it will iterate over product interfaces since these report styles span multiple Products. All product interfaces are defined in the productInterface domain values and the following Report Styles are packaged with Calypso: OptionReportStyle, CashSettledReportStyle, among others. If the column is not located in any of the available interface report styles, then TradeReportStyle attempts to spawn a product specific ReportStyle based on the product type (i.e., SwapReportStyle, BondReportStyle, etc.)

14.1.5 Defining Report Viewers

The Report Viewer renders the report output. Calypso currently provides the following report viewers: HTML, Excel, PDF, CSV, and GUI tables. ReportWindow provides the ability to view any report implemented using the framework. A customer need not worry about implementing a GUI for a report, the report is immediately available via a GUI interface.

A report viewer implements the **`com.calypso.tk.report.ReportViewer`** interface.

Out-of-the-box, Calypso provides the following report viewers:

- **`com.calypso.tk.report.HTMLReportViewer`**
- **`com.calypso.tk.report.ExcelReportViewer`**

- **com.calypso.tk.report.PDFReportViewer**
- **com.calypso.tk.report.CSVReportViewer**
- **com.calypso.apps.reporting.TableReportViewer**
- **com.calypso.apps.reporting.TreeTableReportViewer**
- **com.calypso.apps.reporting.PivotTableReportViewer**

14.1.6 Report Window

All reports that are built on top of the Reporting Framework share the same default GUI code. At its simplest level, a report can be added to any GUI window by adding a new ReportPanel to an existing GUI window:

```
// Insert a Message Report Panel
JPanel messagePanel = new ReportPanel("Message");
```

Of course, this only provides the ability to **embed** a Report in an existing window and does not provide a way to set search criteria (ReportTemplate) on this report. This would must be done programmatically elsewhere in the code.

For most reports, it will often be more practical to use ReportWindow:

```
 // Create a new Message Report Window and make it visible
JFrame frame = new ReportWindow("Message");
Frame.setVisible(true);
```

All reports embedded in the ReportWindow have a similar look and feel.

<table>
<tr><td colspan="2">Search Criteria (ReportTemplatePanel)</td></tr>
<tr><td>Report Filter
(BookHierarchy)</td><td>Report Viewer (ReportPanel)</td></tr>
<tr><td colspan="2">Additional Parameters (Buttons, PricingEnv, etc.)</td></tr>
</table>

The search criteria for the reports are selected at the top in the ReportTemplatePanel. Some control buttons and additional environment settings are available at the bottom of the window in the button panel. There, you can set the Pricing Env, the Valuation Date, and connect to real-time events. These options are only available when applicable.

Given the argument passed to the Report Window ("Message" for example), the following GUI classes will be instantiated and attached to the Report Window (if and when applicable):

- **apps.reporting.MessageReportTemplatePanel**
- **apps.reporting.MessageReportWindowCustomizer**
- **apps.reporting.MessageReportRealTimeHandler**

The corresponding toolkit classes are also instantiated:

- **tk.report.MessageReportTemplate**
- **tk.report.MessageReport**

- `tk.report.MessageReportStyle`

14.1.7 Import/Export

It is possible to import/export report template parameters from/to XML. This provides a convenient way to distribute a suite of out-of-the-box reports to one or more users. The report templates can be stored in a directory and imported into the system on an as-needed basis.

Because the template parameters are stored as attributes, it should be unnecessary to modify or extend the import/export functionality. For reference, the relevant classes can be found in the following package: `com.calypso.bridge.object.reportTemplate`.

14.2 How to Add a New Report

In order to add a new report, create a new ReportTemplate (which may or may not extend an existing one), a new Report, and a new ReportStyle, and place them in the `tk.report` package. For example, in order to create a CustomObject report three classes would be created:

- `tk.report.CustomObjectReportTemplate`
- `tk.report.sql.CustomObjectReportTemplateSQL`
- `tk.report.CustomObjectReport`
- `tk.report.CustomObjectReportStyle`

Add the domain value ("CustomObject" in this example) to the "REPORT.Types" domain, and add a menu item for the report using **Main Entry > Utilities > Main Entry Configurator** as follows:

- Name — Custom Object Report
- Action — `reporting.ReportWindow$CustomObject`

After re-starting Main Entry, it will be possible to access the new report from the Reports menu, and to input the appropriate parameters, load/save templates and, run the report.

14.2.1 How to Create a Report Template Panel

However, in order to be generic, the GUI to manipulate the ReportTemplate is not the most user-friendly. It lists most of the parameters in tabular format and does not do any kind of type-checking (Some minor type-checking is done, whenever possible, by parsing the parameter names). This is certainly not failsafe, however. The limitation here was that the ReportTemplate, which has been available in Calypso for several years, had to remain backward-compatible. Since the template is stored in the database as a Hashtable object, there is no possibility to easily upgrade it.

However, there is a way to customize how the user will interact with your template. All that is needed is to create a class named `apps.reporting.CustomObjectReportTemplatePanel` which extends `com.calypso.apps.reporting.ReportTemplatePanel`. This abstract class extends JPanel and provides two methods to set and get the

ReportTemplate in order to determine how the report template should be displayed on the GUI:

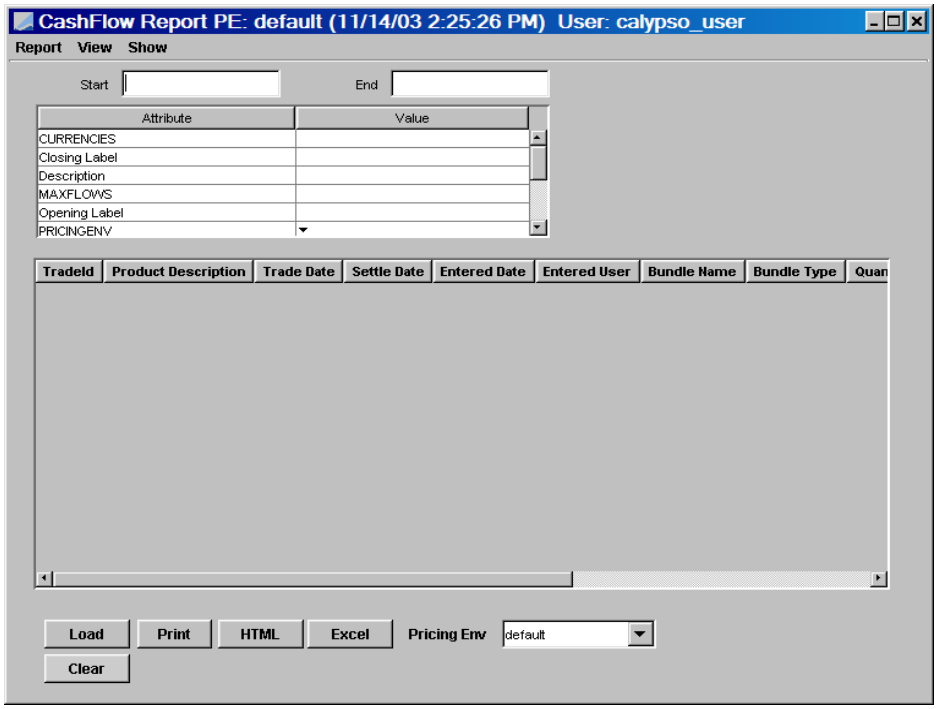
- setTemplate()
- getTemplate()

This panel, once compiled, will automatically be loaded and inserted at the top of the ReportWindow.

- The following ReportTemplatePanels are provided by Calypso and are used in the respective report windows. By launching these various reports, you can see the customized template GUI:
  - AuditReportTemplatePanel
  - DailyBlotterReportTemplatePanel
  - FailsReportTemplatePanel
  - FeeReportTemplatePanel
  - SettlementReportTemplatePanel

In the case of a simple, straightforward report template, the implementation of this class is not required; rather the DefaultReportTemplatePanel is used and will display the report parameters in tabular format, as shown below:

Figure 14-1: DefaultReportTemplatePanel — (Cashflow Report PE)



14.2.2 How to Add a Custom Menu and Custom Processing

The report provides a convenient GUI to manipulate a vast number of objects at one time. The window can be extended by adding a menu with

custom functionality to, for example, allow "manipulation" rather than simply viewing of data.

Continuing with the Custom Object report example, this would be achieved by creating a class named **apps.reporting.CustomObjectReportWindowHandler** that extends **com.calypso.apps.reporting.ReportWindowHandlerAdapter**. These methods can be implemented:

- *public void customizeReportWindow(ReportWindow reportWindow)* — This is where the customization of the report can be done.
- *public boolean callBeforeClose(ReportWindowDefinition definition)* — This method returns true or false, true to authorize the report to be closed, or false to prevent the report to be closed if additional processing must be done like publishing quotes, etc.
- *public void customizeMenuBar(JMenuBar menuBar, RiskPresenterWorker worker)* — This method provides the ability to customize the menu bar within the context of the given RiskPresenterWorker.
- *public void customizePopupMenu(JPopupMenu popup, RiskPresenterWorker worker)* — This method provides the ability to customize the popup menu within the context of the given RiskPresenterWorker.
- *public void callAfterLoadAll(ReportWindow reportWindow)* — Called at the end of ReportWindow.loadAll(), i.e., after the load process has been started for each report in each tab.
- *public boolean isValidLoad(ReportPanel panel)* — Returns a true if the load was successful.

14.2.3 How to Create a Custom Aggregation Function

Create a class named **tk.report.function.<function_name>** that implements **com.calypso.tk.report.function.ReportFunction**.

It will be invoked from **com.calypso.tk.report.function.FunctionFactory**.

Then add the function name to the "**REPORT.Functions**" domain. Out-of-the-box, Calypso provides the following aggregation functions: Count, Sum, Average, Maximum, and Minimum.

14.2.4 How to Create a Custom Sorting Comparator

Create a class named **tk.report.<comparator_name>Comparator** that implements **com.calypso.tk.report.RowComparator**.

It will be invoked from **com.calypso.tk.report.DefaultReportOutput**.

14.2.5 How to Validate a Custom Report Filter

Create a class named **apps.reporting.<report_name>CustomReportFilter** that imple-

ments the interface **com.calypso.apps.reporting.CustomReportFilter**.

This is currently supported by the following reports: Audit, CRE, Message, Payment, Posting, Task Station and Trade.

It is also supported by the report framework through **com.calypso.apps.reporting.ReportPanel**.

Sample Code in calypsox.apps.reporting:

- **AuditCustomReportFilter**
- **CreCustomReportFilter**
- **MessageCustomReportFilter**
- **PaymentCustomReportFilter**
- **PostingCustomReportFilter**
- **TaskCustomReportFilter**
- **TradeCustomReportFilter**.

14.3 How to Customize the Transfer Viewer

Note that the Transfer Viewer is a utility that is only available if the environment property USE_TRANSFER_VIEWER is true. When you right-click a transfer, choose **Show > Transfer Viewer** from the popup menu to display all elements associated with the transfer.

The Transfer Viewer can be customized as follows:

- Adding panels.
- Displaying more data in the Main panel.

Create a class named **apps.reporting.CustomTransferViewerWindow** that implements **com.calypso.apps.reporting.TransferViewerInterface**.

You can implement the following methods:

- *getTransferMainPanel()* — for customizing of the main panel.
- *getTransferTabPanels()* — for adding a panel.
- *showTransfer()* — for displaying additional transfer data.

14.4 How to Customize the Quick Search Window

You can customize the search criteria, the searched objects, and the display of the objects.

Create a class named **apps.reporting.<name>Interface** that implements **com.calypso.apps.reporting.QuickSearchInterface**.

Sample Code in calypsox/apps/reporting/

- QuickSearchWindowInterface.java

14.5 Trade Bulk Entry API

Knowledge of the Calypso Report API is necessary to utilize this API. Please refer to Report, ReportStyle, ReportTemplate, ReportTemplatePanel, and ReportWindowHandler in this guide.

The viewable fields in the TradeBrowser are also available as fields in BulkEntryReport.

BulkEntryReport extends TradeReport. But instead of loading trades from database, BulkEntryReport.load() loads trades from CSV file — a BulkEntryReportTemplate property.

