InfoReach

Engineered for what's next

TMS / Trade Management System

TMS Java Programmer's Guide

Version 9.0

Contents

Introduction

TMS API allows users to interact programmatically with TMS server components. The API consists of two sets of methods where methods of the first ("subscription") set can be used to subscribe custom application component to the TMS event flow, and methods of the other ("instruction") set can be used to directly instruct the TMS to perform an action. The event flow exposed to the custom application in real time includes all order-related information, portfolio-related information, market data events, custom record events, position-related information, and report data. The events notify the application components of any changes taking place in the system and based on that information the instructions can be given to the TMS to perform a set of actions. Sending/modifying/canceling an order, creating new portfolio, modifying set of portfolio's targets, starting/pausing an analytic that automatically sends order waves are some of the actions available to the users' applications.

In addition to the remote API, the mechanism is provided to plug in custom **AutoTrader** analytics, message preprocessors, filters, id generators, GUI defaulters, etc.

API Types: In-Process, Remote and Light Remote

The **in-process** (a.k.a. AutoTrader) API allows developing of the AutoTrader analytic modules. The modules developed with the in-process API will run inside the TMS process space. There is a restriction of one AutoTrader module per one market portfolio. The developer of the analytic module would not need to worry about the thread safety. In most cases, the analytic modules will be purely algorithmic and won't do their own state management. Instead, they would work with the state of the targets and the orders as well as AutoTrader parameters. By subscribing to the order/target event flow a module gets notifications and it can make decisions on whether orders should be sent, modified, canceled, etc. Due to the real-time nature of the TMS and the fact that a given portfolio is locked on any given invocation of analytic module, the code should avoid any heavy processing (e.g. expensive calls to external application or databases). The AutoTrader module can modify its own parameters and control state of individual portfolio targets (e.g. pause/resume/terminate).

The **remote** (a.k.a.external) API allows building external applications that do not run inside the TMS process space and can communicate with the TMS remotely. The external apps can instruct the TMS to create new portfolios, add/remove/modify targets in portfolios, subscribe for event flow, make decisions on whether orders should be sent, modified, canceled, etc.

The **light remote** API is basically a lightweight version of the remote API. Unlike the latter it does not involve initializing various TMS sub-systems inside the external application. The overall API is identical, but some sophisticated remote API capabilities are not supported by the light remote API. See [TMS HOME]/backend/samples/java/TMSClientAppRemote/readme.html for details.

All event notifications to the external applications are asynchronous, no portfolio locks are held, and thus the state changes inside the TMS processes can take place while the application is processing current event. This means that there is a possibility for race conditions (e.g. trying to modify an order while a fill is being processed in TMS).

Normally, the external API is used to create applications implementing "macro-models" for portfolio/target generation/modification and the in-process API is used for creating execution logic (sending/monitoring market orders off the portfolio targets).

The sections below describe the external API usage. In section "Developing Custom Algorithms" the internal API is discussed.

Using "subscription" set of ITMSRemoteClient and ITMSRemoteMarketPortfolio interfaces

An external application can use *subscribe*() methods of the ITMSRemoteClient interface to monitor system-wide order, portfolio, position, market data event flow and it can use *subscribe*() methods of the ITMSRemoteMarketPortfolio interface to monitor portfolio-specific event flow. To get a handle to ITMSRemoteClient use method TMSClientSystem.getRemoteClient(name). The name can be obtain by calling TMSClientSystem.getRemoteClientNames() (see sample JAVA app for more details). To get a handle to ITMSRemoteMarketPortfolio use method *getMarketPortfolio(String portfolioName)* of the ITMSRemoteClient interface.

The following sections describe the methods of the "subscription" set in more details.

Subscribing for order data through ITMSRemoteClient

- *void subscribeForOrderData(UserTicket userTicket, ITMSOrderEventRemoteListener listener, boolean includePortfolioOrders) throws TMSException*
  Subscribes the listener module for events carrying the information about new orders or changes in the existing orders' fields. For example, each time a fill arrives for an order the FillQty field of an order will change and the method onOrderUpdate() of the application's listening module will be called. The listening module must implement interface ITMSOrderEventRemoteListener. See "Listener interfaces" section for more details. This method can be used to only monitor orders that were NOT sent as portfolio target slices if parameter *includePortfolioOrders* is set to "false". If it is set to "true" then events with information about all orders in the system will be sent to the listener.

- *void subscribeForOrderData(UserTicket userTicket, ITMSOrderEventRemoteListener listener, String filterExpression) throws TMSException*
  Subscribes the listener module for events carrying the information about new orders or changes in the existing orders' fields for those orders that match the specified filtering expression. For example, given the expression "Instrument='IBM'", each time a fill arrives for an IBM order the FillQty field of an order will change and the method onOrderUpdate() of the application's listening module will be called. The listening module must implement interface ITMSOrderEventRemoteListener. See "Listener interfaces" section for more details. If during the lifetime of the order its attribute changed so that it stops matching the filtering expression then the application's listening module's method onOrderFilteredOut() will be called.

- *void unsubscribeFromOrderData(UserTicket userTicket, ITMSOrderEventRemoteListener listener) throws TMSException*
  Unsubscribes the module from order-related event flow.

Subscribing for order data through ITMSRemoteMarketPortfolio

- *void subscribeForOrderData(UserTicket userTicket, ITMSOrderEventRemoteListener listener) throws TMSException*
  Subscribes the listener module for events carrying the information about new orders or changes in the existing orders' fields for those orders that belong to this portfolio. For example, each time a fill arrives for an order the FillQty field of an order will change and the method onOrderUpdate() of the application's listening module will be called. The listening module must implement interface ITMSOrderEventRemoteListener. See "Listener interfaces" section for more details. This is used to monitor orders that were sent as this portfolio's

target slices.

- *void subscribeForOrderData(UserTicket userTicket, ITMSOrderEventRemoteListener listener, String filterExpression) throws TMSException*
  Subscribes the listener module for events carrying the information about new orders or changes in the existing orders' fields for those orders that belong to this portfolio and match the specified filtering expression. For example, given the expression "Instrument='IBM'", each time a fill arrives for an IBM order the FillQty field of an order will change and the method onOrderUpdate() of the application's listening module will be called. The listening module must implement interface ITMSOrderEventRemoteListener. See "Listener interfaces" section for more details. If during the lifetime of the order its attributes change so that it stops matching the filtering expression then the application's listening module's method onOrderFilteredOut() will be called.

- *void unsubscribeFromOrderData(UserTicket userTicket, ITMSOrderEventRemoteListener listener) throws TMSException*
  Unsubscribes the module from order-related event flow.

Subscribing for portfolio data through ITMSRemoteClient

- *void subscribeForPortfolioData (UserTicket userTicket, ITMSMarketPortfolioEventRemoteListener listener) throws TMSException*
  *S*ubscribes the module for events carrying the information about new portfolios or any field changes of an existing system portfolio. For example, every time a fill arrives for an order released from target of a portfolio the field FillQty of the portfolio will change and the listener will be notified. The listening module must implement interface *ITMSMarketPortfolioEventRemoteListener*. See "Listener interfaces" section for more details.

- *void subscribeForPortfolioData (UserTicket userTicket, ITMSMarketPortfolioEventRemoteListener listener, boolean includeRecordUpdates, String filterExpression, String[] fieldNames) throws TMSException*
  *S*ubscribes the module for events carrying the information about new portfolios or any field changes of an existing system portfolio. By setting *includeRecordUpdates* parameter to false the user can restrict the system to only notify the listening app only about new portfolios and not send events every time something changes in an existing portfolio. It is also possible to restrict the system to only send events from portfolios matching the specified filtering expression. The *fieldNames* parameter can be specified to limit the set of fields in the events only to the pertinent ones. The listening module must implement interface *ITMSMarketPortfolioEventRemoteListener*. See "Listener interfaces" section for more details. If during the lifetime of the portfolio its attributes change so that it stops matching the filtering expression then the application's listening module's method onPortfolioFilteredOut() will be called.

- *void unsubscribeFromPortfolioData(UserTicket userTicket, ITMSMarketPortfolioEventRemoteListener listener) throws TMSException*
  Unsubscribes the module from portfolio-related event flow.

Subscribing for portfolio data through ITMSRemoteMarketPortfolio

- *void subscribeForPortfolioData (UserTicket userTicket, ITMSMarketPortfolioEventRemoteListener listener) throws TMSException*
  *S*ubscribes the module for events carrying the information about changes in this portfolio

fields. For example, every time a fill arrives for an order released from target of this portfolio the field FillQty of the portfolio will change and the listener will be notified. The listening module must implement interface *ITMSMarketPortfolioEventRemoteListener*. See "Listener interfaces" section for more details.

- *void subscribeForPortfolioData (UserTicket userTicket, ITMSMarketPortfolioEventRemoteListener listener, boolean includeRecordUpdates, String[] fieldNames) throws TMSException*
  *S*ubscribes the module for events carrying the information about changes in this portfolio fields. The *fieldNames* parameter can be specified to limit the set of fields in the events only to the pertinent ones. For example, every time a fill arrives for an order released from target of this portfolio the field FillQty of the portfolio will change and the listener will be notified. If *includeRecordUpdates* is set to false then only event notifying of this portfolio removal will be sent to the app. The listening module must implement interface *ITMSMarketPortfolioEventRemoteListener*. See "Listener interfaces" section for more details.

- *void unsubscribeFromPortfolioData(UserTicket userTicket, ITMSMarketPortfolioEventRemoteListener listener) throws TMSException*
  Unsubscribes the module from portfolio-related event flow.

Subscribing for target data through ITMSRemoteMarketPortfolio

- *void subscribeForTargetData (UserTicket userTicket, ITMSMarketTargetEventRemoteListener listener) throws TMSException*
  *S*ubscribes the module for events carrying the information about changes in this portfolio's targets fields. For example, every time a fill arrives for an order released from target of this portfolio the field FillQty of the target will change and the listener will be notified. The listening module must implement interface *ITMSMarketTargetEventRemoteListener*. See "Listener interfaces" section for more details.