BulkEntryReportStyle extends TradeReportStyle. The display is the same as that of the TradeBrowser, using `TradeReportStyle.getColumnValue()` — an embedded `ProductReportStyle.getColumnValue()` `BulkEntryReportStyle.getPossibleColumnNames()` overrides and restricts `TradeReportStyle.getPossibleColumnNames()` to the editable columns provided by `<productType>BulkEntryItem.getPossibleColumnNames()`

Adding Support for Additional Product Types

1. If it does not exist, add `tk.report.<productType>ReportStyle`.
2. Add `tk.util.bulkentry.<productType>BulkEntryItem` extends `TradeBulkEntryItem` and override:
   - `List<String> getPossibleColumnNames()`: the editable column names (a subset of `TradeReportStyle.getPossibleColumnNames()`)

     The order of the returned column names is not relevant. In a CVS import, Trade/Product fields are set in sequence following this order.
   - `SortedSet<String> getColumnChoices(String column)`: Use to restrict field values to a list of choices. Also used for data validation
   - `void setValue(Trade trade, String column, String value)`: Sets a Trade/Product attribute.

14.6 How to Enable Generic Comments for an Object

To enable generic comments for an object, create a class named `tk.bo.sql.DefaultCommentableObjectSQL` that implements `tk.bo.sql.CommentableObjectSQL`.

It will be invoked from the BackOffice RMI server.

This interface must implement the two following methods:

- public ObjectDescription getCommentableObject(int objectId, String objectClass, Connection dbCon) throws PersistenceException;

  objectClass: "Trade", "Transfer", "Message", "Posting", "YourObjectClass", etc.
- public GenericComment[ ] getObjectComments(ObjectDescription objectDesc, Int showType, String whereClause, Connection dbCon) throws PersistenceException;

  showType: GenericComment.SHOW_ALL, GenericComment.SHOW_PARENTS, GenericComments.SHOW_CHILDREN, GenericComment.SHOW_ALL, GenericComment.SHOW_NONE (no "Show" directive)

Sample Code in `calypsox/tk/bo/sql/`

`DefaultCommentableObjectSQL.java`

15 Risk Analysis

15.1 Analysis

15.1.1 How to Create a Custom Analysis

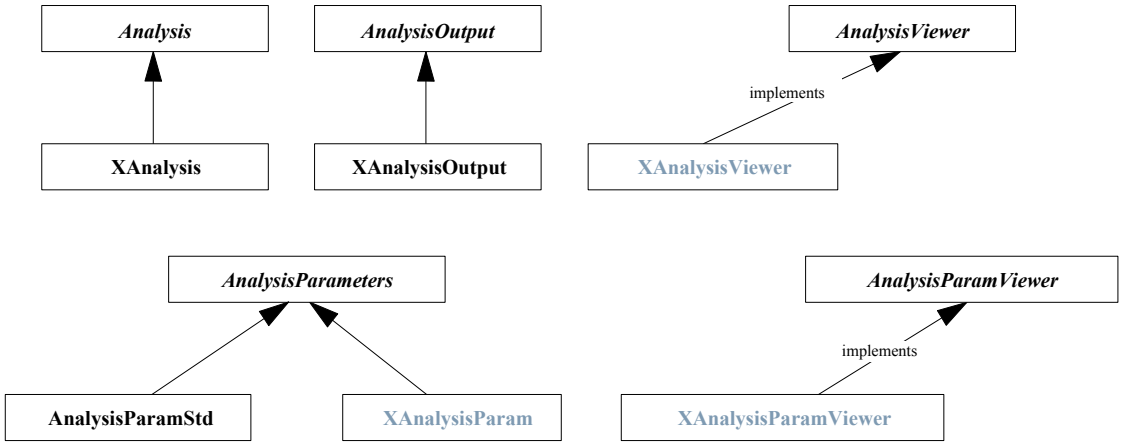
To add your own analysis, create a subclass of Analysis, for example XAnalysis (see illustration below), that implements the *run()* method, and creates an AnalysisOutput class, XAnalysisOutput, that will format the results of your analysis class.

To display your results, Calypso provides a standard viewer, DefaultAnalysisViewer, which can display any report that is a spreadsheet-style table of values. If you wish to create a different type of display, then you can create a class that implements the AnalysisViewer interface. Whatever viewer you choose, you must register it in the Analysis Viewer Config window

Standard analysis parameters are specified in AnalysisParamStd. You may must subclass AnalysisParamStd to add custom parameters, and create a custom parameters viewer that implements AnalysisParamViewer for editing these parameters.

The following class diagram shows you the classes involved in creating a custom analysis.

Figure 15-1: Example of Classes used to Create a Custom Analysis



Class hierarchy diagram: Adding your own Analysis. Optional classes are shown in grey.

Overview of Steps

- Step 1 — Create analysis parameters if applicable
- Step 2 — Create an analysis parameters viewer if applicable
- Step 3 — Create an AnalysisOutput
- Step 4 — Create an Analysis
- Step 5 — Register the new Analysis

Step 1 — Create Analysis Parameters

This step is only necessary if your analysis requires custom parameters.

Do the following for creating the analysis parameters:

1. Create a class named **`tk.risk.<analysis_name>Param`** which extends **`com.calypso.tk.risk.AnalysisParamStd`**.

   This class will be invoked from your Analysis class.

2. To make the analysis parameters persistent, create a class named **`tk.risk.sql.<analysis_name>Param`** which extends **`com.calypso.tk.risk.sql.AnalysisParamStdSQL`**.

3. Register individual parameters using **Main Entry > Configuration > System > Analysis Parameter Name**.

Sample Code in `calypsox/tk/risk/`

AnalysisParamStd samples:

- `DEMOParam.java`
- `ABCParam.java`

The DEMO analysis requires a frequency to generate time buckets and a number of threads for multi-threading.

Sample Code in `calypsox/tk/risk/sql/`

AnalysisParamStdSQL samples in:

- `DEMOParamSQL.java`
- `ABCParamSQL.java.`

Step 2 — Create an Analysis Parameters Viewer

This step is only necessary if your analysis requires custom parameters to be edited by the user.

Create a class named **`apps.risk.<analysis_name>ParamViewer`** which implements the interface **`com.calypso.apps.risk.AnalysisParamViewer`**.

This class will be invoked from your Analysis Parameters class.

Sample Code in `calypsox/apps/risk/`

- `ABCParamViewer.java`
- `DEMOParamViewer.java`

Step 3 — Create an AnalysisOutput

Create a class named **`tk.risk.<analysis_name>Output`** that extends **`com.calypso.tk.risk.AnalysisOutput`**.

Implement the following members and methods:

- A member to store the report. You can model the report as a Vector of TradeItem objects (each TradeItem object represents a row in the report). Note that you can subclass TradeItem as applicable.
- A method for building the report, for example *addItem()* to add rows to the report.
- Methods that will allow the AnalysisViewer to display, print, save, export, aggregate the report. For example, when using the Default-AnalysisViewer you will implement the following methods:
  - *getNumberOfRows()* to return the number of rows in the report
  - *getNumberOfColumns()* to return the number of columns in the report
  - *getHeaderAt(int col)* to return the heading text for a given column
  - *getColumnClassAt(int col)* to return the datatype of the data for a given column
  - *getValueAt(int row, int col)* to return the value for a given cell
  - *getAggregationSource()* to return the aggregation criteria

This class will be invoked from your Analysis class.

Sample Code in `calypsox/tk/risk/`

AnalysisOutput samples:

- `ABCOutput.java`
- `DEMOOutput.java`

TradeItem samples:

- `TradeABCItem.java`
- `DemoItem.java`

Step 4 — Create an Analysis

The Analysis class actually performs the analysis.

Create a class named **`tk.risk.<analysis_name>Analysis`** that extends **`com.calypso.tk.risk.Analysis`**.

Implement the following methods:

- *run()* — Returns an AnalysisOutput object (your custom AnalysisOutput) that contains the results of the analysis. It takes the following parameters:
  - a Vector of trades to be analyzed
  - a valuation date
  - a PricingEnv containing market data
  - an AnalysisParameters object, AnalysisParamStd or your custom AnalysisParamStd
- *getParamNames()* — Returns a list of all the parameters required by the Analysis class.

This class will be invoked from **`com.calypso.tk.risk.AnalysisUtil`**.

Sample Code in `calypsox/tk/risk/`

- `ABCAnalysis.java`
- `DEMOAnalysis.java`

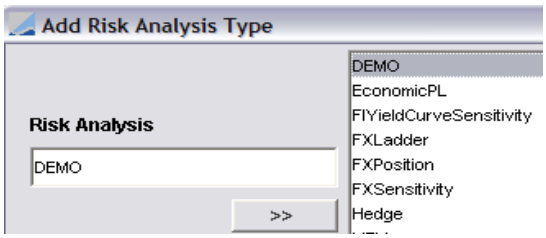
DEMO reports NPV and Break even rate for trades are grouped in user-defined time buckets by maturity date. The analysis makes use of the multi-threading capability of the system.

Step 5 — Register the new Analysis

Do the following to register a new analysis:

1. Add the analysis name to the riskAnalysis domain using the Add Risk Analysis Type window as shown below. This window is accessed from the Risk Analysis window when you click the ... button next to the Analysis Type field.

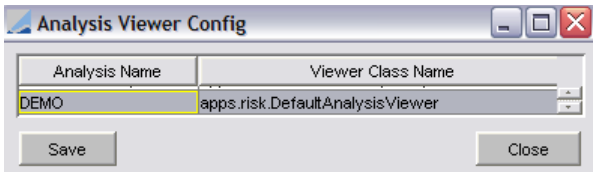
Figure 15-2: Add Risk Analysis Type Window



Save and click "Update Domains" in the Risk Analysis window.

2. Associate the Analysis with an AnalysisViewer using **Main Entry > Configuration > System > Analysis Viewer Configuration** as shown below.

Figure 15-3: Analysis Viewer Config Window



You can use the DefaultAnalysisViewer or create a custom AnalysisViewer as described in the following section.

15.1.2 How to Create a Custom AnalysisViewer

The default AnalysisViewer is **`com.calypso.apps.risk.DefaultAnalysisViewer`**.

Create a class named **`apps.risk.<viewer_class_name>`** which implements the interface **`com.calypso.tk.risk.AnalysisViewer`**.

This class will be invoked from **`com.calypso.tk.risk.AnalysisViewerConfig`**. You can associate the AnalysisViewer with a given Analysis using **Main Entry > Configuration > System > Analysis Viewer Configuration**.

15.1.3 How to Create a Custom Aggregation for an AnalysisViewer

Create a class named **`tk.util.CustomAggregation`** that implements the interface **`com.calypso.tk.util.AggregationInterface`**.

This class will be invoked from **`com.calypso.tk.util.Aggregation`**.

Sample Code in `calypsox/tk/util/`

`SampleCustomAggregation.java`

15.1.4 How to Create a Custom AnalysisHandler

Create a class named **`tk.risk.<analysis_name>Handler`** or **`tk.risk.DefaultAnalysisHandler`** that implements **`com.calypso.tk.risk.AnalysisHandler`**.

This class will be invoked from **`com.calypso.tk.risk.AnalysisUtil`** for exporting AnalysisOutput data to the AnalysisViewer.

15.1.5 How to Create a Custom Analysis Input Verifier

Analysis performs calculations over an AnalysisInput that contains the trades, the PricingEnv, etc. You can create a custom verifier of the input data.

Create a class named **`tk.risk.<analysis_name>Verifier`** or **`tk.risk.CustomAnalysisVerifier`** that implements **`com.calypso.tk.risk.AnalysisVerifier`**.

This class will be invoked from **`com.calypso.tk.risk.AnalysisInput`**.

15.1.6 How to Create Custom Template Keywords

Create a class named **`tk.risk.<analysis_name>RiskFormatter`** that implements **`com.calypso.tk.risk.RiskFormatter`**.

Analysis names are available in the riskAnalysis domain. Implement a *parse<keyword_name>()* method for each custom keyword.

This class will be invoked from **`com.calypso.tk.risk.AnalysisUtil`**.

15.1.7 How to Create a Custom Risk Analysis in the Middle-Tier Servers

The analysis type must be added to the **riskPresenter** domain using the Domain Values window. For example, if the implemented analysis name is MyCustomAnalysis, then you would add "MyCustom" to the **riskPresenter** domain.

Implementation Requirements

AnalysisOutput must implement **PresentableAnaysisOutput** and **Externalizable**. In this case calypsox.tk.risk.XXXCustomAnalysisOutput must implement the mentioned methods.