- *void subscribeForTargetData (UserTicket userTicket, ITMSMarketTargetEventRemoteListener listener, boolean includeRecordUpdates, String filterExpression, String[] fieldNames) throws TMSException*
  *S*ubscribes the module for events carrying the information about changes in this portfolio's targets fields. For example, every time a fill arrives for an order released from target of this portfolio the field FillQty of the target will change and the listener will be notified. By setting *includeRecordUpdates* parameter to false the user can restrict the system to only notify the listening app only about new targets and not send events every time something changes in an existing target. It is also possible to restrict the system to only send events from targets matching the specified filtering expression. The *fieldNames* parameter can be specified to limit the set of fields in the events only to the pertinent ones. The listening module must implement interface *ITMSMarketTargetEventRemoteListener*. See "Listener interfaces" section for more details. If during the lifetime of the target its attributes change so that it stops matching the filtering expression then the application's listening module's method onTargetFilteredOut() will be called.

- *void unsubscribeFromTargetData(UserTicket userTicket, ITMSMarketTargetEventRemoteListener listener) throws TMSException*
  Unsubscribes the module from portfolio-related event flow.

Subscribing for market data through ITMSRemoteClient

- *void subscribeForMarketData(UserTicket userTicket, String instrumentId, ITMSMarketDataEventListener listener) throws TMSException*
  Subscribes the listener module for market data events for the given instrument. For example, each time a tick arrives for instrument *instrumented* the method onMarketDataUpdate() of the application's listening module will be called. The listening module must implement interface ITMSMarketDataEventListener. See "Listener interfaces" section for more details.

- *void unsubscribeFromMarketData(UserTicket userTicket, String instrumentId, ITMSMarketDataEventListener listener) throws TMSException*
  Unsubscribes the module from market data events for the given instrument.

- *void subscribeForMarketData(UserTicket userTicket, String[] instrumentIds, ITMSMarketDataEventListener listener) throws TMSException*
  Subscribes the listener module for market data events for the given list of instruments.

- *void unsubscribeFromMarketData(UserTicket userTicket, String[] instrumentIds, ITMSMarketDataEventListener listener) throws TMSException*
  Unsubscribes the module from market data events for the given list of instruments.

Subscribing for custom data through ITMSRemoteClient

When custom record data source is plugged into TMS server backend it is possible to subscribe remote applications to this source's event flow

- *void ITMSRemoteClient.subscribeForCustomRecordData(UserTicket userTicket, String datasourceName, String recordId, ITMSCustomRecordDataEventListener listener) throws TMSException*
  Subscribes the listener module for custom record events from the specified source for the given record id.

- *void ITMSRemoteClient.unsubscribeFromCustomtData(UserTicket userTicket, String datasourceName, String recordId, ITMSCustomRecordDataEventListener listener) throws TMSException*
  Unsubscribes the module from custom record events for the given record id.

- *void ITMSRemoteClient.subscribeForCustomtData(UserTicket userTicket, String datasourceName, String[]recordIds, ITMSCustomRecordDataEventListener listener) throws TMSException*
  Subscribes the listener module for custom record events from the specified source for the given record ids.

- *void ITMSRemoteClient.unsubscribeFromCustomData(UserTicket userTicket, String datasourceName, String[]recordIds, ITMSCustomRecordDataEventListener listener) throws TMSException*
  Unsubscribes the module from custom record events for the given list of record ids.

Subscribing for position data through ITMSRemoteClient

InfoReach TMS can be configured to maintain position information for instruments within multiple categories. For example, if TMS is configured to have Account and Strategy position categories then for each instrument the position information per each account and per each strategy, as well, as global instrument position will be maintained. All these position numbers are accessible through API methods:

- *void subscribeForCategoryInstrumentPositionData (UserTicket userTicket, String categoryType, String categoryName, String instrumentId, ITMSInstrumentPositionEventRemoteListener rlistener) throws TMSException*
  Subscribes the module for events carrying the information about changes in instrument position within specified category. For example, specifying categoryType="Account" and categoryName="Acc000000" and instrument="IBM" will subscribe the module to the info about IBM position for account Acc000000 and every time a fill arrives for an IBM order for this account the position will change and the listener will be notified. The listening module must implement interface *ITMSInstrumentPositionEventRemoteListener*. See "Listener interfaces" section for more details.

- *void subscribeForCategoryInstrumentPositionData (UserTicket userTicket, String categoryType, String categoryName, ITMSInstrumentPositionEventRemoteListener listener) throws TMSException*
  Subscribes the module for events carrying the information about changes in all instrument positions within specified category. For example, specifying categoryType="Account" and categoryName="Acc000000" will subscribe the module to position events for all instruments traded for account Acc000000.

- *void unsubscribeFromCategoryInstrumentPositionData(UserTicket userTicket, ITMSInstrumentPositionEventRemoteListener listener) throws TMSException*
  Unsubscribes the module from instrument position events.

- *void subscribeForCategoryPositionData (UserTicket userTicket, String categoryType, String categoryName, ITMSCategoryPositionEventRemoteListener listener) throws TMSException*
  Subscribes the module for events carrying the information about changes in category position values. For example, specifying categoryType="Account" and categoryName="Acc000000" will subscribe the module to position events carrying total numbers for account Acc000000. The listening module must implement interface *ITMSCategoryPositionEventRemoteListener*. See "Listener interfaces" section for more details.

- *void unsubscribeFromCategoryPositionData(UserTicket userTicket, ITMSCategoryPositionEventRemoteListener listener) throws TMSException*
  Unsubscribes the module from category position events.

- *void subscribeForGlobaInstrumentPositionData(UserTicket userTicket, String instrument, ITMSInstrumentPositionEventRemoteListener listener) throws TMSException*
  Subscribes the module for events carrying the information about changes in specified global (system-wide) instrument position. For example, every time a fill arrives for an order for a specific instrument the global instrument position will change and the listener will be notified. The listening module must implement interface *ITMSPositionRemoteListener*. See "Listener interfaces" section for more details.

- *void subscribeForGlobaInstrumentPositionData(UserTicket userTicket, TMSInstrumentPositionEventRemoteListener listener) throws TMSException*
  Subscribes the module for events carrying the information about changes in ANY

instrument's global (system-wide) position. For example, every time a fill arrives for an order for any instrument this instrument's global position will change and the listener will be notified. The listening module must implement interface *ITMSPositionRemoteListener*. See "Listener interfaces" section for more details.

- *void unsubscribeFromGlobaInstrumentPositionData (UserTicket userTicket, ITMSInstrumentPositionEventRemoteListener listener) throws TMSException*
  Unsubscribes the module from global instrument position events.

- *void subscribeForGlobaPositionData(UserTicket userTicket, ITMSCategoryPositionEventRemoteListener listener) throws TMSException*
  Subscribes the module for events carrying the information about changes in any category position values. The listening module must implement interface *ITMSCategoryPositionEventRemoteListener*. See "Listener interfaces" section for more details.

- *void unsubscribeFromGlobaPositionData (UserTicket userTicket, ITMSCategoryPositionEventRemoteListener listener) throws TMSException*
  Unsubscribes the module from global category position events.

Subscribing for position data through ITMSRemoteMarketPortfolio

InfoReach TMS maintains instrument positions within individual portfolios (useful in case there is more than one target for the same instrument in the same portfolio). An application can subscribe to position data of the portfolio with these calls:

- *void subscribeForInstrumentPositionData (UserTicket userTicket, String instrumentId, ITMSInstrumentPositionEventRemoteListener rlistener) throws TMSException*
  Subscribes the module for events carrying the information about changes in specified instrument's position within this portfolio. The listening module must implement interface *ITMSInstrumentPositionEventRemoteListener*. See "Listener interfaces" section for more details.

- *void subscribeForInstrumentPositionData (UserTicket userTicket, ITMSInstrumentPositionEventRemoteListener rlistener) throws TMSException*
  Subscribes the module for events carrying the information about changes in ALL instruments' position within this portfolio. The listening module must implement interface *ITMSInstrumentPositionEventRemoteListener*. See "Listener interfaces" section for more details.

- *void unsubscribeFromInstrumentPositionData(UserTicket userTicket, ITMSInstrumentPositionEventRemoteListener listener) throws TMSException*
  Unsubscribes the module from instrument position events.

- *void subscribeForPortfolioPositionData (UserTicket userTicket, String categoryType, String categoryName, ITMSCategoryPositionEventRemoteListener listener) throws TMSException*
  Subscribes the module for events carrying the information about changes in total portfolio position. The listening module must implement interface *ITMSCategoryPositionEventRemoteListener*. See "Listener interfaces" section for more details.

- *void unsubscribeFromPortfolioPositionData(UserTicket userTicket, ITMSCategoryPositionEventRemoteListener listener) throws TMSException*
  Unsubscribes the module from category position events.

Subscribing for AutoTrader data through ITMSRemoteMarketPortfolio

- *void subscribeForAutoTraderData(UserTicket userTicket, ITMSAutoTraderEventRemoteListener listener) throws RPTException*
  Subscribes listener for this portfolio's AutoTrader events. The listening module must implement interface ITMSAutoTraderEventRemoteListener. If subscribed, onAutoTrader*() methods will be invoked.

- *void unsubscribeFromAutoTraderData(UserTicket userTicket, ITMSAutoTraderEventRemoteListener listener) throws RPTException*
  Unsubscribes the listener.

Subscribing for Wave data through ITMSRemoteMarketPortfolio

- *void subscribeForWaveData(UserTicket userTicket, ITMSWaveEventRemoteListener listener) throws RPTException*
  Subscribes listener for this portfolio's Wave events. The listening module must implement interface ITMSWaveEventRemoteListener. If subscribed, onWave*() methods will be invoked.

- *void unsubscribeFromWaveData(UserTicket userTicket, ITMSWaveEventRemoteListener listener) throws RPTException*
  Unsubscribes the listener.

Subscribing for action errors, warnings, alerts

Most of the methods from the "instruction" set of the TMS API are executed by the TMS asynchronously. This has the effect of the application not knowing right away whether a certain method was executed successfully. In case an application wants to be notified of errors/warnings/alerts caused by a specific method call it should subscribe the action listening module to the notifications sent by TMS and remember the action id returned from each called method. When the error/warning/alert notification arrives the id in the arrived event can be matched to the recorded action ids. Besides errors, warnings and alerts the notifications of successful completion of the actions is also sent to the application.

- *void addActionEventListener(IActionEventListener listener)*
  Subscribes the listening module for action errors/warnings/alerts.

- *void removeActionEventListener (IActionEventListener listener)*
  Unsubscribe the listener.

Listener interfaces

Interface ITMSOrderEventRemoteListener

The listening component's methods from this interface will be called by the TMS if subscribeForOrderData() call was issued by the application.

- *void onOrderAdded(Object orderId, IRecord record)*
  This method is called when a new order is issued in the TMS. Use methods of IRecord interface to examine values of the record representing this order.

- *void onOrderUpdate(Object orderId, IRecordUpdate recordUpdate)*
  This method is called when any of the order fields change (e.g. status or fillQty). Use methods of *IRecordUpdate* interface to examine changed values of the record representing this order.

- *void onOrderDataFeedDisconnected()*
  This method is called when the listener is disconnected from the remote event source for any reason. The listener would try to reconnect automatically.

- *void onOrderDataFeedReconnected()*
  This method is called when the listener is automatically reconnected to the remote source after the disconnect. **NOTE: after the reconnect the *onOrderAdded()* method will be called for all orders in TMS.**

- *void onInitialStateReceived()*
  When application issues a subscribe() call it first will be notified of all existing market orders in the system(or portfolio) through the *onOrderAdded ()* method and then it will continue to receive additional events as orders get added/removed/modified. The *onInitialStateReceived()* method of the listener will be called immediately after the last *onOrderAdded* () method for an existing order is called to mark the end of the initial state.

- *void onOrderFilteredOut(Object ordereId)*
  When application subscribes to order data with order filtering expression this method will be called when some particular order stops satisfying the filtering expression as a result of some changes in order's fields.

Interface ITMSMarketPortfolioEventRemoteListener

The listening component's methods from this interface will be called by the TMS if subscribeForPortfolioData() call was issued by the application.

- *void onPortfolioAdded (String portfolioName, IRecord record)*
  This method is called when a new portfolio is added to TMS. Use methods of IRecord interface to examine values of the record representing this portfolio.

- *void onPortfolioUpdate(Object portfolioName, IRecordUpdate recordUpdate)*
  This method is called when any of the portfolio's fields change (e.g. PnL or fillQty). Use methods of *IRecordUpdate* interface to examine changed values of the record representing this portfolio.

- *void onPortfolioRemoved(String portfolioName)*
  This method is called when portfolio is removed.

- *void onPortfolioDataFeedDisconnected()*
  This method is called when the listener is disconnected from the remote event source for any reason. The listener would try to reconnect automatically.

  void onPortfolioDataFeedReconnected()
  This method is called when the listener is automatically reconnected to the remote source

after the disconnect. **NOTE: after the reconnect the *onPortfolioAdded()* method will be called for all portfolios in TMS.**

- *void onInitialStateReceived()*
  When application issues a subscribe() call it first will be notified of all existing market portfolios in the system through the *onPortfolioAdded()* method and then it will continue to receive additional events as portfolios get added/removed/modified. The *onInitialStateReceived()* method of the listener will be called immediately after the last *onPortfolioAdded*() method for an existing portfolio is called to mark the end of the initial state.

- *void onPortfolioFilteredOut(String portfolioName)*
  When application subscribes to portfolio data with portfolio filtering expression this method will be called when some particular portfolio stops satisfying the filtering expression as a result of some changes in portfolio's fields.

Interface ITMSMarketTargetEventRemoteListener

nn

The listening component's methods from this interface will be called by the TMS if subscribeForTargetData() call was issued by the application.

- *void onTargetAdded (Object targetId, IRecord record)*
  This method is called when a new target is added to the portfolio. Use methods of IRecord interface to examine values of the record representing this portfolio.

- *void onTargetUpdate(Object targetId, IRecordUpdate recordUpdate)*
  This method is called when any of the target's fields change (e.g. PnL or fillQty). Use methods of *IRecordUpdate* interface to examine changed values of the record representing this target.

- *void onTargetRemoved(Object targetId)*
  This method is called when target is removed.

- *void onTargetPaused(Object targetId)*
  This method is called when target is paused.

- *void onTargetResumed(Object targetId)*
  This method is called when target is resumed.

- *void onTargetTerminated(Object targetId)*
  This method is called when target is terminated.

- *void onTargetDataFeedDisconnected()*
  This method is called when the listener is disconnected from the remote event source for any reason. The listener would try to reconnect automatically.

- *void onTargetDataFeedReconnected()*
  This method is called when the listener is automatically reconnected to the remote source after the disconnect. **NOTE: after the reconnect the *onTargetAdded()* method will be called for all targets in the portfolio.**

- *void onInitialStateReceived()*
  When application issues a subscribe() call it first will be notified of all existing targets in the portfolio through the *onTargetAdded()* method and then it will continue to receive additional

events as targets get added/removed/modified. The *onInitialStateReceived()* method of the listener will be called immediately after the last *onTargetAdded*() method for an existing target is called to mark the end of the initial state.

- *void onPortfolioFilteredOut(Object targetId)*
  When application subscribes to target data with target filtering expression this method will be called when some particular target stops satisfying the filtering expression as a result of some changes in target's fields.

Interface ITMSMarketDataEventListener

The listening component's methods from this interface will be called by the TMS if one of the subscribeForMarketData() calls was issued by the application.

- *void onMarketDataUpdate(String instrumentId, IRDRecord record)*
  called when the TMS processes a market data tick for an instrument for which there was a subscription.

- *void onMarketDataFeedDisconnected()*
  called when the TMS detects the disconnect from the market data feed.

- *void onMarketDataFeedReconnected ()*
  called when the TMS detects the re-connect to the market data feed.

Interface ITMSCustomRecordDataEventListener

The listening component's methods from this interface will be called by the TMS if one of the subscribeForCustomRecordData() calls was issued by the application.

- *void onCustomRecordDataUpdate(String dataSourceName, String recordId, IRDRecord record)*
  called when the TMS processes a custom record update for a record id for which there was a subscription.

- *void onCustomRecordDataFeedDisconnected()*
  called when the TMS detects the disconnect from the custom data feed.

- *void onCustomRecordDataFeedReconnected()*
  called when the TMS detects the re-connect to the custom data feed.

Interface ITMSInstrumentPositionEventRemoteListener

The listening component's methods from this interface will be called by the TMS if one of the subscribeForCategoryInstrumentPositionData() or subscribeForGlobalInstrumentPositionData() calls was issued by the application.

- *void onInstrumentPositionAdded(String instrument, IRecord instrumentPositionRecord)*
  This method is called when position for a new instrument is recorded in TMS. Use methods of IRecord interface to examine field values of the received record representing this position.

- *void onPositionUpdate* (*String instrument, IRecordUpdate instrumentPositionRecord Update)*
  This method is called when any of the position fields change (e.g. BuyFillQty). Use methods of *IRecordUpdate* interface to examine changed field values of the record representing this position.

- *void onPositionDataFeedDisconnected()*
  This method is called when the listener is disconnected from the remote event source for any reason. The listener would try to reconnect automatically.

- *void onPositionDataFeedReconnected()*
  This method is called when the listener is automatically reconnected to the remote source after the disconnect. **NOTE: after the reconnect the *onInstrumentPositionAdded()* method will be called for all pertinent instrument positions in TMS.**

- *void onInitialStateReceived()*
  When application issues a subscribe() call it first will be notified of all existing positions in the system through the *onInstrumentPositionAdded ()* method and then it will continue to receive additional events as positions get added/removed/modified. The *onInitialStateReceived()* method of the listener will be called immediately after the last *onInstrumentPositionAdded* () method for an existing position is called to mark the end of the initial state.

Interface ITMSCategoryPositionEventRemoteListener

The listening component's methods from this interface will be called by the TMS if one of the subscribeForCategoryInstrumentPositionData() or subscribeForCategoryPositionData() or subsribeForGlobalPositionData() calls was issued by the application.

- *void onCategoryPositionAdded(String categoryType, String categoryName, IRecord positionRecord)*
  This method is called when position for a new category is recorded in TMS. Use methods of IRecord interface to examine values of the record representing this position.

- *void onCategoryPositionUpdate(String categoryType, String categoryName, IRecordUpdate positionRecordUpdate)*
  This method is called when any of the position fields change (e.g. BuyFillQty). Use methods of *IRecordUpdate* interface to examine changed values of the record representing this position.

- *void onCategoryPositionDataFeedDisconnected()*
  This method is called when the listener is disconnected from the remote event source for any reason. The listener would try to reconnect automatically.

- *void onCategoryPositionDataFeedReconnected()*
  This method is called when the listener is automatically reconnected to the remote source after the disconnect. **NOTE: after the reconnect the *onCategoryPositionAdded()* method will be called for all portfolios in TMS.**

- *void onInitialStateReceived()*
  When application issues a subscribe() call it first will be notified of all existing positions in the system through the *onCategoryPositionAdded ()* method and then it will continue to receive additional events as positions get added/removed/modified. The

onInitialStateReceived() method of the listener will be called immediately after the last *onCategoryPositionAdded* () method for an existing position is called to mark the end of the initial state.

Interface ITMSAutoTraderEventRemoteListener

The listening component's methods from this interface will be called by the TMS if the subscribeForAutoTraderData() call was issued by the application.

- *void onAutoTraderCreated ()*
  This method is called when new AutoTrader module is assigned to the portfolio.

- *void onAutoTraderTerminated()*
  This method is called when the AutoTrader module is terminated.

- *void onAutoTraderPaused()*
  This method is called when the AutoTrader module is paused.

- *void onAutoTraderResumed()*
  This method is called when the AutoTrader module is resumed.

- *void onAutoTraderDataFeedDisconnected()*
  called when the TMS detects the disconnect from the AutoTrader data feed.

- *void onAutoTraderDataFeedReconnected()*
  called when the TMS detects the re-connect to the AutoTrader data feed.

Interface ITMSWaveEventRemoteListener

The listening component's methods from this interface will be called by the TMS if the subscribeForWaveData() call was issued by the application.

- *void onWaveAdded (String portfolioName, String waveId, IRecord record)*
  This method is called when new wave of order is sent. The record parameter contains the fields with summary values for the wave. For example, field OrdQty will contain total order quantity of orders that belong to this wave.

- *void onWaveUpdate (String portfolioName, String waveId, IRecordUpdate record)*
  This method is called when the wave numbers change (e.g. as a result of some wave orders being cancelled). The record can then be examined to obtain the new wave values.

- *void onWaveDataFeedDisconnected ()*
  This method is called when the listener is disconnected from the remote event source for any reason. The listener would try to reconnect automatically.

- *void onWaveDataFeedReconnected ()*
  This method is called when the listener is automatically reconnected to the remote source after the disconnect. **NOTE: after the reconnect the *onWaveAdded()* method will be called for all waves in the portfolio.**

- *void onInitialStateReceived()*
  When application issues a subscribe() call it first will be notified of all existing waves in

the portfolio through the *onWaveAdded()* method and then it will continue to receive additional events as waves get sent/ modified. The *onInitialStateReceived()* method of the listener will be called immediately after the last *onWaveAdded*() method is called to mark the end of the initial state.