- Implementation of the **getRiskPresenterHeader(String user)** method is required — this provides the header information used by the risk presenter. The method should construct **RiskPresenterHeaderResult** with the following collections:

1. **All headers** — Map of headers that comprises of BASE_CURRENCY and ELAPSED_TIME.
2. **All columns list** — All column names. In XXXCustomAnalysisOutput, it is the **all headers** string array transformed to a collection as a value tagged to the key "Default."

3. **All aggregation lists** — The entire collection of column names tagged to "Default" key.
4. **All Column Meta Data collection** — For each column, there is a ColumnMetaData constructed from **default_obj_key** that is dimensional, is a key column, is part of the source, is not additive, is not editable, is not numeric, and is not visible. Next, each column depending upon which category it belongs to (from the following bulleted categories), is dimensional, distinct, keyed, numeric (if it is a pricer measure value), and additive. The numeric columns are primarily measure columns and not dimensional. For the sake of the RiskAnalysisOutputModel object creation, it is necessary that there is at least one measure column. Therefore it is a collection of all the columns that is transformed to equivalent ColumnMetaData objects.

   Metadata for the columns of the analysis must be correctly defined (minimally, the following):

   - setNumeric(true/false) — Whether or not the column is numeric.

     Example: Trade Id, NPV.
   - setDimension(true/false) — A dimension is a column that you can aggregate.

     Example: Book, Currency.

     Measure columns such as NPV are non-dimensional
   - setName(String) — The column name.
   - setColumnClass(Class) — The column's class.

     The Supported Data types in Middle Tier are: String, Tenor, Boolean, JDate, JDatetime, DisplayPeriodization, Integer, Double, and DisplayValue (Amount, Rate, Spread, BondPrice, Price)
   - setSource(true/false) — Is the column is the source identifier. If not defined, default_obj_id is set as the source.

- Must implement the **readExternal(ObjectInput)** and **writeExternal(ObjectOutput)** methods with all member attributes that must be serialized.
- Must populate the values for metadata columns, which includes the following columns. This can be done by using a utility method on **AnalysisOutput.populateMetaData(TradeArray arry)**
  - default_obj
  - default_obj_id
  - default_obj_le
  - default_obj_version
  - default_obj_book_id
  - default_obj_product_i
  - default_statu

- default_obj_key
- default_obj_settle_date)
- At a minimum, the column default_obj_book_id, which is used for access permission in presentation server, must be populated.

Implement a translator for the analysis output. The translator should extend **`com.calypso.tk.risk.translator.AbstractTranslator`**. The naming convention is (*XXX*Translator) where *XXX* is the analysis name. The translator should:

- Define an overridable LOG string of "XXXTranslator."
- Implement the abstract method, getData(AnalysisOutput, List<String> columnNames) which will:
  - Return the collection of rows or items in an analysis output with each element in the collection being a map that consist of keys as column names and values
  - For each row, create a collection of cell values for each column as key. This provides an entire 2D data array.

Supported Middle-Tier Data Types

- String
- Tenor
- Boolean
- JDate
- JDatetime
- DisplayPeriodization
- Integer
- Double
- DisplayValue
  - Amount
  - Rate
  - Spread
  - BondPrice
  - Price

15.2 Extending PL Explain

If a new type of market data item is added to the system, a corresponding effect automatically appears in P&L Explained, if a pricer declares its dependency on the item. The pricer must contain an appropriate implementation of the getMarketDataItemIds() method.

If it is necessary to modify the system's default behavior regarding a certain market data item, you need to implement a new class named Cus-

tomMarketDataEffectCalculator. This class must reside in package tk.risk.pl and implement the interface MarketDataEffectCalculator.

Sample code in `calypsox/tk/risk/pl/`

`CustomMarketDataEffectCalculator.java`

If it is necessary to modify the system's default behavior regarding a certain trade lifecycle effect, implement a new class named CustomLifecycleEffectCalculator. This class must reside in package tk.risk.pl and implement the LifecycleEffectCalculator interface.

Sample code in `calypsox/tk/risk/pl/`

`CustomLifecycleEffectCalculator.java`

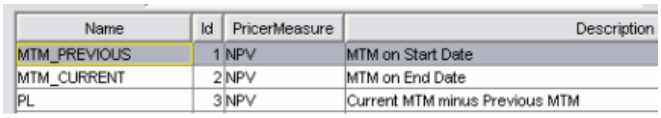
15.3 How to Customize EconomicPLAnalysis

EconomicPLAnalysis explains the variation of Profit & Loss between two dates according to different effects such as Interest Rate change, Trade Amendment, etc. The explanation is based upon a Pricer Measure, varying one component of pricing while leaving all others unchanged.

The explanation is calculated within a calculator which can be product and/or effect specific. The calculator is defined by the interface **`tk.risk.pl.PLCalculator`**. Most calculation effects are implemented within **`tk.risk.pl.DefaultPLCalculator`**. Some products have their own PLCalculator, <product_type>PLCalculator.

Each effect is a column EconomicPLColumn in EconomicPLAnalysis. Existing columns are defined in the eco_pl_column domain. Each column is linked to an id and a PricerMeasure using the Economic PL Column window as shown below.

Figure 15-4: Economic PL Column



Name	Id	PricerMeasure	Description
MTM_PREVIOUS	1	NPV	MTM on Start Date
MTM_CURRENT	2	NPV	MTM on End Date
PL	3	NPV	Current MTM minus Previous MTM

The Economic PL Column window is invoked from the Economic PL Param Viewer when you click **Create New Items**.

To create a custom PLCalculator, create a class named **`tk.risk.pl.<product_type>PLCalculator`** which extends DefaultPLCalculator or an existing PLCalculator for a given product.

Implement the *process()* method.

This class will be invoked from **`com.calypso.tk.risk.pl.PLCalculatorUtil`**.

Sample Code

- Customizing a column from DefaultPLCalculator:

```
public class MY_PRODUCTPLCalculator extends DefaultPLCalculator { }

public void process(EconomicPLCol column, Trade todayTrade,
      Trade yesterdayTrade, Pricer pricer,
      PricerMeasure[] measures, boolean isPositionB)
   throws PricerException {
      switch(column.getType()) {
         case CALYPSO_COLUMN_ID:{
            need to fill the  PricerMeasure[] measures
            pricer.price(…..measures)
         }
         break;
         default:super.process(column,todayTrade,yesterdayTrade,pricer,measures,
            isPositionB);
         break;
      }
   }
```

- Customizing a column from a product PLCalculator:

```
public class MY_PRODUCTPLCalculator extends SwapPLCalculator { }
public void process(EconomicPLCol column, Trade todayTrade,
   Trade yesterdayTrade, Pricer pricer,
   PricerMeasure[] measures, boolean isPositionB)
   throws PricerException {
      switch(column.getType()) {
         case CALYPSO_COLUMN_ID:{
            need to fill the  PricerMeasure[] measures
            pricer.price(...measures)
        }
      break;
      default:super.process(column,todayTrade,yesterdayTrade,pricer,
         measures,
         isPositionB);
       break;
      }
   }
```

- Creating a custom column:

Note: Your Column ID must be greater than 100 to avoid conflicts with Calypso IDs.

The new column must be registered in the **eco_pl_column** domain, and in the Economic PL Column window.

```
public class MY_PRODUCTPLCalculator extends DefaultPLCalculator { }

public void process(EconomicPLCol column, Trade todayTrade,
      Trade yesterdayTrade, Pricer pricer,
      PricerMeasure[] measures, boolean isPositionB)
   throws PricerException {
      switch(column.getType()) {
         case MY_COLUMN_ID:{
```

```
            need to fill the  PricerMeasure[] measures
         }
         break;
         default:super.process(column,todayTrade,yesterdayTrade,pricer,measures,
            isPositionB);
         break;
      }
}
```

15.4 How to Customize ScenarioAnalysis

ScenarioAnalysis allows defining market data scenarios to be applied to a set of trades, and calculates risk measures for those scenarios. You can create custom scenario market data, custom scenario rules, custom report viewers, and custom report viewer converters.

15.4.1 Creating a Custom Scenario Rule

Create a class named **tk.risk.ScenarioRule<name>** that implements **com.calypso.tk.risk.ScenarioRule**.

This class will be invoked from **com.calypso.apps.risk.ScenarioRulePanel**.

You must register the custom rule name with the customScenarioRule domain.

Sample Code in calypsox/tk/risk/

ScenarioRuleCustomZeroInterest.java

15.4.2 Creating a Custom Scenario Market Data

Create a class named **tk.risk.CustomScenarioMarketData** that implements **com.calypso.tk.risk.CustomScenarioMarketDataInterface**.

This class will be invoked from **com.calypso.tk.risk.ScenarioMarketData**.

Sample Code in calypsox/tk/risk/

SampleCustomScenarioMarketData.java

15.4.3 Creating a Custom Report Viewer

Create a class named **tk.risk.<viewer>** that implements **com.calypso.tk.risk.ScenarioReportViewInterface**.

This class will be invoked from **com.calypso.tk.risk.ScenarioReportView** and **com.calypso.apps.risk.ScenarioReportViewWindow**.

The custom viewers must be registered in the domain ScenarioViewerClassNames.

15.4.4 Creating a Custom Report Viewer Converter

Create a class named **tk.risk.<viewer converter>** that implements **com.calypso.tk.risk.ScenarioReportViewConverterInterface**.

This class will be invoked from **com.calypso.tk.risk.ScenarioReportView** to convert the standard columns names to user-defined column names.

Sample Code in calypsox/tk/risk/

ScenarioReportViewConverterSample.java

15.4.5 Creating a Custom Notifcation Before/After Pricing a Trade

The interface CustomScenarioAnalysisInterface has been added, it has the following methods: *beforeApplyingAllRules()* and *afterApplyingAll-Rules()*.

Create a class named **tk.risk.DefaultCustomScenarioAnalysisInterface** that implements **com.tk.risk.CustomScenarioAnalysisInterface**.

This class will be invoked from ScenarioAnalysis.

Sample Code in calypsox/tk/risk/

DefaultCustomScenarioAnalysisInterface.java

15.5 Distributed Processing

15.5.1 How to Apply Distributed Processing to a Client Application

A client application will instantiate a DispatcherUser that connects to the Dispatcher for sending jobs and receiving the results.

For applying distributed processing to a risk analysis, see Section 15.5.3, "How to Apply Distributed Processing to a Risk Analysis," on page 155.

Overview of Steps

- Step 1 — Create a DispatcherJob
- Step 2 — Create a DispatcherJobOutput
- Step 3 — Implement DispatcherUserListener and instantiate DispatcherUser

Step 1 — Creating a DispatcherJob

A DispatcherJob is an individual job sent to the Dispatcher for calculation.

Create a class named **apps.distproc.<name>Job** or **tk.distproc.<name>Job** that extends **com.calypso.tk.distproc.DispatcherJob**.

Implement the *process()* method. The *process()* method will return its results as a DispatcherJobOutput.

Sample Code in `calypsox/tk/distproc/`

`DispatcherJobPricing.java`

Step 2 — Creating a DispatcherJobOutput

DispatcherJobOutput contains the results of DispatcherJob.

Create a class named **`apps.distproc.<name>JobOutput`** or **`tk.distproc.<name>JobOutput`** that extends **`com.calypso.tk.distproc.DispatcherJobOutput`**.

Implement *get()* and *set()* methods in your DispatcherJobOutput class that allow the calculation routine in your DispatcherJob to set the calculated result values. Usually, these result values are PricerMeasures.

This class will be invoked from your DispatcherJob.

Sample Code in `calypsox/tk/distproc/`

`DispatcherJobOutputPricing.java`

Step 3 — Implementing DispatcherUserListener and Instantiating DispatcherUser

Your client application must implement **`com.calypso.tk.distproc.DispatcherUserListener`**.

Any client of the distributed processing system must be able to:

- Receive a large task from the user and divide that task into smaller jobs
- Send the jobs (using a DispatcherUser)
- Receive and relay the results
- Assess completion of the jobs
- Handle errors

Receiving and Dividing the Task

For example, you can use a Trade Filter to collect the trades that you want to process and create a DispatcherJob for every ten trades in the Trade Filter.

Also, a table of ratio by product type allows splitting the jobs through the classes **`tk.distproc.SwaptionProductRatio`** and **`tk.distproc.CancellableSwapRatio`**.

Currently, for a European swaption, the ratio is set to 5, and for a Bermudan swaption, the ratio is set to 10. Those settings can be modified in the above-mentioned classes.