Interface IActionEventListener

The listening component's methods from this interface will be called by the TMS if the *addActionEventListener* () call was issued by the application.

- *void onErrorOccurred(long actionId, String actionDescription, Throwable exception, Object extraInfo)*
  Notifies application about errors that have occurred as a result of performing the action with id actionId.

- *void onFinished(long actionId, String actionDescription, Object extraInfo)*
  Notifies application that action with id actionId was completed. Note that application can be notified of the action completion even if it was previously notified of action errors.

- *void onAlert(long actionId, String actionDescription, String message, Object extraInfo, boolean isUrgent)*
  If an action posts an alert during execution the application will be notified through this callback.

- *void onEvent(long actionId, String actionDescription, Obect event)*
  If an action posts a custom event during execution the application will be notified through this callback.

- *void onProgressReport(long actionId, String actionDescription, String statusMessage, int currentStep, int totalSteps, Object extraInfo)*
  Some actions notify calling application about their progress. The currentStep and totalSteps parameters can be used to calculate how much work has been done by the action already.

Using "instruction" set of ITMSRemoteClient API

Getting resources

- *ITMSRemoteMarketPortfolio getMarketPortfolio(String name) throws TMSException*
  Get an instance of the portfolio wih the given name.

- *ITMSRemoteMarketPortfolio addMarketPortfolio(UserTicket userTicket, String name, int type) throws TMSException*
  Add new portfolio with of the specified type and name. There are currently two portfolio types that can be used: 0 – index portfolio (qauntity and side of the target is not defined), 1 – market portfolio (quantity and side of the target is defined). This method returns a handle to the newly created portfolio.

- *String[] getPortfolioList (UserTicket ticket) throws TMSException*
  Returns the list of all portfolios accessible by the user specified in the userTicket.

- *String[] getPortfolioList (UserTicket ticket, int type) throws TMSException*
  Returns the list of all portfolios of specified type accessible by the user specified in the userTicket.
  The type is one of:

  INDEX_MARKET_PORTFOLIO = 0;
  PURE_MARKET_PORTFOLIO = 1;
  LINKED_MARKET_PORTFOLIO = 2;
  CORRELATED_MARKET_PORTFOLIO = 3;

- *IRDRecord getMarketDataRecord(String instrumentId)*
  Get market data record for the specified instrument.

- *ITMSStaticDataFacility getStaticDataSource()*
  Get a handle to the security master resource (see Static Data section below).

- *IRDRecord getCustomDataRecord(String dataSouceName, String recordId)*
  Get custom data record from the specified source for the specified record id.

- *String getClientName()*
  Returns the name of the portfolio domain.

- *String[] getCategoryNames*(*UserTicket ticket, String categoryType) throws TMSException*
  When subscribing for or retrieving position data for some categories an application may first retrieve all category names for specific category type, for example all Account names.

Stand-alone order instructions

The methods described in this section perform different operation on orders. The orderId(s) parameter that has to be passed to some of these methods can be obtained by the application in several different ways. If the application subscribes to the order flow it receives the order ids in the callback methods of the ITMSOrderEventRemoteListener interface (onOrderAdded(), onOrderUpdate()). If the application does not subscribe to the event flow or does not use it for this purpose then it should itself populate ClOrdId field of every order message it sends and then, in case it needs to call any of the modify/cancel/… methods it should construct the order id by concatenating the ClOrdId plus char '@' plus the name of the FIX connection where the order was sent. For example, if the application needs to send an order on connection "test" with ClOrdId equal to "orderN" and then needs to modify this order then it first would call sendOrder() method with field ClOrdId populated in the TMSNewOrderMessage parameter and then call modifyOrder() method passing string "orderN@test" as an orderId parameter.

All these methods return an action id that can be passed to the action event listener (IActionEventListener) and then matched to the events carrying the error/warning info for this action (See sample JAVA app).

- *long sendOrder(UserTicket userTicket,*
  TMSNewOrderMessage orderMessage) throws TMSException
  Send an order through the FIX connection with the field specified in the TMSNewOrderMessage object. The transaction destination will be one of the fields in the orderMessage.

- *long sendOrders(UserTicket userTicket,*
  TMSNewOrderMessage[] orderMessages) throws TMSException
  Send multiple orders through the FIX connection(s) with the fields for each order specified in the TMSNewOrderMessage object from the array. The transaction destination will be one of

the fields in each orderMessage.

- *long modifyOrder(UserTicket userTicket, Object orderId, TMSModifyOrderMessage fields) throws TMSException*
  Modify order with id orderId. The new fields are specified in the TMSModifyOrderMessage object.

- *long modifyOrders(UserTicket userTicket, Object[] orderIds, TMSModifyOrderMessage[] fields) throws TMSException*
  Modify multiple orders.

- *long modifyOrdersToMarketOrdType(UserTicket userTicket, Object[] orderIds) throws TMSException*
  Modify multiple orders to order type "Market".

- *long cancelOrder(UserTicket userTicket, Object orderId, TMSCancelOrderMessage orderMessage) throws TMSException*
  Cancel order with id orderId. The additional fields of the Cancel request may be specified in the TMSCancelOrderMessage object.

- *long cancelOrders(UserTicket userTicket, Object[] orderIds, TMSCancelOrderMessage[] orderMessages) throws TMSException*
  Cancel multiple orders.

- *long rejectOrder(UserTicket userTicket, TMSRejectOrderMessage orderMessage) throws TMSException*
  Reject order with id orderId. The fields of the Reject report will be specified in the TMSRejectOrderMessage object.

- *long rejectOrders(UserTicket userTicket, TMSRejectOrderMessage[] orderMessages) throws TMSException*
  Reject multiple orders.

- *long confirmOrder(UserTicket userTicket, TMSConfirmOrderMessage orderMessage) throws TMSException*
  Confirm order with id orderId. The fields of the Confirm report will be specified in the TMConfirmOrderMessage object.

- *long confirmOrders(UserTicket userTicket, TMSConfirmOrderMessage[] orderMessages) throws TMSException*
  Confirm multiple orders.

- *long fillOrder(UserTicket userTicket, TMSFillOrderMessage orderMessage) throws TMSException*
  Fill order with id orderId. The fields of the Fill report will be specified in the TMSFillOrderMessage object.

- *long fillOrders(UserTicket userTicket, TMSFillOrderMessage[] orderMessages) throws TMSException*
  Fill multiple orders.

- *long outOpenOrders(UserTicket userTicket, String orderFilter, TMSOutOnOrderMessage messageFields) throws TMSException*
  Injects a replace, a cancel or an "unsolicited cancel" for all orders matching specified filtering

expression. The replace will be injected if there is a pending modify request for the order, the cancel will be injected if there is a pending cancel request and the "unsolicited cancel" wil be injected if there are not pending replace/cancel requests.

- *long rejectOpenOrders(UserTicket userTicket, String orderFilter, TMSRejectOrderMessage messageFields) throws TMSException*
  Injects a reject for all orders matching specified filtering expression.

Portfolio instructions

In this section all methods of ITMSRemoteMarketPortfolio interface are described. The handle to ITMSRemoteMarketPortfolio is obtained either by getMarketPortfolio or addMarketPortfolio methods described above.

- *String getName()*
  Returns portfolio name.

- *int getType()*
  Returns portfolioType. One of the

  INDEX_MARKET_PORTFOLIO = 0;
  PURE_MARKET_PORTFOLIO = 1;
  LINKED_MARKET_PORTFOLIO = 2;
  CORRELATED_MARKET_PORTFOLIO = 3;

- *IRecord getPortfolioRecord()throws TMSException*
  Returns the record for thisportfolio. The fields of the record will contain portfolio summary values.

- *long modifyPortfolio(UserTicket userTicket, FieldsContainer fields) throws TMSException*
  Allows to set portfolio fields (e.g. Portfolio owner, WaveSize). This method returns an action id that can be passed to the action event listener (IActionEventListener) and then matched to the events carrying the error/warning info for this action (See sample JAVA app).

- *void postAlertMessageToOwner(UserTicket userTicket, String type, String description, String details, boolean isUrgent) throws TMSException*
  Sends an alert message to the portfolio owner's GUI. The *type* param is an arbitrary string, e.g. "Warning", the *description* param is a short alert info and the *details* param will contain more details about the alert that can be shown in the "Details" panel.

Portfolio targets instructions

- *long loadTargetsFromResource(UserTicket userTicket, String targetsResourceName, String formatResourceName, boolean removeExistingTargets) throws TMSException*
  Use this method to instruct the system to add targets to this portfolio from a file with name *targetsResourceName*. The *formatResourceName* points to the format file containing the conversion rules and the mapping for the values in the file (can be null). If *removeExistingTargets* is set to true then the current portfolio targets, if any, will be deleted, otherwise the targets from the file will be added/merged to the existing targets.

- *long addTargets(UserTicket userTicket, FieldsContainer[] fields ,boolean allowReplace) throws TMSException*
  Adds new targets to a portfolio. The fields of the targets are given in the FieldsContainer array. If allowReplace parameter is set to true then the new targets will replace the old ones

(matched based on Instrument, Side and other instructions), otherwise the target quantity of the new target will be added to the quantity of the existing target. This method returns an action id that can be passed to the action event listener (IActionEventListener) and then matched to the events carrying the error/warning info for this action (See sample JAVA app).

- *Object[] addTargetsAndGetIds(UserTicket userTicket, FieldsContainer[] fields, boolean allowReplace, MutableLong actionId) throws TMSException*
  Adds new targets to a portfolio. The fields of the targets are given in the FieldsContainer array. This method returns the ids of the added targets (in the same order as the order of target fields) that can be used by the application later to do some operations on the targets. If allowReplace parameter is set to true then the new targets will replace the old ones (matched based on Instrument, Side and other instructions), otherwise the target quantity of the new targets will be added to the quantity of the existing targetss. The actionId parameter will be populated with an action id that can be passed to the action event listener (IActionEventListener) and then matched to the events carrying the error/warning info for this action (See sample JAVA app).

- *Object addTargetAndGetId(UserTicket userTicket, FieldsContainer fields, boolean allowReplace, MutableLong actionId) throws TMSException*
  Adds new target to a portfolio. The fields of the target are given in the FieldsContainer object. This method returns the id of the added that can be used by the application later to do some operations on the targets. If allowReplace parameter is set to true then the new target will replace the old one (matched based on Instrument, Side and other instructions), otherwise the target quantity of the new target will be added to the quantity of the existing target. The actionId parameter will be populated with an action id that can be passed to the action event listener (IActionEventListener) and then matched to the events carrying the error/warning info for this action (See sample JAVA app).

- *IRecord[] addAndGetTargets(UserTicket userTicket, FieldsContainer[] fields, boolean allowReplace, MutableLong actionId) throws TMSException*
  Adds new target to a portfolio. The fields of the target are given in the FieldsContainer object. This method returns the record representing the added target. If allowReplace parameter is set to true then the new target will replace the old one (matched based on Instrument, Side and other instructions), otherwise the target quantity of the new target will be added to the quantity of the existing target. The actionId parameter will be populated with an action id that can be passed to the action event listener (IActionEventListener) and then matched to the events carrying the error/warning info for this action (See sample JAVA app).

- *IRecord addAndGetTarget(UserTicket userTicket, FieldsContainer fields, boolean allowReplace, MutableLong actionId) throws TMSException*
  Adds new target to a portfolio. The fields of the target are given in the FieldsContainer object. This method returns the record representing the added target. If allowReplace parameter is set to true then the new target will replace the old one (matched based on Instrument, Side and other instructions), otherwise the target quantity of the new target will be added to the quantity of the existing target. The actionId parameter will be populated with an action id that can be passed to the action event listener (IActionEventListener) and then matched to the events carrying the error/warning info for this action (See sample JAVA app).

- *IRecord[] getTargets(UserTicket userTicket, Object[] targetIds) throws TMSException*
  This method can be used to synchronously obtain target records with the specified ids.

- *IRecord getTarget(UserTicket userTicket, Object targetId) throws TMSException*
  This method can be used to synchronously obtain target record with the specified id.

- *IRecord[] getTargets(UserTicket userTicket, String filterExpression) throws TMSException*
  This method can be used to synchronously obtain the records for all targets of the portfolio. The filterExpression can be used to filter out targets; in case it is null all targets will be returned.

- *void modifyTargets(UserTicket userTicket, Object[] targetIds,*
  FieldsContainer[] fields) throws TMSException
  Modifies targets with given ids. The new targets' fields are given in the FieldsContainer array.

- *void removeTargets(UserTicket userTicket, Object[] targetIds) throws TMSException*
  Removes targets with given ids from the portfolio

- *void removeAllTargets(UserTicket userTicket) throws TMSException*
  Removes all targets from the portfolio

Portfolio order instructions

- *void sendOrder(UserTicket userTicket, Object targetId,*
  TMSNewOrderMessage orderMessage) throws TMSException
  Send an order through the FIX connection with the field specified in the TMSNewOrderMessage object. The remaining fields will be set in the order from the target with id targetId. Basically, the target has a role of a template from which the orders are sliced. Target's released/unreleased qty will change.

- *void sendOrders(UserTicket userTicket, Object[] targetIds,*
  TMSNewOrderMessage[] orderMessages) throws TMSException
  Same as sendOrder, only sends multiple.

- *void modifyOrdersToMarketOrdType(UserTicket userTicket,*
  String waveId) throws TMSException
  This method is used to quickly change all orders from the same wave to market order type (as opposed to calling *modifyOrder()* methods of ITMSRemoteClient). See section "Portfolio wave instructions" on how the waves can be generated.

- *void cancelOpenOrders(UserTicket userTicket, Object targetId) throws TMSException*
  Cancel open orders released from the target with id targetId

- *void cancelOpenOrders(UserTicket userTicket) throws TMSException*
  Cancel all open orders released from the targets of this portfolio

Portfolio wave instructions

- *void startWave(UserTicket userTicket) throws TMSException*
  Increases the wave marker. After this call any orders released from the portfolio targets will have new wave id

- *void sendWave(UserTicket userTicket, Object[] targetIds,*
  TMSNewOrderMessage[] orderMessages,
  FieldsContainer targetFields) throws TMSException
  Starts a new wave and sends orders as part of that wave. Each targetId from the targetIds array has a corresponding orderMessage from orderMessages array. Also the targetFields array contains the field values that will be set in each target prior to releasing the wave.

- *void sendWave(UserTicket userTicket) throws TMSException*
  Starts new wave and sends an order for all targets of the portfolio based on field values and instructions in the targets (should be correctly set prior to issuing this call)

- *void cancelWave(UserTicket userTicket, String waveId) throws TMSException*
  Cancel all orders with given wave id

Portfolio AutoTrader instructions

- *void createAutoTrader(UserTicket userTicket,*
  ITMSAutoTraderSpec autoTraderSpec) throws TMSException
  Create new AutoTrader module attached to this portfolio. See section "Interface ITMSAutoTraderSpec" for more details.

- *ITMSMutableAutoTraderSpec getAutoTraderSpec(UserTicket userTicket) throws TMSException*
  Return the spec of the AutoTrader attached to this portfolio

- *void modifyAutoTrader(UserTicket userTicket,*
  ITMSAutoTraderSpec autoTraderSpec) throws TMSException
  Modify the spec of the AutoTrader attached to this portfolio

- *void pauseAutoTrader(UserTicket userTicket) throws TMSException*
  Pause AutoTrader attached to this portfolio

- *void resumeAutoTrader(UserTicket userTicket) throws TMSException*
  Resume AutoTrader attached to this portfolio

- *void terminateAutoTrader(UserTicket userTicket) throws TMSException*
  Terminate AutoTrader attached to this portfolio

- *void pauseTargets(UserTicket userTicket, Object[] targetIds) throws TMSException*
  Pause individual portfolio targets. The paused targets will not be operated upon by the AutoTrader module. They can still be traded manually

- *void resumerTargets(UserTicket userTicket,*
  Object[] targetIds) throws TMSException
  Resume individual portfolio targets.

- *void terminateTargets(UserTicket userTicket,*
  Object[] targetIds) throws TMSException
  Terminate individual portfolio targets. Terminated targets cannot be traded by either AutoTrader module or manually and cannot be restored for further trading.

- *void suspendTargets(UserTicket userTicket,*
  Object[] targetIds) throws TMSException
  Suspend individual portfolio targets. Suspend targets cannot be traded by either AutoTrader module or. Use resumeTargets() to restore targets from the suspended state.

- *void executeAutoTraderAction(UserTicket userTicket,*
  Configurator params) throws TMSException
  This method is called when custom instruction has to be given to the AutoTrader. This call causes the AutoTrader's method onAutoTraderAction() to be called. The code implementing this method in the AutoTrader will know how to interpret parameters set in the params Configurator object.

FIX Engine Instructions

If an application needs to send/post FIX messages through TMS it can use TMS' engine gateway API. The handle to the engine gateway is obtained through ITMSRemoteClient's method getEngineGateway() which returns ITMSEngineGateway. The following methods of ITMSEngineGateway may be called:

- *boolean isConnectedTo(String transactionDestination)*
- *void sendMessage(String transactionDestination, ELTMessage m) throws Exception*
- *void sendMessage(String transactionDestination, ELTMessage m, boolean forceSend) throws Exception*
- *void sendMessage(String transactionDestination, ELTFieldGroup fields) throws Exception*
- *void sendMessage(String transactionDestination, ELTFieldGroup fields, boolean forceSend) throws Exception*
- *void sendMessage(String transactionDestination, String messageString) throws Exception*
- *void sendMessage(String transactionDestination, String messageString, boolean forceSend) throws Exception*
- *void sendMultipleMessages(String transactionDestination, ELTFieldGroup[] messages) throws Exception*
- *void sendMultipleMessages(String transactionDestination, ELTFieldGroup[] messages, boolean forceSend) throws Exception*
- *void postInternalMessage(String transactionDestination, ELTAppMessage m) throws Exception*
- *void postInternalMessage(String transactionDestination, ELTFieldGroup fields) throws Exception*
- *void postInternalMessage(String transactionDestination, String messageString) throws Exception*
- *void postMultipleInternalMessages(String transactionDestination, ELTFieldGroup[] messages) throws Exception*
- *ELTEngineProxy getEngineProxy()*

Alert Instructions

It is possible for an external application to post an alert into TMS so that it appears in users' GUIs.

- *void postAlertMessageToUsers(UserTicket userTicket, String[] userList, String type, String description, String details, boolean isUrgent)*
  This method can post errors/warnings/info into the AlertsConsole of all specified users.

- *postAlertMessageToUserGroups(UserTicket userTicket, String[] userGroupList, String type, String description, String details, boolean isUrgent)*
  This method can post errors/warnings/info into the AlertsConsole of users that belong to specified groups.

Timer Instructions

It is possible for an external application to instruct the TMS to schedule a timer and be notified by the TMS when a timer triggers. The added value of doing it through TMS as opposed to standard JAVA timers is that the TMS API provides more convenient methods and also time period randomization.

- *long scheduleNonRepeatingTimer(Date time, ITMSTimerHandler handler)*
  This method schedules a timer that will fire at the specified time and call the handler's onTimer(timerId) method. Returns timer id. When onTimer() method of the application's timer handling module is called the timerId is passed so that the application knows which timer fired. This allows application to register the same timer handler multiple times.

- *long scheduleNonRepeatingTimer(int delayInSeconds, ITMSTimerHandler handler)*
  This method schedules a timer that will fire after specified number of seconds and call the handler's onTimer(timerId) method.. Returns timer id.

- *long scheduleRepeatingTimer(int delayInSeconds, int periodInSeconds, ITMSTimerHandler handler)*
  This method schedules a timer that will fire after specified number of seconds and continue firing repeatedly each *periodInSeconds*. Returns timer id.

- *long scheduleRepeatingTimer(int delayInSeconds, int periodInSeconds, int deviationInSeconds, ITMSTimerHandler handler)*
  This method schedules a timer that will fire after specified number of seconds and continue firing repeatedly with a certain interval with the interval being a random number uniformly distributed between (*periodInSeconds – deviationInSeconds)* and (*periodInSeconds + deviationInSeconds)*. Returns timer id.

- *void cancelTimer(long timerId)*
  Cancels the timer with the specified id.

Accessing Static Data

Call method *getStaticDataSource()* of the ITMSRemoteClient interface to get a handle to *ITMSStaticDataAccessor* object. Then use methods of *ITMSStaticDataAccessor* to get the necessary static data.

Accessing Market Data

Call method getMarketDataRecord*(String instrId)* of the ITMSRemoteClient interface to get market data record for the given instruments. Then the fields of the record may be examined.