Suppose the number of trades per job is set to 10:

- Each swap is counted for 1 trade. For 50 swaps in a portfolio, it will take 5 jobs to complete the task.
- Each European swaption is counted for 5 trades. For 50 swaptions in a portfolio, it will take 25 jobs to complete the task.
- Each Bermudan swaption is counted for 10 trades. For 50 swaptions in a portfolio, it will take 50 jobs to complete the task.

Sending Jobs

Your client application will create a DispatcherUser to send jobs to the Dispatcher. Your application will need a DispatcherConfig in order to specify the location of the running Dispatcher process, so that it can create the DispatcherUser as a client of the Dispatcher. In the line below, c is our DispatcherConfig object:

DispatcherUser d= new DispatcherUser(c);

The DispatcherUser can exist for the life of the client application, sending many DispatcherJobs via its *send()* method:

DispatcherUser.send(DispatcherJob);

When sending jobs, you can set the COMPRESS_DISPATCHER_JOBS environment property to true to compress the jobs, or to false otherwise. The default value is true. Note that on large jobs over a fast network, it may be faster to send the jobs uncompressed.

Receiving Results

The method for receiving results is established in the DispatcherUserListener interface:

public void jobFinished(DispatcherJobOutput out);

The *jobFinished()* method will receive and relay the results of the calculation when a job is finished. Inside *jobFinished()*, you should implement the work necessary to relay the results using a DispatcherJobOutput.

Assessing Completion

Most client applications will use a counter to count results as they arrive. Once the number of results equals the number of jobs sent, the client application can exit or proceed to other work.

Handling Errors

The method for handling errors is established in the DispatcherUserListener interface:

```
public void onDisconnect();
```

The *onDisconnect()* method is called if your connection to the Dispatcher is dropped. In this method you should implement error reporting, informing the user that the calculation has failed due to the lost connection.

Sample Code in calypsox/tk/distproc/

BatchPricing.java

15.5.2 How to Create a Custom Ratio Dispatcher by Product

Create a class named **tk.distproc.<product_type>ProductRatio** that implements the interface **tk.distproc.ProductRatio**.

This class will be invoked from **calypso.tk.distproc.ProductRatioUtil**.

Sample Code in calypsox/tk/distproc/

SwapProductRatio.java

15.5.3 How to Apply Distributed Processing to a Risk Analysis

Create a class named **tk.distproc.<analysis_name>AnalysisDispatcher** which implements **com.calypso.tk.distproc.AnalysisDispatcher**.

In order to detect errors returned by individual jobs that will make any further calculation meaningless, and to terminate the entire dispatch job, call the *DistAnalysis.stopAll()* method from *AnalysisDispatcher.taskFinished()*.

This class will be invoked from **com.calypso.tk.distproc.AnalysisDispatcher**.

Sample Code in com/calypso/tk/distproc/

- MTMAnalysisDispatcher — implements the split and aggregate methods which determine how the job is divided into tasks and how the result is aggregated.
- MTMJob — extends DispatcherJob, runs the MTM analysis, and returns the output.
- MTMJobOutput — extends DispatcherJobOutput, and returns the results from the MTMJob.


MTMAnalysisDispatcher.java


MTMJob.java


MTMJobOutput.java

15.5.4 How to Create a Custom Error Notification

The default error notification is an email when the dispatcher or the calculator are disconnected from the system. Email server host and port come from `calpyso_mail.properties`. This means that such a file should exist in a directory that is in the classpath on all hosts involved: client host, dispatcher host, and all calculator hosts. The DispatcherConfig provides the "from" and "to" email addresses.

To use this email-on-errors feature, the Calculators must know what DispatcherConfig to use. That means that either:

- The Calculators are started with "-config <DispatcherConfig_name>" at the command line, or
- The Calculators are started in GUI mode, and the DispatcherConfig is chosen from the drop-down list.

To create a custom error notification, create a class named **tk.distproc.CustomErrorNotifier** that implements **com.calypso.tk.distproc.ErrorNotifier**.

This class will be invoked from **com.calypso.apps.distproc.DispatcherConfigWindow**, **com.calypso.tk.distproc.Calculator**, and **com.calypso.tk.distproc.Dispatcher**.

16 Reference Data

16.1 Legal Entities

16.1.1 How to Create Custom Attributes on Legal Entities

You can build a custom input window. Your custom window can contain the fields you need in order to record the additional attributes. Users can then launch the window by clicking the **custom** button in the legal entity window

Create a class named **apps.refdata.LegalEntityCustomInputWindow** which implements the interface **com.calypso.apps.refdata.LegalEntityCustomInput**.

You must define the following methods in your LegalEntityCustomInputWindow class:

- *input()* — applies the display for your window.
- *allowAttributeWindow()* — specifies if you still allow Calypso's Legal Entity Attribute window to be launched.

This class will be invoked from **com.calypso.apps.refdata.BOLegalEntityWindow**.

Sample Code in calypsox/apps/refdata/

- SampleLegalEntityCustomInputWindow.java

16.1.2 How to Apply Custom Validation to a LegalEntity

Create a class named **apps.refdata.CustomLegalEntityValidator** which implements the interface **com.calypso.apps.refdata.LegalEntityValidator**.

This class will be invoked from **com.calypso.apps.refdata.BOLegalEntityWindow**.

Sample Code in calypsox/apps/refdata/

- SampleCustomLegalEntityValidator.java
- CustomLegalEntityValidatorWithLEAttributeCodeValidation.java

  CustomLegalEntityValidatorWithLEAttributeCodeValidation leaves a LegalEntity in the Disabled status as long as its attributes do not fit the requirements defined in the LegalEntityAttributeCode window.

16.1.3 How to Apply Custom Validation to a Legal Agreement

Create a class named **apps.refdata.CustomLegalAgreementValidator** that implements **com.calypso.apps.refdata.LegalAgreementValidator**.

This class will be invoked from
com.calypso.apps.refdata.BOLegalAgreementWindow.

Sample Code in calypsox/apps/refdata/

CustomLegalAgreementValidator.java

16.1.4 How to Apply Custom Validation to a LegalEntity Contact

Create a class named **apps.refdata.CustomLEContactValidator** which implements the interface
com.calypso.apps.refdata.LEContactValidator.

This class will be invoked from
com.calypso.apps.refdata.BOLEContactWindow.

Sample Code in calypsox/apps/refdata/

SampleCustomLEContactValidator.java

Shows the validation of the Swift code.

16.1.5 How to Apply Custom Validation to LegalEntity Attributes

Create a class named **apps.refdata.CustomLEAttributeValidator** which implements the interface
com.calypso.apps.refdata.LEAttributeValidator.

This class will be invoked from
com.calypso.apps.refdata.BOLegalEntityAttributeWindow.

Sample Code in calypsox/apps/refdata/

SampleCustomLEAttributeValidator.java

16.1.6 How to Apply Custom Validation to LegalEntity Registrations

Create a class named **apps.refdata.CustomRegistrValidator** which implements the interface
com.calypso.apps.refdata.RegistrValidator.

This class will be invoked from
com.calypso.apps.refdata.BOLERegistrationWindow.

16.2 Applying Custom Validation to a Margin Call Config

Create a class named
apps.refdata.CustomMarginCallConfigValidator which implements the interface
com.calypso.apps.refdata.MarginCallConfigValidator.

This class will be invoked from
com.calypso.apps.refdata.BOMarginCallConfigWindow.

Sample Code in calypsox/apps/refdata/

SampleCustomMarginCallConfigValidator.java

16.3 Settlement and Delivery Instructions (SDI)

16.3.1 How to Create a Custom SDI Selector

Refer to the *Calypso Settlements User Guide* for the methodology used by Calypso for selecting settlement and delivery instructions for trades.

Create a class named **`tk.bo.<product_type>SDISelector`** which implements the interface **`com.calypso.tk.bo.SDISelector`**.

This class will be invoked from **`com.calypso.tk.bo.SDISelectorUtil`**.

SDISelector contains the following methods:

- *validSDIList()* — returns a list of Valid SDI List.
- *validate()* — checks whether Static Data is still valid.
- *checkSettleDate()* — rejects any SDI record that uses an agent or intermediary that is not open for business on the transfer settlement date.
- *getSettlementMethod()* — returns the Settlement Method applicable for the Trade. The purpose is to restrict the search of Settlement Instruction based on a specific information set on the Trade. If the settlement Method returns null, the standard SDI search logic is applied.
- *getSettlementMethods()* — checks multiple transfer rules for ensuring that both pay and receive legs are using the same settlement method.
- *setTradeTransferRule()* — overrides, if necessary TradeTransferRule created by each BOProductHandler.
- *getValidManualSDIList()* — returns the first valid manual SDI or null if none found.
- *isBridgePossible()* — checks whether two Settle and Delivery Instructions are compatible. Compares the Settlement Methods of the Processing Organization and the Counterparty and returns whether Settlement Instructions are compatible.
- *matchSDIList()* — compares the two lists of Settle and Delivery Instructions and removes the incompatible SDI from the List. Both lists must be sorted by preference.
- *matchManualSDIList()* — same as above for Manual SDI.

Sample Code in `calypsox/tk/bo/`

`FXSDISelector.java`

16.3.2 How to Create a Custom SDI Sort Order

Create a class named **`tk.refdata.CustomComparatorSDI`** which implements the interface **`java.util.Comparator`**.

This class will be invoked from **`com.calypso.tk.util.ComparatorFactory`**.

Sample Code in calypsox/tk/refdata/

CustomComparatorSDI.java

16.3.3 How to Create a Custom SDI Description

Since users will be using the description to choose the proper instruction for making manual assignments of settlement and delivery instructions, you may wish to use custom descriptions.

Create a class named **tk.refdata.CustomSDIDescription** which implements the interface **com.calypso.tk.refdata.SDIDescription**.

This class will be invoked from **com.calypso.tk.refdata.SettleDeliveryInstruction** and **com.calypso.apps.refdata.BOSettlDeliveryWindow**.

Sample Code in calypsox/tk/refdata/

CustomSDIDescription.java

16.3.4 How to Apply a Custom Validation to an SDI

Create class named **apps.refdata.CustomSDIValidator** which implements the interface **com.calypso.apps.refdata.SDIValidator**.

This class will be invoked from **com.calypso.apps.refdata.BOSettlDeliveryWindow**.

Sample Code in calypsox/apps/refdata/

SampleCustomSDIValidator.java

16.3.5 How to add a Custom Menu Item to the SDI Window

Create a class named **apps.refdata.CustomSDIMenu** which implements the interface **com.calypso.apps.refdata.SDIMenu**.

This class will be invoked from **com.calypso.apps.refdata.BOSettlDeliveryWindow** for any utility function

Sample Code in calypsox/apps/refdata/

SDIMenu.java

16.3.6 How to Create a Custom Summary Panel on the SDI Window

Create a class named **apps.refdata.CustomSDISummaryPanel** which implements the interface **com.calypso.apps.refdata.SDISummaryPanel**.

This class will be invoked from **com.calypso.apps.refdata.BOSettlDeliveryWindow**.

Sample Code in calypsox/apps/refdata/

CustomSDISummaryPanel.java

16.3.7 How to Apply Custom Validation to an SDI Relationship

Create a class named **apps.refdata.CustomSDIRelationShipValidator** which implements the interface **com.calypso.apps.refdata.SDIRelationShipValidator**.

This class will be invoked from **com.calypso.apps.refdata.SDIRelationShipWindow**.

Sample Code in calypsox/apps/refdata/

SampleCustomSDIRelationShipValidator.java

16.3.8 How to Apply Custom Validation to a Manual SDI

Create a class named **apps.refdata.ManualSDIValidator** which implements the interface **com.calypso.apps.refdataCustomManual.SDIValidator**.

This class will be invoked from **com.calypso.apps.refdata.ManualSDIWindow**.

Sample Code in calypsox/apps/refdata/

SampleCustomManualSDIValidator.java

16.4 How to Apply Custom Validation to a Book

Create a class named **apps.refdata.CustomBookValidator** which implements the interface **com.calypso.apps.refdata.BookValidator**.

This class will be invoked from **com.calypso.apps.refdata.BookWindow**.

Sample Code in calypsox/apps/refdata/

CustomBookValidator.java

16.5 Static Data Filter

16.5.1 Creating a Custom StaticDataFilter Attribute

Do the following for create a custom StaticDataFilter attribute:

1. Create a class named **tk.refdata.CustomStaticDataFilter** which implements the interface **com.calypso.tk.refdata.StaticDataFilterInterface**.