Entity Objects and Interfaces

The methods of the ITMSRemoteClient and ITMSRemoteMarketPortfolio API often return objects to the caller or require objects to be passed as parameters. The interfaces to these objects that represent orders/targets/portfolios/messages in the TMS are described below.

Interface IRecord

- *double getNumericFieldValue(String fieldId)*
- *String getStringFieldValue(String fieldId)*
- *char getCharFieldValue(String fieldId)*
- *boolean isFieldNumeric(String fieldId)*
- *boolean getBooleanFieldValue(String fieldId)*
- *long getTimeFieldValue(String fieldId)*

TMSNewOrderMessage

Objects of this type are passed to methods sending/posting new individual orders/slices through TMS. This object extends ELTFieldGroup and the ETLTFieldGroup API should be used to set any of the required FIX fields. The field names and tags can be looked up in the configuration file *Directory/ElTrader/FIXTMSMetaData/FIXFieldsSuperset.dtd* (see "INFOREACH FIX Java Programmers Guide.doc" section 2.2.1 for ELTFieldGroup API). The constants from ITMSConstants file starting with MSGFLDTAG_ prefix may be used in place of integer tags. The TMSNewOrderMessage also extends TMSBaseMessage and the TMSBaseMessage's method *setCommandFlag(int flag)* can be used to instruct the TMS to treat the new order message in several different ways:

- if *setCommandFlag(ITMSContanst.COMMANDFLAG_POST_MESSAGE*) is called then the message will be "posted" into the TMS rather than "sent". The difference between posting and sending is that in case of posting all the same processing of the message is done except that the message is not physically sent to the counterparty.

- if *setCommandFlag(ITMSContanst.COMMANDFLAG_IGNORE_ROUND_TO_LOT*) is called then the message will be sliced from the target and it is necessary to instruct the system to ignore round-to-lot instructions set in the target. In case of non-slice orders the round to lot flag has no effect as there is no rounding don anyway.

- if *setCommandFlag(ITMSContanst.COMMANDFLAG_DONE_AWAY*) is called then the message will posted into the system together with the artificial confirm and the Fill message with the fill quantity equal to the order quantity set in the message and the fill price equal to the limit price set in the message. Used for "tallying" market orders.

TMSModifyOrderMessage

Same as TMSNewOrderMessage only is used to modify existing order. In the simplest case will only contain fields that have to be modified. Additional fields may be set to be sent in the modify request. Supports *setCommandFlag(ITMSContanst.COMMANDFLAG_POST_MESSAGE*) call.

TMSCancelOrderMessage

Same as TMSNewOrderMessage only is used to cancel existing order. Additional fields may be set to be sent in the cancel request. Supports *setCommandFlag(ITMSContanst.COMMANDFLAG_POST_MESSAGE*) call.

TMSOutOnOrderMessage

Same as TMSNewOrderMessage. Is used when "injecting" Cancel report as though it came from the market.

TMSRejectOrderMessage

Same as TMSNewOrderMessage. Is used when "injecting" Reject report as though it came from the market.

Interface ITMSAutoTraderSpec

All AutoTrader specs are usually stored as XML files that can be loaded and converted to object of type ITMSAutoTrader spec by calling static method TMSAutoTraderSpec.valueOf(xmlSpec) (See sample JAVA app for more details).

ELTFieldGroup

See "INFOREACH FIX Java Programmers Guide.doc" section 2.2.1

FieldsContainer

Objects of this type are passed to methods adding/modifying portfolios/targets in TMS. This object's API should be used to set any of the required portfolio/target fields. The field names and tags can be looked up in the configuration file *Directory/ElTrader/TMS/Config?ElementSettings/TableElement/PortfolioTableFieldsMetadata.xml*. The constants from ITMSConstants file starting with PFFLDNAME_ prefix may be used in place of field ids.

- *setFieldValue(Object fieldId, String value)*
- *void setFieldValue(Object fieldId, double value)*
- *void setFieldValue(Object fieldId, boolean value)*
- *void setFieldValue(Object fieldId, char value)*
- *void setFieldValue(Object fieldId, long value)*
- *void clear()*
- *boolean containsStringField(Object fieldId)*
- *boolean containsNumericField(Object fieldId)*
- *String getStringFieldValue(Object fieldId)*
- *double getNumericFieldValue(Object fieldId)*
- *Iterator getStringFields()*
- *Iterator getNumericFields()*

Using ITMSRemoteClient API for report data monitoring

Requesting report data snapshots

The user's application can request data from the TMS reports residing on the server in a form of a snapshot. Once the snapshot request is made the application's listener component will receive report data for all records/nodes of the report corresponding to the request's query. The returned snapshot may come in multiple events. The first and last events will be marked with special flags.

The following methods of ITMSRemoteClient can be used to request data from any report including any user-defined report.

- *void requestTableReportData(UserTicket userTicket, String domainManagerName, String reportName, String filter, IRPTReconnectableEventListener listener) throws RPTException*

  The client application calls this method whenever it needs to obtain information about records in the table report. The first two parameters identify the report by name and by the name of the domain where this report was instantiated. By default, there is only one domain manager for reports in the TMS: "Portfolio report manager". Depending on the TMS configuration more domain managers may be present. The filtering criteria may be passed to this method in case only certain records from the report are needed. In the example below information is requested for all market targets for instrument 'IBM':

```
requestTableReportData(GlobalSystem.getUserTicket(),
            "Portfolio report manager",
            "Market Targets",
            "\"Instrument\" = 'IBM'",
            listener);
```

- *void requestTableReportDataAsXML(UserTicket userTicket, String domainManagerName, String reportName, String filter, Writer writer) throws RPTException*

  Same as above only instead of events it writes the report data as XML string into the Writer stream passed to this method.

- *void requestTreeReportData(UserTicket userTicket, String domainManagerName, String reportName, String level, String filter, IRPTReconnectableEventListener listener) throws RPTException*

  The client application calls this method whenever it needs to obtain information about nodes in the tree report. The first two parameters identify the report by name and by the name of the domain where this report was instantiated. By default, there is only one domain manager for reports in the TMS: "Portfolio report manager". Depending on the TMS configuration more domain managers may be present. The level parameter specifies the level in the tree whose nodes are requested. The filtering criteria may be passed to this method in case only certain nodes from the report level are needed. In the example below information is requested for all order transaction nodes for instrument 'IBM':

```
requestTreeReportData(GlobalSystem.getUserTicket(),
            "Portfolio report manager",
            "Single Order System Report",
            "OrdTrnId",
            "\"Instrument\" = 'IBM'",
```

```
            listener);
```

- *void requestTreeReportDataAsXML(UserTicket userTicket, String domainManagerName, String reportName, String filter, Writer writer) throws RPTException*

  Same as above only instead of events it writes the report data as XML string into the Writer stream passed to this method.

The requested data may come in multiple *RPTStateEvent* events. The first and last event will be specially marked. Each event will contain multiple records (one per target) and each record will contain the field values that can be examined. See Listing 1 for an example of the *processEvent(RPTEvent event)* implementation from IRPTReconnectableEventListener interface.

```
public void processEvent(RPTEvent event)
{
   if (event instanceof RPTRecordEvent)
   {
        processRecordEvent((RPTRecordEvent)event));
   }
   else
   if (event instanceof RPTStateEvent)
   {
       processStateEvent((RPTStateEvent)event);
   }
   else
   {
       System.out.println("(" + name_ + ") event ---> " + event);
   }
}

public void processStateEvent(RPTStateEvent event)
{
    if (stateEvent.isLast())
    {
        // On request state, we will know when we have all the
        // data when we receive the state event where isLast() is
        // true!
        System.out.println("(" + name_ + ") last stateEvent (" +
                        stateEvent.getEventCount() + ") ---> " + event);
    }
    else
    {
         System.out.println("(" + name_ + ") stateEvent (" +
                     stateEvent.getEventCount() + ") ---> " + event);
    }

    for (int i=0; i < stateEvent.getEventCount(); i++)
         processRecordEvent((RPTRecordEvent)stateEvent.getEventAt(i));
}

public void processRecordEvent(RPTRecordEvent event)
{
     // extract record from the record event
    IRPTRecord rec = (recEvent).getNewRecord();

    StringBuffer buf = new StringBuffer();
    buf.append("(");
    buf.append(name_);
    buf.append(") processing record: ");

    // examine the action:
    switch (recEvent.getAction())
    {
```

```
            case RPTRecordEvent.EVENT_RECORD_CHANGED :
                 buf.append("changed");
                 break;
            case RPTRecordEvent.EVENT_RECORD_INSERTED :
                 buf.append("inserted");
                 break;
            case RPTRecordEvent.EVENT_RECORD_REMOVED :
                 buf.append("removed");
                 break;
            case RPTRecordEvent.EVENT_RECORD_UPDATED :
                 buf.append("updated");
        }

        buf.append(" ID=");
        buf.append(rec.getIdentifier());

        // Extract information from the record!
        int fieldCount = rec.getRecordContext().getNumericFieldCount() +
                         rec.getRecordContext().getStringFieldCount();
        for (int i = 0; i < fieldCount; i++)
        {
            buf.append(", ");
            buf.append(rec.getRecordContext().getFieldIdentifierAt(i));
            buf.append("=");
            if (rec.getRecordContext().isFieldNumericAt(i))
                buf.append(rec.getNumericFieldValueAt(i));
            else
                buf.append(rec.getStringFieldValueAt(i));
        }

        System.out.println(buf);
    }
```

Listing 1.

Subscribing for report data flow

The user's application can subscribe for the data flow from the report residing on the server. When the subscribe call is made the data currently existent in the report will be sent to the calling component in a form of a snapshot (in multiple state events) and then the record events will continuously arrive from the report until unsubscribe call is made. The following methods of ITMSRemoteClient can be used to subscribe for data from any report including any user-defined report.