   This class will be invoked from **com.calypso.tk.refdata.StaticDataFilterElement** and from **com.calypso.tk.refdata.StaticDataMaintenanceElement**.

2. Create a custom panel for the new attribute as described in the following section.

Sample Code in calypsox/tk/refdata/

CustomStaticDataFilter.java

16.5.2 Creating a Custom Attribute Panel

Create a class named **apps.refdata.<attribute_name>CustomAttributePanel** that implements the interface **com.calypso.apps.refdata.CustomAttributePanel**.

This class will be invoked from **com.calypso.apps.refdata.StaticDataFilterWindow**.

Sample Code in calypsox/apps/refdata/

IS_NULLCustomAttributePanel.java

16.5.3 Applying Custom Validation to a Static Data Filter

To perform custom validation on a Static Data Filter, create a class named **tk.refdata.CustomStaticDataFilterValidator** that implements **com.calypso.tk.refdata.StaticDataFilterValidator**.

Sample Code in calypsox/tk/refdata/

CustomStaticDataFilterValidator.java.

16.6 Trade Filter

16.6.1 Position Based Products

Added domain PositionBasedProducts - List of products that return true in their implementation of *isPositionBased()*. This list is used internally for excluding the trades whose products are position based from trade filters with that criteria.

This list is not to be modified, and should include at most all products that are position based products. Including products which return false from their *isPositionBased()* implementations will result in incorrect behavior when loaded through trade filters with the property setIncludePositionBased to false.

Not including all position based products in this list will only result in lower performance and higher memory requirements when loading trade filters with the property setIncludePositionBased to false.

16.6.2 Creating a Custom Trade Filter Attribute

Follow the steps below to create a custom Trade Filter attribute:

1. Create a class named **tk.mo.CustomCriterion<name>** that implements **com.calypso.tk.mo.CustomCriterion**.

   When implementing a custom attribute, you can decide whether to generate the SQL clause or not. The SQL clause will be appended to the Trade Filter SQL statement for loading trades. Generating the

SQL clause allows loading trades more efficiently. However, in cases where generating the SQL clause is not feasible, you can let the *accept()* method in TradeFilter doing the filtering on the loaded trades.

This class will be invoked from **com.calypso.tk.mo.CustomCriterion**.

2. Register the new attribute in the **customCriterion** domain.
3. Create a custom panel for the new attribute as described in the following section.

Sample Code in calypsox/tk/mo/

- CustomCriterionIssuer.java
- CustomCriterionFeeType.java.

Issuer is implemented without generating the SQL clause, and FeeType generates the SQL clause.

16.6.3 Creating a Custom Trade Filter Validator

To validate the contents of a Trade Filter prior to saving it, create a class named **apps.refdata.CustomTradeFilterValidator** that implements the interface **com.calypso.apps.refdata.CustomTradeFilterValidator**.

CustomTradeFilterValidator is invoked from **com.calypso.apps.refdata.TradeFilterWindow**.

Method **isValidInput**

isValidInput in CustomTradeFilterValidator performs validation of Trade Filter fields.

Parameters **TradeFilter tradefilter —** The name of the Trade Filter.

Frame w — The Trade Filter window handle to use for editing.

Vector messages — A Vector of string messages that the method can attach messages for the user.

Returns **False —** Saves not permitted.

True — Saves premitted

Sample Code in calypsox/apps/refdata/

- CustomTradeFilterValidator.java

16.6.4 Creating a Custom Attribute Panel

Create a class named **apps.refdata.<*attribute_name*>Panel** that implements the interface **com.calypso.apps.refdata.CustomCriterionPanelInterface**.

<*attribute_name*>Panel is invoked from **com.calypso.apps.refdata.TradeFilterWindow**.

16.7 Applying Custom Validation to a Fee Grid

Create a class named **apps.refdata.CustomFeeGridValidator** that implements **com.calypso.apps.refdata.FeeGridValidator**.

CustomFeeGridValidator is invoked from **com.calypso.apps.refdata.FeeGridWindow**.

16.8 CFD

16.8.1 How to Apply Custom Validation to a CFDContractDefinition

Create a class named **apps.refdata.CustomCFDContractValidator** which implements the interface **com.calypso.apps.refdata.CFDContractValidator**.

This class will be invoked from **com.calypso.apps.refdata.CFDContractWindow**.

Sample Code in calypsox/apps/refdata/

SampleCustomCFDContractValidator.java

16.8.2 How to Apply Custom Validation to a CFDCountryGrid

Create a class named **apps.refdata.CustomCFDCountryGridValidator** which implements the interface **com.calypso.apps.refdata.CFDCountryGridValidator**.

This class will be invoked from **com.calypso.apps.refdata.CFDCountryGridWindow**.

Sample Code in calypsox/apps/refdata/

SampleCustomCFDCountryGridValidator.java

16.9 Audit and Authorization

16.9.1 How to make a Class Auditable and Authorizable

Auditable means that all changes to an object such as INSERT, AMEND and REMOVE are recorded with the following information:

- What has changed in the object
- When was the object changed
- Who made the changes

Authorizable means that when an object is changed, another user has to authorize the changes.

The activation of Audit and/or Authorization is done through the Admin Window. Refer to the *Calypso System Guide* for details.

Overview of Steps

- Step 1 — Create a class that implements Auditable or Authorizable (which in turn extends Auditable)
- Step 2 — Make the class persistent
- Step 3 —Register the class for audit and authorization

Step 1 — Create a Class that Implements Auditable or Authorizable

Create a class that implements **`com.calypso.tk.core.Auditable`** for audit only, or **`com.calypso.tk.core.Authorizable`** for both audit and authorization.

Audit

Implement the following methods on Auditable:

- doAudit()
- undo()
- clone()
- getVersion()
- setVersion()

You must also implement the following methods:

- getUser()
- setUser()

The following variables must be defined:

- int version
- String user

Authorization

Implement the following methods on Authorizable:

- getId()
- setId()
- diff()
- apply()
- getAuthName()

The following variable must be defined:

- int id

Sample Code

`samples.cookbook.tk.refdata.LegalEntityLimit.java`

Step 2 — Make the Class Persistent

To make the class persistent, create a class that extends **`com.calypso.tk.core.sql.AuditSQL`** for audit only, or **`com.calypso.tk.core.sql.AuthorizableSQL`** for both audit and authorization.

The following methods must be overloaded in your SQL class:

- save()

- remove()
- find()

The database table does not need any field to store audit information. This is all taken care of by Calypso using **bo_audit**. However the table needs the following:

- A unique identifier, preferably an integer
- A version number (Calypso will control the versioning for you)

Sample Code

samples.cookbook.tk.refdata.sql.LegalEntityLimitSQL.java

Database scripts:

samples/cookbook/le_limit.sql (Sybase)

samples/cookbook/le_limit_Oracle.sql (Oracle)

Step 3 — Registering the Class for Audit and Authorization

The class name should be registered in the classAuditMode domain for audit, and in the classAuthMode domain for authorization.

Sample Code in samples/cookbook/

The LegalEntityLimit sample can be activated with the following additional files:

apps/refdata/LegalEntityLimitWindow.java

apps/refdata/CustomDataServer.java

apps/refdata/RemoteCustomData.java

le_limit.sql (for Sybase)

le_limit_Oracle.sql (for Oracle)

To run the sample, you must run the following at the command line to start the custom server:

```
rmic samples.cookbook.tk.service.CustomDataServer
```

When the Audit and Authorization modes are on, you will be able to save a LegalEntityLimit in the LegalEntityLimitWindow, and you will see that the values can be authorized using **Main Entry > Processing > Data Authorization**, and audited using **Main Entry > Reports > Audit > Audit Report**.

16.9.2 How to Create a Custom Authorization Window

The default Authorization Window is **com.calypso.apps.refdata.AuthorizationWindow**.

Create a class named **apps.refdata.<name>AuthViewer** that implements **com.calypso.apps.refdata.AuthViewer**.

This class will be invoked from **com.calypso.apps.refdata.AuthViewerUtil**.

16.9.3 How to add Custom Authorization to a Class

Create a class named **apps.util.<class_name>CheckAuthorization** that implements **com.calypso.apps.util.CheckAuthorization**.

This class will be invoked from **com.calypso.tk.refdata.AccessUtil**.

16.9.4 How to Create Custom Authorization Behavior

You can implement a custom authorization behavior that will be invoked when clicking the Accept button in the Authorization window. Create a class named **tk.refdata.CustomPreAuthorize** that implements **CustomPreAuthorizeInterface**.

16.10 Authentication Service

16.10.1 Overview

The Calypso Authentication Service (AS) is a stand-alone process responsible for the management of user identities and the authentication of all Calypso clients. It exposes a series of RMI services used by all clients for authentication. All clients must first login to the AS in order to receive a token that may then be used for subsequent requests to services and messages.

This process is managed entirely for all clients via the standard **ConnectionUtil.connect()** method.

Internally, the authentication service leverages ACEGI Security (Spring Security) framework: http://www.acegisecurity.org

16.10.2 Extensions

The AS contains multiple extension points which may be used to connect Calypso to an external identity management system such as LDAP, Active Directory, etc. These extension points allow users to login to Calypso using credentials managed in an external or centralized system.

Calypso does not currently support the additional ability to implement a single sign on, which implies removing the need for users to login to Calypso.

The extension points within the AS are defined below:

UserDetailService Extension

The default configured AuthenticationProvider is ACEGI's DAOAuthenticationProvider type, which gives an extension point for loading user details based on username.

If the requirement is to load user information from other (i.e., non-Calypso database) userstores (Stores that contains user information) such as LDAP or a client's custom database, then you must replace the default UserDetailsService with a custom User Details Service and corresponding PasswordEncoder.

The PasswordEncoder will compare the incoming (user provided) unencrypted password with the encrypted password from the userstore, while the custom implementation of the User Details Service must return **userdetails** of type **com.calypso.infra.authentication.providers.CalypsoUserDetails**.

When the user details service is replaced, the PasswordEncoder must also be replaced because the encrypted password in the custom userstore is could potentially be encrypted using a different encryption algorithm.

A DAO named UserSQL is available for any implementation of the UserDetailsService to provide it with access to the Calypso User Repository. Scenarios in which user attributes, such as locked state, are not managed by the external identity store may require the use of the UserDetailsService. This reference can be passed to the UserDetailsService using the same pattern as is used by the CalypsoUserDetails service in the configuration section (outlined below).

AuthenticationProvider

This component is the lowest level responsible for the entire authentication process. The default implementation of this component is the:

com.calypso.infra.authentication.providers.CalypsoAuthenticationProvider

CalypsoAuthenticationProvider extends the ACEGI's DaoAuthenticationProvider. This default implementation leverages the UserDetailsService and the PasswordEncoder to process the authentication of the user. In most cases, connectivity to eternal identity systems only require extension of the UserDetailsService and the PasswordEncoder.

In some cases, however, clients may wish to completely control the authentication process and may override the AuthenticationProvider. New AuthenticationProviders must return extensions of:

com.calypso.infra.authentication.providers.CalypsoAuthenticationToken

This token must contain user details that extend:

com.calypso.infra.authentication.providers.CalypsoUserDetails

16.10.3 Configuration

The AS components are defined in the following Spring configuration file:

`$CALYPSO_HOME/resources/appconfig/AuthenticationServer.xml`

Implementations for the various components defined above are changed here.

The Authentication provider is defined in the `<bean id="authenticationManager" />` node. New authentication providers can be configured to replace the default Calypso Authentication Provider or can be added to the existing list.

The default Authentication Provider is defined in the `<bean id="defaultAuthenticationProvider"/>` node. This node is

where you would change the UserDetailsService and PaswordEncoder beans to custom implementations.