- *void subscribeForTableReportData(UserTicket userTicket, String domainManagerName, String reportName, String filter, IRPTReconnectableEventListener listener) throws RPTException*

  The client application calls this method whenever it needs to obtain information about records in the table report. The first two parameters identify the report by name and by the name of the domain where this report was instantiated. By default, there is only one domain manager for reports in the TMS: "Portfolio report manager". Depending on the TMS configuration more domain managers may be present. The filtering criteria may be passed to this method in case only certain records from the report are needed. In the example below application subscribes to the event for all market targets for instrument 'IBM':

```
subscribeForTableReportData(GlobalSystem.getUserTicket(),
            "Portfolio report manager",
            "Market Targets",
```

```
            "\"Instrument\" = 'IBM'",
            listener);
```

- *void subscribeForTreeReportData(UserTicket userTicket, String domainManagerName, String reportName, String level, String filter, IRPTReconnectableEventListener listener) throws RPTException*

  The client application calls this method whenever it needs to obtain information about nodes in the tree report. The first two parameters identify the report by name and by the name of the domain where this report was instantiated. By default, there is only one domain manager for reports in the TMS: "Portfolio report manager". Depending on the TMS configuration more domain managers may be present. The level parameter specifies the level in the tree whose nodes are requested. The filtering criteria may be passed to this method in case only certain nodes from the report level are needed. In the example below application subscribes to the event for all order transaction nodes for instrument 'IBM':

```
subscribeForTreeReportData(GlobalSystem.getUserTicket(),
            "Portfolio report manager",
            "Single Order System Report",
            "OrdTrnId",
            "\"Instrument\" = 'IBM'",
            listener);
```

After the initial state is sent to the subscribing component the record events corresponding to the changes in the report will be sent continuously to the subscribing component until an unsubscribe call is made. Each record event will contain information on what exactly happened to the report's nodes and the nodes' fields values.
There are three possible changes that can be reflected in the record event:

RPTRecordEvent.EVENT_RECORD_CHANGED --- sent when fields of the node change

RPTRecordEvent.EVENT_RECORD_INSERTED --- sent when new node is added

RPTRecordEvent.EVENT_RECORD_REMOVED --- sent when node is removed

The change indicator is the record event through the *getAction()* call. See Listing 1 for an example of the *processEvent(RPTEvent event)* implementation. Appendix B lists the default set of fields that can be used for specifying the request criteria. Users can expand this list by developing their own fields and placing them in the reports.

Examining received record events

The record events arriving to the listening component contain records of values that can be accessed and examined. To find out what report the event came from use *((IRPTReport)event.getSource()).getName().* The record object is obtained form the event by calling getNewRecord() method. Once the record is obtained the fields of the record are retrieved by calling either getNumericFieldValueAt(int index) or getStringFieldValueAt(int index) methods. The type of the field dictates which method should be called. Appendix B contains description of fields that can be received inside the record event and their type and 'representation'. For fields with NUMERIC representation the getNumericFieldValueAt method should be called and for fields with STRING representation the getStringFieldValueAt method should be called. The index of the field in the record that can be passed to the get…FieldValueAt() methods is obtained from the *record context* by supplying the field's name to the record context's getFieldIndex() method. For example, to retrieve value of the Instrument field from the record the following sequence of calls should be made:

```
IRPTRecord rec = recEvent.getNewRecord();
IRPTRecordContext recContext = rec.getRecordContext();

 fieldIndex = recContext.getFieldIndex("Instrument");

if (fieldIndex >= 0)
          String instr = rec.getStringFieldValueAt(fieldIndex));
```

Even when representation of the field is not known it is still possible to call the right get…() method if one extra call is made to discover the representation of a given field:

```
 IRPTRecord rec = recEvent.getNewRecord();
IRPTRecordContext recContext = rec.getRecordContext();

 fieldIndex = recContext.getFieldIndex("fieldName");

if (fieldIndex >= 0)
{
      if (recContext.isFieldNumericAt(fieldIndex))
            double d = rec.getNumericFieldValueAt(fieldIndex));
      else
            String s = rec.getStringFieldValueAt(fieldIndex));
}
```

It is important to note that multiple field types have NUMERIC representation and are accessed via getNumericFieldValueAt() method. Types *double, int, long, float, Date, UTCTimestamp, Boolean, char* all have Numeric representation and therefore require extra manipulation of the number returned by getNumericFieldValueAt() method in order to get a final value of the right type. The samples below show how the value returned by getNumericFieldValueAt() method has to be handled in order to obtain result of the right type:

```
// for chars:
char c = (char) getNumericFieldValueAt(index);

// for UTCTimestamp:
Date d = new Date((long) getNumericFieldValueAt(index));

// for Date (will have hour  = 0, min = 0, sec = 0):
Date d = new Date((long) getNumericFieldValueAt(index));

// for Boolean (FIX Boolean is actually a char with values 'Y' or 'N'):
char b = (char) getNumericFieldValueAt(index);

// for int:
int i = (int) getNumericFieldValueAt(index);

// for long:
long l = (long) getNumericFieldValueAt(index);

// for float:
float f = (float) getNumericFieldValueAt(index);
```

File [inforeach_home]/backend/Directory/ElTrader/TMS/Config/ElementSettings/TableElement/PortfolioTableFieldsMetaData.xml contains definition of all TMS fields for portfolio targets or portfolio summary

records. When examining target/portfolio events received in the external applications or inside the analytics the exact field names and their types may be looked up in this file.

File [inforeach_home]/backend/Directory/ElTrader/TMS/Config/MarketData/MarketQuoteFieldsMetaData.xml contains definition of all TMS fields for market data records. When examining market data events received in the external applications or inside the analytics the exact field names and their types may be looked up in this file.

File [inforeach_home]/backend/Directory/ElTrader/TMS/Config/ElementSettings/TreeElement/SingleOrderTreeFieldsMetaData.xml contains definition of all TMS fields for order nodes. When examining order events received in the external applications or inside the analytics the exact field names and their types may be looked up in this file.

Unsubscribing from data flow

The following method takes as a parameter the listener object that was subscribed to the report prevoiously and unsubscribes it.

- *void unsubscribeFromReportData(UserTicket userTicket,*
  String domainManagerName,
  String reportName,
  IRPTDisconnectionListener listener)
  throws RPTException

Creating report

It is possible to create completely new report on the server through by calling this method

- *IRPTReport createReport(UserTicket userTicket, String reportSpecResource) throws RPTException*
  The reportSpecResource parameter is a name of the file residing in Directory containing XML-formatted description of the report.

- *IRPTReport createReport(UserTicket userTicket, XMLData reportSpec) throws RPTException*
  The reportSpec parameter is an object representing XML structure of the report. It can be created by calling a constructor that takes report file name as a parameter.

Removing report

It is possible to remove a report from the server:

- IRPTReport createReport(UserTicket userTicket,
  String domainManagerName,
  String reportName) throws RPTException

Checking for report

It is possible to check whether the report is accessible by the external application:

- *boolean isReportAvailable (String domainManagerName,*
  String reportName) throws RPTException

Developing Custom Algorithms

Overview

In TMS, the tradable entities are represented as *portfolio targets*. A target is a collection of fields that define the tradable entity. For example, total 100000 shares of IBM common stock to be traded as limit orders on the NYSE is considered a target. Targets are combined into portfolios and then the portfolios can be traded manually (by simple slicing) or automatically by associating an instance of an analytic (***AutoTrader ©***) with the portfolio and giving it control over order generation and order life cycle. In the simplest case, targets will have final target quantity and side defined from the beginning and the analytic will stop trading an instrument once the target quantity is reached. However, InfoReach TMS also supports targets that have nothing but instrument id defined for them in the beginning and then it is left to the analytic itself to decide on the number and the parameters of the orders generated for each target.

AutoTrader instances will subscribe for certain events such as order, target and market data changes and then make a decision on generating/modifying/canceling an order or adding/removing/modifying portfolio targets or even controlling its own activity cycle.

InfoReach TMS provides the mechanism for plugging-in of custom analytics for order generation and management. The Java API exposed to users for developing such algorithms has, on one hand, methods for subscribing for different types of events generated by the TMS or requesting information from TMS synchronously and, on the other hand, methods for instructing the system to perform an action based on the analytic module's calculations. The event flow available to the analytic modules includes market data ticks, changes to the orders' statuses, or changes to the field values of any live report. In addition, any external event data source may be plugged into the system and subscribed to by the analytic module. The instructions that can be given to the system include instructions to generate new orders, to modify or cancel outstanding orders, to change the set of the portfolio targets managed by the module, to change the module's parameters at runtime and much more.

All custom analytic modules will extend class
com.inforeach.eltrader.tms.domain.portfolio.autotrader.modules.TMSAbstractAnalyticModule.

The implementer of the module has to provide implementation for the onAutoTraderCreated() and onAutoTraderRecovered() methods where all the necessary subscribe…() and schedule…() methods will be called. Once the module is subscribed to the required event flow (e.g. market data ticks) and scheduled the timers to be triggered at certain intervals the TMS then will call one of the module's *on…()* methods whenever the corresponding event occurs. The custom module's implementation of a particular *on...()* method will perform event processing and then turn around and instruct the TMS to do some action. The method ***getAutoTrader()*** of the TMSAbstractAnalyticModule will return ***ITMSAutoTrader*** interface through which the module will be able to request data from the TMS and issue instructions to the TMS. As a result of this instruction the changes in TMS will occur, however, no new triggering events will be generated to avoid a feedback loop. All the information that can be required by the analytic module about the new state of the TMS will be returned synchronously to the module in the result of the call.

The methods of the ***ITMSAutoTrader*** interface are grouped into the categories described below. Some of these methods return or take as parameters entity objects representing orders, targets, etc. See Entity Objects section for detailed description of these objects' interfaces.

Guidelines for implementing algorithms

INFOREACH TMS is a client-server application where trade execution management, reports calculation, as well hosting of algorithms (AutoTraders) takes place on the server side. Therefore, when you create custom AutoTraders it is essential to keep in mind the following.

- Setup development environment for coding your custom AutoTrader
- Create and compile your AutoTrader
- Create GUI panels for traders to parameterize AutoTrader
- Configure TMS to incorporate your AutoTrader
- Run TMS server
- Run TMS front-end application and invoke your AutoTrader.

Creating development environment

Copy all TMS libraries from [TMS HOME]/common/lib to your local drive and include them into your CLASSPATH. Folder [TMS HOME]/backend/samples/java/SampleAnalytic contains sample analytic code and project setup for easy compilation.

Creating AutoTrader

Use APIs documented in this section of the manual, as well as provided samples (available in [TMS HOME]/backend/samples/java/SampleAnalytic) to code your custom algorithm.