16.10.4 Example Custom UserDetailsService Extension

Changes to AuthenticationService.xml

Replace the beans passwordEncoder and userDetailsService with the customs versions as shown below.

```
<bean id="passwordEncoder"
     class="com.calypso.authentication.providers.encoding.DummyEncoder" />
<bean id="userDetailsService"
     class="com.calypso.authentication.userdetails.DummyUserDetailsService"
     init-method="init"/>
```

PasswordEncoder

This custom implementation does not encode the password data stored in the userstore nor is it encrypted.

```
public class DummyEncoder implements PasswordEncoder{

  @Override
  public String encodePassword(String arg0, Object arg1)
     throws DataAccessException {
    // TODO Auto-generated method stub
    return null;
  }

  @Override
  public boolean isPasswordValid(String enc, String plain, Object arg2)
     throws DataAccessException {
    return (enc.equals(plain));
  }


}
```

UserDetailsService

This custom User Details Service holds on the user/data in memory and the password as an unencrypted Map<user, password> password:

```
public class DummyUserDetailsService implements UserDetailsService{

  private Map<String, String> users = new HashMap<String, String>();

  public void init(){
    users.put("calypso_user", "test");
    users.put("admin", "test1");
  }

  @Override
  public UserDetails loadUserByUsername(String username)
     throws UsernameNotFoundException, DataAccessException {
```

```
    CalypsoUserDetails userDetails = new CalypsoUserDetails(username,
        users.get(username), true,
        true, true, true,
        new GrantedAuthority[] {
        new GrantedAuthorityImpl("ROLE_ADMIN"),

    });
    userDetails.setFullName("FullName");
    userDetails.setAdmin(true);
    return userDetails;
  }
}
```

16.10.5 Authentication Service Setup

1. AUTHSERVERHOST
2. AUTHSERVER_RMI_REGISTRY_PORT
3. AUTH_SERVICE_PORT_RANGE_START
4. SESSIONTIMEOUT_INMINUTES (for the AuthService and clients of AuthService)
5. MIN_TOKEN_VERIFY_ATTEMPTS (Clients of AuthService)
6. FORBID_MULTIPLE_LOGIN (Application names that should be forbidden from multiple login attempts. The string is a comma separated list of Application Names, for example, **MainEntry,Admin**).

Authentication is based on ACEGI Security and can be configured by updating the `AuthenticationServer.xml` spring config file to replace the **userDetailsService** beans and the **passwordEncoder** bean with custom implementations to load and authenticate the user against a third-party CredentialsStore such as LDAP or other client Database.

UserDetailsService is an implementation of the interface **`org.acegisecurity.userdetails.UserDetailsService`** that has one method, **`UserDetails loadUserByUsername(String username)`**, this method should be implemented to load the user information for the username.

In addition to the ACEGI security, Calypso expects the returned UserDetails CalypsoUserDetails object.

```
CalypsoUserDetails userDetails = new CalypsoUserDetails(username,
    new String(Base64.encodeBase64(calypsoUser.getCryptedPassword())), true,
    true, true, accountNonLocked,
    new GrantedAuthority[] { new GrantedAuthorityImpl("ROLE_ADMIN"),});
userDetails.setFullName(calypsoUser.getFullName());
userDetails.setAdmin(this.userSQL.isAdmin(calypsoUser));
```

Calypso does not use role-based Authorization. The only role that is allowed in the ROLE_ADMIN is hard coded in the above code.

Details of whether the user is an admin should be fetched from the calypso DB as this information is *not* available in the external CredentialsStore.

PasswordEncoder is the implementation of the interface org.acegisecurity.providers.encoding.PasswordEncoder. Currently, Calypso only needs the the implemented method:

```
public boolean isPasswordValid(String encPass, String rawPass, Object salt)
```

Where *encPass* is the encrypted password from the CredentialsStore, *rawPass* is the user presented password, and salt is the encryption salt.

verifyToken

A call to verify token extends the validity of the AuthenticationToken by SESSIONTIMEOUT_INMINUTES minutes. Client applications call the SESSIONTIMEOUT_INMINUTES method "MIN_TOKEN_VERIFY_ATTEMPTS" times within a SESSIONTIMEOUT_INMINUTES duration.

e.g. if SESSIONTIMEOUT_INMINUTES=30 and MIN_TOKEN_VERIFY_ATTEMPTS=4, a verifyToken method call is made every 7.5 minutes (30/4 = 7.5 minutes).

The Authentication Service stores Tokens in the DB. The tokens are deleted when they expire or if the client make an explicit logout call.

Applications that are logged in are considered logged in for SESSIONTIMEOUT_INMINUTES duration, even in the event of a Authentication Service crash.

Applications are typically stopped using the stopAll feature in Admin, which stop all applications except the Event Server, which functions as a JMS messaging bus (Calypso ships with activemq components/services).

In the event of system wide crash, some users may remain logged in for SESSIONTIMEOUT_INMINUTES minutes, which may prevent certain engines from starting because only single instance of the engine is permitted in the system.

In the event of a system crash (not an orderly restart) Calypso advises that user restart the Authentication Service with the **-clean** argument to force a clean instance of Authentication Service to launch which will purge all tokens from the tokenstore table. After a clean start of the Authentication Service any users who remained logged in must be restarted as well..

```
java com.calypso.apps.statup.StartAuthService -user user -password passwd
         -env envname -clean
```

getConnectedClients returns the list of currently logged in clients based on the session timeout and the clients connection status with the event server.

The granularity of the SESSIONTIMEOUT_INMINUTES is in minutes, which is large enough to ignore transient network issues that prevents the clients from being able to renew the token. The secondary requirement is to quickly determine if client is still connected. To obtain feedback more rapidly, the connection with the Event Server is used to provide client connection status.

In effect, **getConnectedClient** returns the list of currently logged in clients (tokens are still valid in the DB) that are connected to the Event Server.

Applications that do not need a connection to the Event Server rely on the granularity of this method based on the SESSIONTIMEOUT_INMINUTES setting.

Configuring the Event Server

The System Administrator must allocate one port to allow the Event Server to operate through a firewall. The default Event Server port is 2099. To change the port number, edit the Calypso environment file and change the value of the Port field in the Event Server section.

The Event server must be started as an independent process to use a defined port number.

16.11 Access Permissions

16.11.1 How to add Custom Access Permission Functions

To add a new function, add the function name to the function domain. If it is a restriction, add it to restriction domain instead.

For example, we add MyCheck to the function domain.

Then in your custom code, you can check the function using:

```
If AccessUtil.isAuthorized('MyCheck')
```

16.11.2 How add Custom Access Permissions to a Class

Create a class named **apps.util.<class_name>CheckAccess** which implements the interface **com.calypso.tk.refdata.CheckAccess**.

Sample Code in calypsox/apps/util/

- SampleLEContactCheckAccess.java
- BondAssetBackedCheckAccess.java
- SampleSettleDeliveryInstructionCheckAccess.java

16.11.3 How to Create Custom Trade Access Permissions

Create a class named **tk.refdata.CustomTradeAccess** that implements the interface **com.calypso.tk.refdata.TradeAccess**.

16.11.4 How to Create a Custom User Setup

To control which GUI and config properties for a given "reference user" should be duplicated when adding a new user or changing a user's group in the User Access Permission window.

Create a class named **apps.refdata.UserSetup** which implements the interface **com.calypso.apps.refdata.CustomUserSetup**.

Sample Code in calypsox/apps/refdata/

- UserSetup.java

16.11.5 How to Apply Custom Validation to User Access Permissions

Create a class named **tk.refdata.DefaultCustomProfileValidator** which implements the interface **com.calypso.tk.refdata.CustomProfileValidator**.

Sample Code in calypsox/tk/refdata/

- DefaultCustomProfileValidator.java

16.12 Scheduled Tasks

Scheduled Tasks can be used to run tasks on a regular basis, such as exporting data and and importing data on a regular basis.

Out-of-the-box, Calypso provides a number of Scheduled Tasks described in the *Calypso Trade Lifecycle User Guide*.

16.12.1 How to Create a Custom Scheduled Task

Create a class named **tk.util.ScheduledTask<name>** which extends the class **com.calypso.tk.util.ScheduledTask**.

Note that a scheduled task cannot be executed inside the Data Server.

Note: It is not recommended to audit every data member encapsulated within a custom ScheduledTask implementation. You should consider the following alternatives:

- If you are extending ScheduledTask, then override +doAudit:void. In this operation you can pick and choose the data members you wish audited.
- If you want simply exclude a data member from the audit process, rename the data member by pre-appending a double underscore to its moniker (for example, change class variable foo to __foo).

This class will be invoked from **com.calypso.tk.util.ScheduledTask**.

Sample Code in calypsox/tk/util/

- ScheduledTaskDEAL_REPORT.java
- ScheduledTaskFXNDF_CHECK.java
- ScheduledTaskTASK_STAT_REPORT.java
- ScheduledTaskWAIT_STOP_ENGINE.java.

16.12.2 How to Customize Scheduled Task MESSAGE_MATCHING

The Scheduled Task MESSAGE_MATCHING can be used for matching external SWIFT messages. It can be customized in the following manner.

- **com.calypso.tk.util.SwiftMessageInput** — Write a class called CustomSwiftMessageInput which implements SwiftMessageInput.
- If you do not write CustomSwiftMessageInput the scheduled task reads the text file Incomingswift.txt where the messages are separated by the separator specified in the scheduled task attributes.

- **com.calypso.tk.util.swiftparser.TagParser** — For parsing YYY tag you must write TagYYParser which implements TagParser.
- **com.calypso.tk.util.swiftparser.MessageMatcher** — For matching the "MT000" type of message you will need MessageMT000Matcher which implements MessageMatcher.
- **com.calypso.tk.util.swiftparser.MessageProcessor** — For processing the message MT000 (matched/unmatched) you must write MT000MessageProcessor which implements MessageProcessor. This class gets the BOMessage created using swift, and if matched then a BOMessage which is matched. Here you can do the final processing for that message.

16.12.3 Customizing INVENTORY_SNAPSHOT

The INVENTORY_SNAPSHOT scheduled task has been modified to allow customization of the snapshot name and to allow generating a snapshot by currency.

The constraint on the name of the customized snapshot is that the 4th character must be an underscore (_). The following methods are now protected:

- **getChangedPositionClass()** — This is the method that generates the name of the snapshot.
- **getDateFromSnapshot()** — This method decodes the date of the snapshot from it's name.
- **getLatestSnapshot()** — This method returns the latest snapshot name which needs to be retained for the purge. It is called from purgeSnapshot() method. It has to be done for each position class (Internal, client, External).

The methods **getExtraCashSQL()** and **getExtraSecuritySQL()** have been added to allow adding an SQL where clause.

17 Workflow

17.1 Workflow Process

17.1.1 How to Create a Custom Exception Handler

Create a class named **tk.bo.workflow.exhandler.<exception_type>ExceptionHandler** which implements the interface **com.calypso.tk.bo.workflow.ExceptionHandler**.

This class will be invoked from **com.calypso.tk.bo.workflow.ExceptionHandlerUtil**.

Sample Code in calypsox/tk/bo/workflow/exhandler/

EX_MISSING_SIExceptionHandler.java

17.1.2 How to Create a Custom KickOffDate, CutOffDate

Create a class named **tk.bo.workflow.KickOffCalculator<config_name>** which implements the interface **com.calypso.tk.bo.workflow.KickOffCalculator**.

This class will be invoked from **com.calypso.tk.bo.workflow.KickOffCalculatorUtil** to override the KickOffDate and/or CutOffDate calculation for a particular KickOffCutOffConfig.

Note: For performance reasons, workflow rules are executed within the Data Server. Be very careful to clone any objects retrieved from the Data Server prior to modifying them.

Sample Code in calypsox/tk/bo/workflow/

KickOffCalculatorTest.java

17.1.3 How to Create Custom Data for a Task

Create a class named **tk.bo.CustomTaskInfo** which implements the interface **com.calypso.tk.bo.TaskFillInfo**.

This class will be invoked from **com.calypso.tk.bo.TaskFillInfoUtil**.

Sample Code in calypsox/tk/bo/

SampleCustomTaskInfo.java

17.1.4 How to Create Custom Rules, Actions, and Statuses

Do the following to create custom Rules, Actions, and Statuses for any workflow type:

1. Create a class named **tk.bo.workflow.rule.<component_name><workflow_type>Rule** that implements the interface **com.calypso.tk.bo.workflow.WfRule**.

   This class will be invoked from **com.calypso.tk.bo.workflow.WorkflowRuleUtil**.

2. Register the new workflow type in the workflowType domain.
3. Add the new statuses, actions, and rules to this workflow type using **Main Entry > Configuration > Workflow > Workflow Configuration > Domains > Entity** as applicable.

See also, Section 17.4.6, "How to add Custom Menu Items to the Task Station," on page 181 for adding custom actions.

17.2 How to Create a Custom Workflow Rule

To create a custom Trade Workflow Rule, create a class named **tk.bo.workflow.rule.<rule_name>TradeRule** which implements the interface **com.calypso.tk.bo.workflow.WfTradeRule**.

To create a custom Message Workflow Rule, create a class named **tk.bo.workflow.rule.<rule_name>MessageRule** which implements the interface **com.calypso.tk.bo.workflow.WfMessageRule**.

To create a custom Transfer Workflow Rule, create a class named **tk.bo.workflow.rule.<rule_name>TransferRule** which implements the interface **com.calypso.tk.bo.workflow.WfTransferRule**.

Note: The rule names must be registered with the appropriate workflow rule domains: workflowRuleMessage, workflowRuleTrade, and workflowRuleTransfer.

The following methods must be implemented on the interfaces WfTradeRule, WfTransferRule and WfMessageRule:

- *check()* — This method should only contain tests, and no object should be modified in this method. The reason is that if a given transition has more than one rule, the system will first call all the *check()* methods, and if all of them return true, it will call the *update()* methods, and then save any object as applicable.

  This method can be run on both client and data server sides. When applying a transition for saving/updating objects, the workflow will run on the server, but if a user wants to simulate a transition, the workflow runs on the client side.

  - When loading static data, you should use BOCache, LocalCache, and the remote services when possible. The workflow will know by itself on which side the code runs. The following code for example, can be run on both sides.

```
LegalEntity po = BOCache.getLegalEntity(dsCon, transfer.getProcessingOrg());
```

  When loading active data, you should first check if you run on the client or server side - to know that you have to test if the dbCon is null or not.

  - If it is not null, you run on the server side and you must use the SQL class. Otherwise, you must use the remote services. Note that a DSConnection is never null, even on the server side. Therefore, the code should be like:

```
if (dbCon != null) {
   trade = TradeSQL.getTrade(id, (Connection)dbCon);
} else {
   trade = dsCon.getRemoteTrade().getTrade(id);
}
```

- *getDescription()* — This method will be called from the Workflow Config window to display information about the rule.
- *update()* — This method will be called by the system when all rules return true from the *check()* methods. You can modify object in this

method. Note that this method will **always** be run on the server side. Therefore, **only** the dbCon can be used. For example, you can do:

```
TaskSQL.save(newTask, (Connection)dbCon);
```

Moreover, if you want to save and publish new events inside a workflow rule, that must be done in the *update()* method. You must create the event and add it to the events vector that is one of the arguments of the method. For example, you can do:

```
TaskSQL.save(newTask, (Connection)dbCon);
PSEventTask taskEvent = new PSEventTask();
taskEvent.setTask(newTask);
events.addElement(taskEvent);
```

If you want to create exception tasks, it is recommended to create BOException objects and add them to the excps vector that is one of the arguments of the method. For example, you can do:

```
BOException boExcp = new BOException(tradeId,
   this.getClass().getName(),
   "Exception Message",
   BOException.INFORMATION);
   excps.addElement(boExcp);
```

These classes will be invoked from **com.calypso.tk.bo.workflow.WorkflowRuleUtil**.

Sample Code

calypsox.tk.bo.workflow.rule.

17.3 How to Implement a Custom Workflow

Calypso offers the ability to implement a workflow for any entity. We will use the LegalEntity object to illustrate the implementation of a custom workflow.

An implementation using the Book class was also successfully implemented and tested. Note however that the current implementation stores the entity id as an integer. This raises some issues as to the feasibility to use classes which use a String identifier.

17.3.1 Entity

The object for which you want to implement a custom workflow must be identified as an Entity.

Implementing EntityObject

The object for which you want to implement a generic workflow must implement **com.calypso.tk.core.EntityObject**.

For example, the following code was added to the class **com.calypso.tk.core.LegalEntity** to become an EntityObject.

New imports are needed:

```
import java.sql.Connection;
import com.calypso.tk.core.sql.LegalEntitySQL;
```

The following methods provide a simple yet sufficient implementation of EntityObject interface:

```
/**
* New class field keeps a reference to EntityState
* @see com.calypso.tk.core.EntityState
*/
protected EntityState  _entityState = new EntityState();

/**
* Returns a unique id for this EntityObject. Note that together with
* the value returned by <code>getEntityType()</code>, the ID-type pair
* must uniquely identify this Entity Object in the system.<br>
* It is possible, however, for 2 or more EntityObjects to have the same
* id if they have different Entity Types.
* <p>
*/
public int  getEntityId()  { return getId(); }

/**
* Returns a type that uniquely identifies this EntityObject "type".
* Typically, the simplest way to implement this method is simply to
* return getClass().toString(). However, this is left as an implementation
* detail to permit more customization control.
*/
public String getEntityType()  { return "LegalEntity"; }

/**
* Returns an object that encapsulates the Workflow State for this
* <code>EntityObject</code>.
*
* @return the state associated to this entity object
* @see com.calypso.tk.core.EntityState
*/
public EntityState getEntityState() { return _entityState; }

/**
* Sets the object which encapsulates the Workflow State for this
* <code>EntityObject</code>
*
* @param state the workflow state to associate to this entity object.
* @see com.calypso.tk.core.EntityState
*/
public void setEntityState(EntityState state) { _entityState = state; }

/**
* Returns the Processing Org associated with this entity. Note
* that if not applicable, the method should return "ALL", preferably.
*
*/
public String getProcessingOrg()   { return “ALL”; }
```

It is also important to remember to update the **clone()**, **readExternal(...)**, and **writeExternal(...)** methods. It is straightforward but quite important to pass the information through RMI:

```
public void writeExternal(ObjectOutput out)
   throws IOException {
      ...
      boolean v=_entityState != null;
      out.writeBoolean(v);
      if(v) _entityState.writeExternal(out);
   }

public void readExternal(ObjectInput in)
   throws IOException,ClassNotFoundException {
      ...
      if(in.readBoolean()) {
         _entityState = new EntityState();
         try {_entityState.readExternal(in);}
         catch(Exception e) {
            Log.error(this, e);
               throw new IOException(e.getMessage());
            }
         }
   }
```

Lastly, you must adjust the **serialVersionUID** field.

Implementing EntityPersistence

When an object goes through the workflow, the actual saving to the database is done via the EntityObject interface. The object goes through the workflow and, if no error is raised, the object and its state are saved to persistent data by calling *EntityObjectSQL.save(EntityObject, Connection)*. To properly persist your object at this point, you must implement the appropriate persistence class. For example, if you have the following class **`calypsox.tk.mypackage.MyObject`** which implements EntityObject, then you must create **`calypsox.tk.mypackage.sql.MyObjectSQL`** that implements **`com.calypso.tk.core.sql.EntityPersistence`**. In doing so, you ensure that EntityObjectSQL is able to save, retrieve, remove your objects properly as the object goes through the workflow.

The changes to EntityObjectSQL are minimal. For this example, the changes are made to **`com.calypso.tk.core.sql.LegalEntitySQL`**. The idea is for the object's associated EntityState to be saved/removed/retrieved as needed. Hence, all those methods which retrieve the LegalEntity object from the database make a call as follows:

```
EntityObjectSQL.setEntityState(legalEntity, con);
```

Similarly, the following calls are used in the save and remove methods, respectively:

```
EntityObjectSQL.saveEntityState(legalEntity, con);
EntityObjectSQL.removeEntityState(legalEntity, con);
```

We have established an association between the LegalEntity, an entity object, and its associated EntityState, its state. To ensure data integrity we must be sure that the memory image for the object matches that in the database.

Modifying ReferenceDataServerImpl

You must change the API for saving the object.

For this example, we change the *save(LegalEntity)* method. Currently, the save operation is invoked as follows:

```
int lid = LegalEntitySQL.save(legalEntity);
```

By changing this to the following, everything is handled in the workflow, including the actual "save" operation:

```
saveEntityObject(legalEntity);
int lid = legalEntity.getId();
```

Adding Workflow Rules

On occasion you may need to add workflow rules for triggering the workflow.

In this example, we have created a simple class **`com.calypso.tk.bo.workflow.rule.CheckValidLegalEntityRule`** that checks various properties of a Legal Entity to determine whether or not it is valid.

The workflow can have the following status: NONE, PENDING, or VERIFIED. The Action NEW creates the transition from NONE to PENDING. The Action AMEND, with STP flag on, links PENDING to VERIFIED with a call to this rule. Lastly, there is a link back from VERIFIED to PENDING, also on AMEND, so that any changes to the LegalEntity are validated back through the rule.

17.3.2 Domain Data

The following domain values should be added:

1. Add the entity to the workflowType domain. In our example, we add LegalEntity.
2. Create the domain workflowRule<entity>. In our example, it is workflowRuleLegalEntity.
3. Add the workflow rules that you have created to the workflowRule<entity> domain. In our example, we add CheckValid to the workflowRuleLegalEntity domain.

17.3.3 Workflow

Once the domain values have been set, the workflow can be configured using Main Entry > Configuration > Workflow > Workflow Configuration.

Make sure to add the actions, rules, and status codes as applicable using the menu items under Domain > Entity. You will be prompted to enter the entity (LegalEntity in our example).

We provide a sample configuration of the LegalEntity workflow. You must start with a clean database and apply Demonstration Data in order to load the sample configuration.

17.4 Task Station

17.4.1 How to Create a Custom Action Task Handler

A custom action task handler allows adding custom processing (such as displaying a warning message, prompting the user to enter additional data, etc.), when a given action is applied.

Create a class named **tk.bo.workflow.TaskHandler<action>** that implements **com.calypso.tk.bo.workflow.TaskHandler**.

This class is invoked from **com.calypso.tk.bo.workflow.TaskHandlerUtil**.

17.4.2 How to Create a Custom Summary in the Trade Panel

Create a class named **apps.reporting CustomTradeSummaryPanel** which implements the interface **com.calypso.apps.reporting.TradeSummaryPanel**.

This class is invoked from **com.calypso.apps.reporting.TaskStationJFrame**.

Sample Code in calypsox/apps/reporting/

SampleCustomTradeSummaryPanel.java

17.4.3 How to Create a Custom Summary in the Message Panel

Create a class named **apps.reporting.CustomMessageSummaryPanel** which implements the interface **com.calypso.apps.reporting.MessageSummaryPanel**.

This class is invoked from **com.calypso.apps.reporting.TaskStationJFrame**.

Sample Code in calypsox/apps/reporting/

SampleCustomMessageSummaryPanel.java

17.4.4 How to Create a Custom Summary in the Transfer Panel

Create class named **apps.reporting.CustomTransferSummaryPanel** which implements the interface **com.calypso.apps.reporting.TransferSummaryPanel**.

This is invoked from **com.calypso.apps.reporting.TaskStationJFrame**.

Sample Code in calypsox/apps/reporting/

- SampleCustomTransferSummaryPanel.java

17.4.5 How to Create a Custom Summary in the Exception Panel

Create a class named **apps.reporting.CustomExceptionSummaryPanel** which implements the interface **com.calypso.apps.reporting.ExceptionSummaryPanel**.

This class will be invoked from **com.calypso.apps.reporting.TaskStationJFrame**.

Sample Code in calypsox/apps/reporting/

- SampleCustomExceptionSummaryPanel.java

17.4.6 How to add Custom Menu Items to the Task Station

Create a class named **apps.reporting.CustomTaskStationMenu** which implements the interface **com.calypso.apps.reporting.CustomTaskStationMenu**.

To create a custom action to be applied on trades, transfers or messages, implement the methods *handleWorkflowAction()* and *isWorkflowActionImplemented()*.

This class will be invoked from **com.calypso.apps.reporting.TaskStationUtil**.

Sample Code in calypsox/apps/

Menu items samples in :

- reporting/CustomTaskStationMenuMessage.java
- reporting/CustomTaskStationMenuTrade.java
- reporting/CustomTaskStationMenuTransfer.java
- reporting/Tag72InputCustomTaskStationMenuMessage.java

Action samples in:

- reporting/CustomTaskStationMenuTrade.java
- trading/AuthorizeTradeWindow.java

The Authorize trade action has been added to forbid the authorization of a manual amendment without checking the amended fields.

Insert the Authorize action in the needed transitions of your trade workflow. In this example, the Authorize action takes place between PENDING and VERIFIED. Each time an amendment is done manually, the trade goes to PENDING and another user must authorize it.

On the Task Station, select your trade and process it. Then choose the Authorize action. A popup window will prompt you to authorize or reject the trade.

17.4.7 How to Create Custom Columns in the Task Station

Create a class named **apps.reporting.CustomTaskStationColumn** which implements the interface **com.calypso.apps.reporting.CustomTaskStationColumn**.

This class will be invoked from **com.calypso.apps.reporting.TaskStationUtil**.

Sample Code in calypsox/apps/reporting/

SampleCustomTaskStationColumn.java

17.4.8 How to Apply Custom Validation to the Copy Message Panel

Create a class named **apps.reporting.CustomTSCopyMessageValidator** which implements the interface **com.calypso.apps.reporting.TSCopyMessageValidator**.

This class will be invoked from **com.calypso.apps.reporting.TSCopyMessagePanel**.

17.4.9 How to Create a Custom Copied Message

Create a class named **apps.reporting.CustomTSMessageHandler** that implements **com.calypso.apps.reporting.TSMessageHandler**.

This class will be invoked from **com.calypso.apps.reporting.TSCopyMessagePanel**.

17.4.10 How to Apply Custom Validation to the Assign Window

Create class named **apps.refdata.CustomTSTransferValidator** which implements the interface **com.calypso.apps.reporting.TSTransferValidator**.

This class will be invoked from **com.calypso.apps.reporting.TSAssignmentJFrame**.

17.4.11 How to Apply Custom Validation to the Netting Manager Window

Create a class named **apps.refdata.CustomTSNettingManagerValidator** that implements the interface **com.calypso.apps.reporting.TSTransferValidator**.

This class will be invoked from **com.calypso.apps.reporting.TSNettingManagerJFrame**.

17.4.12 How to Apply Custom Validation to the Split Panel

Create a class named **apps.reporting.CustomTSSplitTransferValidator** that implements **apps.reporting.TSSplitTransferValidator**.

This class will be invoked from **com.calypso.apps.reporting.TSSplitPanel**.

17.4.13 How to Create a Custom PO SDI Selection

You can override the PO SDI selection whenever the counterparty SDI is manually selected in the Netting Manager, Assign and Split windows. See the Tip section under Section 12.9, "Creating a Custom BO Trade Display," on page 123 for details.

17.4.14 How to Apply Custom Completion Rules

You can specify custom rules to determine whether a task can be completed or not.

Create a class named **`apps.reporting.CustomTaskStationCompleteTask`** that implements **`com.calypso.apps.reporting.TaskStationCompleteTask`**.

Sample Code in `calypsox/apps/reporting/`

`CustomTaskStationCompleteTask.java`

18 Cache Framework

The Cache Framework allows any cache mechanism to be plugged-in at the data level. For example, accounts can be cached using the LRU cache mechanism, while postings can be cached using a custom cache mechanism.

Out-of-the-box, the following cache mechanisms are available:

- The LRU (Least Recently Used) cache — This is a fixed size LRU cache. The cache has a maximum size specified when it is created. When an item is added to the cache, if the cache is already at the maximum size, the least recently used item is deleted, then the new item is added.
- The LFU (Least Frequently Used) cache — The behavior is identical to that of HashtableCache until the cache gets full. If and when the cache gets full, all the items in cache are sorted based on their popularity, and the 10% least popular objects in the cache are removed.

  Popularity is defined as the number of requests for that particular object in cache, so a popular object will be requested more than an unpopular object.

18.1 How to Disable Caching for a Given Object

Create a class named **`tk.util.<cache_name>CacheValidator`** that implements **`com.calypso.tk.util.cache.CacheValidator`**.

This class will be invoked from **`com.calypso.tk.core.CacheUtil`**.

Sample Code in `calypsox/tk/util/cache/`

`CurveCacheValidator.java`

19 The Extension Point Factory Framework

The Extension Point Factory provides a common mechanism to load classes that are extension points to Calypso. Extension points in Calypso are classes that provide additional functionality. For example, the **FilterSet** class can be extended to handle additional conditions by providing an implementation of the **CustomFilterInterface**.

Prior to the Extension Point Factory, the approach was to simply attempt loading a well known class name. If the class loaded, then obviously the extension is available. However, if an extension did not load and run, there was no means to determine the cause. The application returned a Null in both cases, when the load failed or if the class was missing.

Using the Extension Point Factory, it is possible to determine an exact cause of failure. In the event of a failure, a log entry is made with detailing the problem. An incorrect assignable type, a failure to load, or a missing class file all result in a throwable RuntimeException and a log entry to that effect.

```
public Object getExtensionPoint(  Class theType,
                                  String extensionName,
                                  String defaultExtensionClassname)
```

Where:

Table 19-1: Parameters for getExtensionPoint

Parameter	Description
theType	The type expected for the extension class.
extensionName	The name of the extension point without the package prefix.
defaultExtensionClassname	**No Backward compatibility**: Set to Null. This forces users to specify the extension point's classname (either fully qualified or without the package prefix) in a property name having the form: *extensionName*_EXTENSION_CLASSNAME where ***extensionName*** is the class name without the package prefix. On failure, a Null is returned and a RuntimeException is thrown and a log entry is made (wrong assignable type or extension point not found). On success, **ExtensionPointFactory** returns an instance of the extension type associated with the **extensionName**. — Or — **Support Backward Compatibility**: Set to the extension point's classname (either fully qualified or with the package prefix). For example, **CustomFilter, tk.marketdata.CustomFilter**, or **client.tk.marketdata.CustomFilter**. If **ExtensionPointFactory** does not find an explicitly named in the application's properties file, it then attempts to load the classname specified by **defaultExtensionClassname**. On failure, a Null is returned and a "failed to load" log entry is made. On success, **ExtensionPointFactory** returns an instance of the extension type associated with the **extensionName**.

20 Administration

20.1 How to customize the login dialg.

The following section allows you to add custom panels to the login dialog. These panels are for display use only, and do not modify the form inputs.

Create a class named `tk.util.ClientVersion` that implements `com.calypso.tk.util.ClientVersionInterface`.

This class will be invoked from `com.calypso.apps.util.CalypsoLoginDialog`.

20.2 How to Create Custom Version Information

Create a class named `tk.util.ClientVersion` that implements `com.calypso.tk.util.ClientVersionInterface`.

This class will be invoked from `com.calypso.apps.util.CalypsoLoginDialog`.

20.3 How to Create a Custom About Window

Create a class named **apps.main.ServerInfoDialog** that implements **com.calypso.apps.main.ServerInfoDialogInterface**.

This class is invoked from **com.calypso.apps.main.ServerInfoDialog** (the dialog that appears under **Main Entry > Help > About**).

Sample Code in calypsox/apps/main/

ServerInfoDialog.java

20.4 How to Create Custom Keyboard Accelerators

Create a class named **apps.util.DefaultCustomListener** which implements the interface **com.calypso.apps.util.CustomListener**.

This class will be invoked from **com.calypso.apps.util.AppUtil**.

Sample Code in calypsox/apps/util/

DefaultCustomListener.java

Demonstrates the binding of an action to the F1 key

20.5 How to Allow Custom Date Patterns

Create a class named **apps.util.CalypsoDateDocument** that implements **javax.swing.text.PlainDocument**.

Invoke this class from **com/calypso/apps/util/AppUtil**.

Sample Code in calypsox/apps/util/

CalypsoDateDocument.java

Shows support for various date formats, such as 14-Mar-05 and 14-Mar-2005.

20.6 How to Extend the Admin Window

Create a class named **apps.util.CustomExtendAdmin** that implements **com.calypso.apps.util.ExtendAdmin**.

Invoke this class from **com.calypso.apps.util.AdminFrame**.

Sample Code in calypsox/apps/util/

CustomExtendAdmin.java

This sample shows how to add a new menu item and a custom tab to the Admin window.

21 Developer's Notes

21.1 How to Add a Non-transient Attribute to an Externalizable Class

21.1.1 Release

You are adding the attribute under `release`.

1. Look at AUDIT_VERSION in **`release.src.com.calypso.tk.core.CalypsoVersion`**. For example, it is 90100.
2. In the *readExternal()* method of the Externalizable class in release, call *CalypsoVersion.checkAuditVersion(__auditVersion, 90100)*. This means that this attribute was added in version 90100.

21.1.2 Patch

You are patching the attribute into one code line.

1. Look at AUDIT_VERSION in **`patch.com.calypso.tk.core.CalypsoVersion`**. For example, it is 90200.
2. In the *readExternal()* method of the Externalizable class **in both the release and the patch**, call *CalypsoVersion.checkAuditVersion(__auditVersion, 90100, 90200)*. This means that this attribute was added in version 90100 and patched in 90200.

   You are patching the attribute into two code lines.

3. Look at AUDIT_VERSION in **`patch.com.calypso.tk.core.CalypsoVersion`**. For example, it is 90200.
4. In the *readExternal()* method of the Externalizable class **in the release and in both patches**, call *CalypsoVersion.checkAuditVersion(__auditVersion,* 80400, 80100, 60024*)*. This means that this attribute was added in version 80400, and patched in 80100 and 60024.

21.2 How to use the Comparator Factory

Do the following to create a comparator class:

1. Create a class named **`com.calypso.tk.util.Comparator<name>`** that implements **`java.util.Comparator`**.

2. Register the class with ComparatorFactory as shown in the ComparatorAuthorizable example, below:

```
public static Comparator getAuthComparator(){
        if (_authComp == null){
            _authComp = new ComparatorAuthorizable();
        }
        return _authComp;
    }
```

3. Access the comparator using the ComparatorFactory methods as shown below.

```
Comparator comparator = ComparatorFactory.getAuthComparator ();
```


Index

S

T

V

W

X

Contacting Calypso

Americas

SAN FRANCISCO

595 Market Street
Suite 1800
San Francisco, CA 94105
USA
T +1 415 817 2400
F +1 415 284 1222
info@calypso.com

NEW YORK

99 Park Avenue,
Suite 930
New York, NY 10016
USA
T +1 212 905 0700
F +1 212 905 0724
sales_nyc@calypso.com

Europe, Middle East, Africa

LONDON

One New Change, 6th Floor
London EC4M 9AF
United Kingdom
T +44 20 7826 2500
F +44 20 7826 2501
sales_london@calypso.com

PARIS

106/108 rue de La Boëtie
75008 Paris
France
T +33 1 44 50 13 99
F +33 1 44 50 12 84
sales_paris@calypso.com

FRANKFURT

Junghofstrasse 24
2nd floor
60313 Frankfurt Am Main
Germany
T +49 69 920389 0
F +49 69 920389 11
sales_frankfurt@calypso.com

JOHANNESBURG

Nelson Mandela Square
2nd Floor, West Tower
Maude Street
Sandton, 2196
South Africa
T +27 11 881 5708
F +27 11 881 5611
sales_johannesburg@calypso.com

COPENHAGEN

Calypso Technology Denmark
Regus City Center, Office 308
Larsbjornstraede 3,1454 Copenhagen
T +45 33 37 71 79
F +45 33 32 43 70
sales_copenhagen@calypso.com

MOSCOW

Office number 532
4, 4th Lesnoy pereulok
125047 Moscow Russia
T +7 495 663 80 63
F +7 495 225 8500
sales_moscow@calypso.com

Asia Pacific

TOKYO

Kojimachi Place 10F
2-3 Kojimachi
Chiyoda-ku
Tokyo 102-0083
Japan
T +81 (0)3 5214 1800
F +81 (0)3 5214 1801
sales_tokyo@calypso.com

SINGAPORE

1 Phillip Street, #12-02
Singapore 048692
Singapore
T +65 6372 1121
F +65 6372 1161
sales_singapore@calypso.com

SYDNEY

Level 4, 95 Pitt Street
Sydney, NSW 2000
Australia
T +61 2 8249 8115
F +61 2 8249 8116
sales_sydney@calypso.com

MUMBAI

Unit 401, Akruti Center Point
MIDC Central Road
Near Marol Tel. Exchange
MIDC, Andheri (E)
Mumbai 400 093
India
T +91 22 6681 5600
F +91 22 6681 5611
sales_mumbai@calypso.com

CHENNAI

Plot No.23 & 24
Door No.33
South Beach Avenue
MRC Nagar, Chennai
Tamil Nadu 600028
India
T +91 44 4347 5350
F +91 44 4347 5378
saleschennai@calypso.com

HONG KONG

Unit 1238, Level 12, Infinitus Plaza,
199 Des Voeux Road Central,
Hong Kong
T +852 3478 3796
+852 3478 3797
F +852 3478 3880
sales_hongkong@calypso.com