Creating AutoTrader GUI

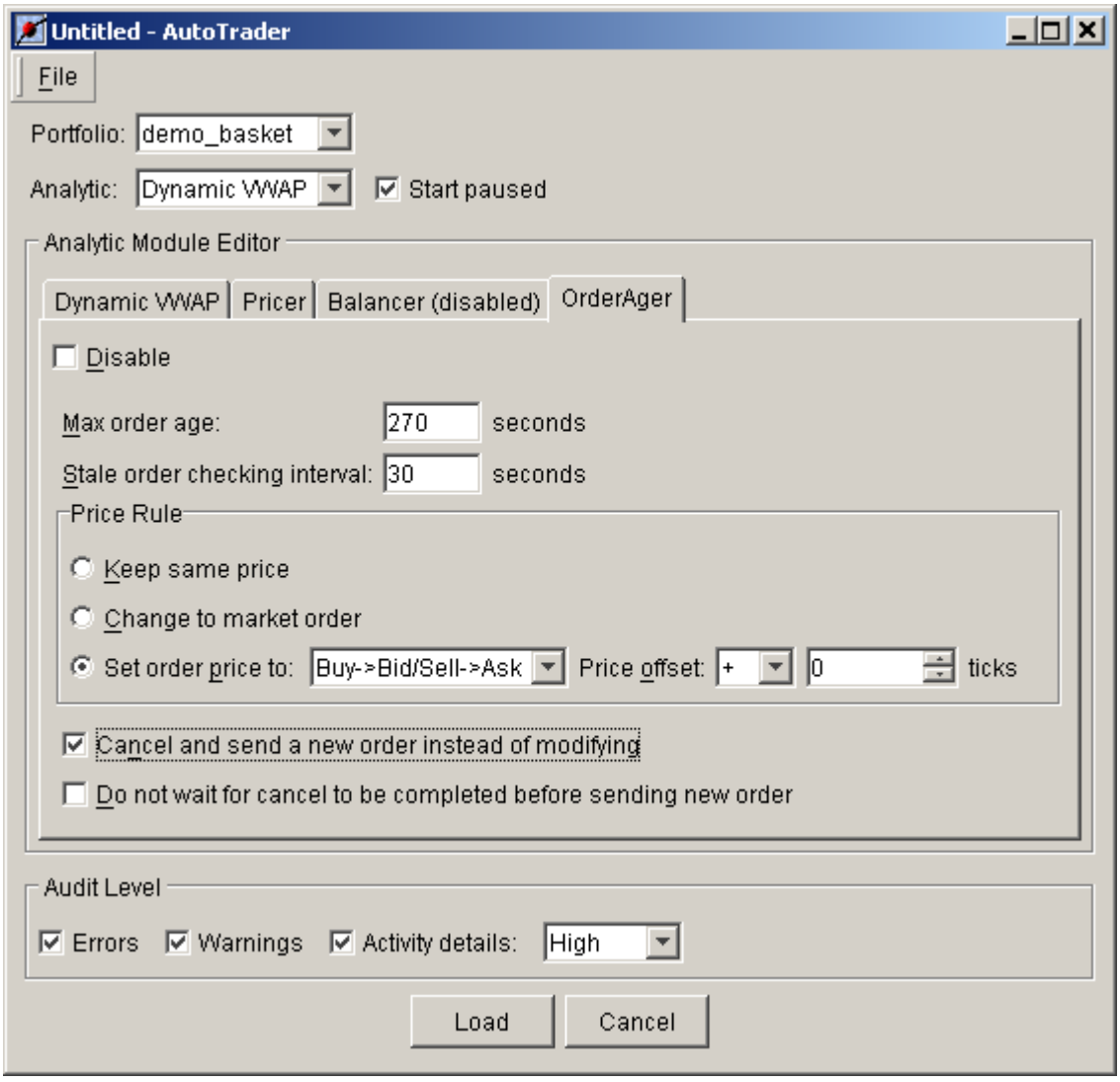
TMS AutoTraders are invoked and parameterized by traders from TMS GUI via AutoTrader panel (unless configured to start automatically). AutoTrader panel allows traders to select AutoTrader, portfolio for AutoTrader and set parameters common for all targets that form given portfolio.

Using SWING (JFC)

This approach could be selected if developer is familiar with developing of Java GUI and wishes to provide "cleaner" custom GUI interface to manage parameters of the AutoTrader.

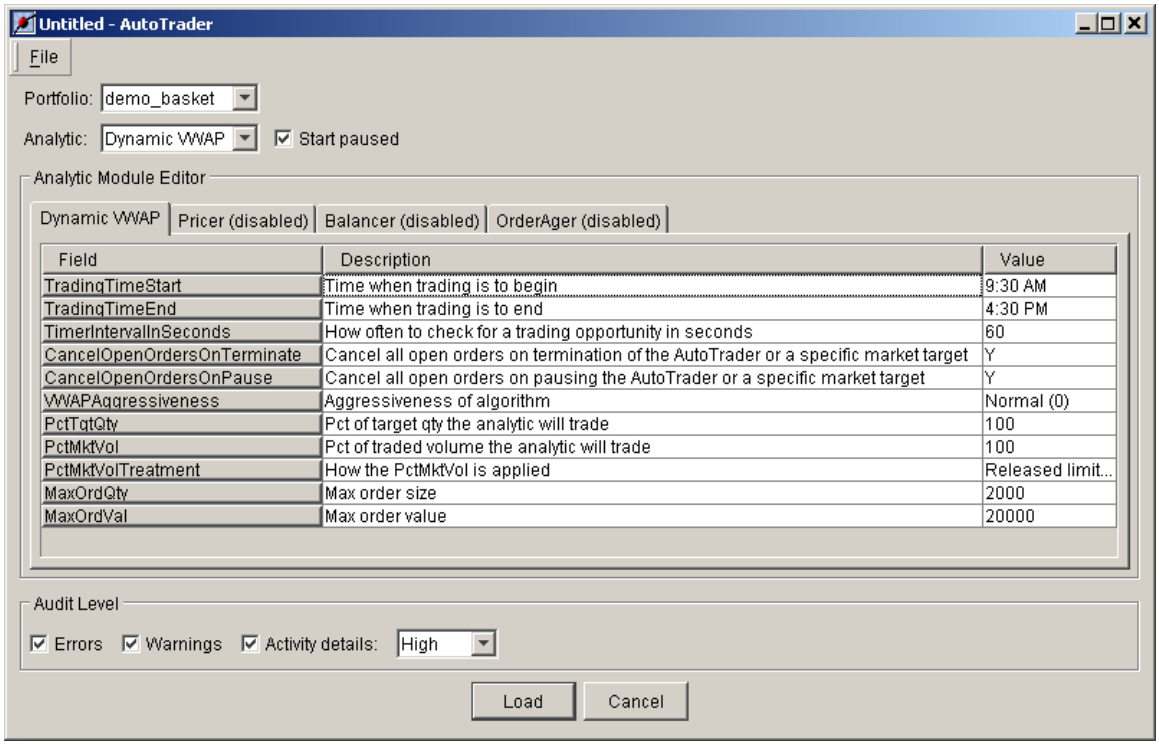
Please keep in mind that when AutoTrader implementation is modified to the extend that it is required to make adjustments to the set of parameters you might have to make changes to AutoTrader panel and recompile it along with AutoTrader.

Below is the snapshot of the AutoTrader panel tab implemented with SWING.



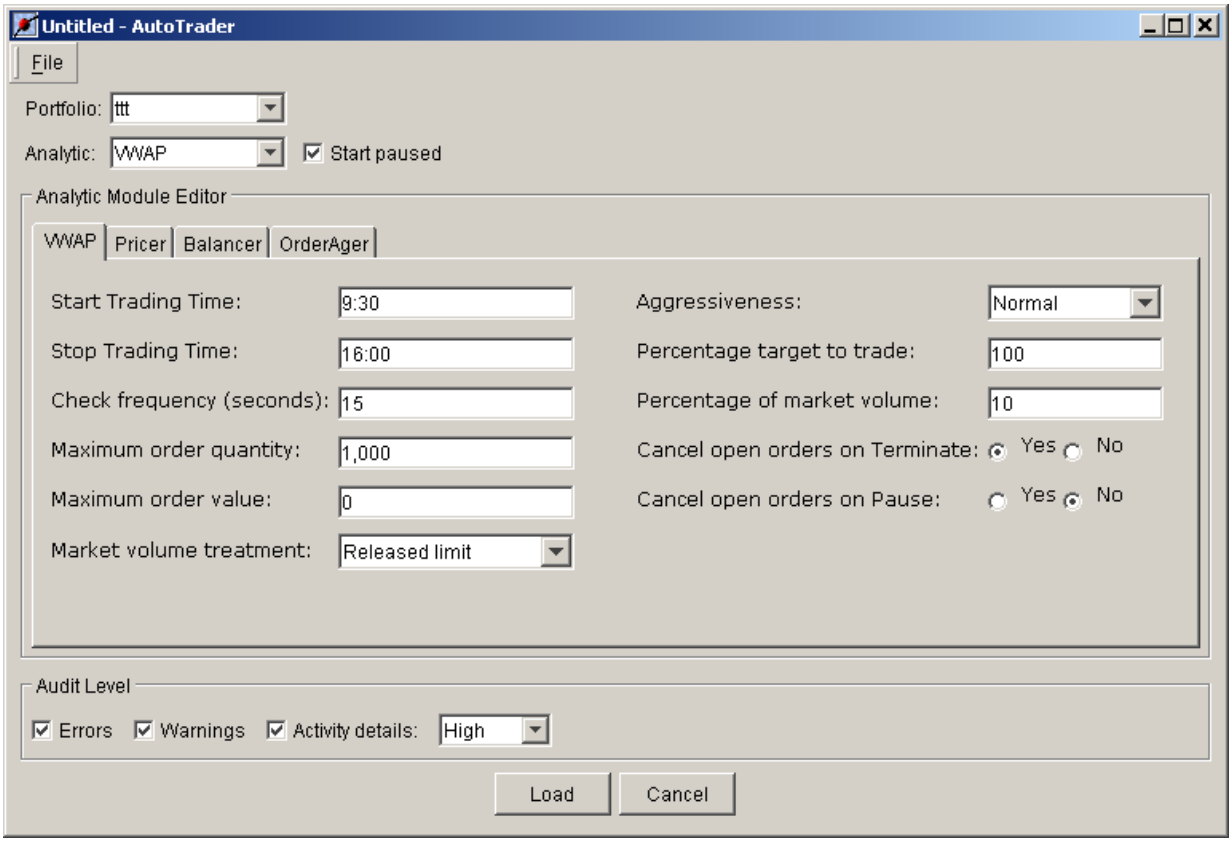
Using generic AutoTrader panel

Generic AutoTrader panel (snapshot below) is provided by TMS to facilitate quick creation of AutoTrader panel without coding and compiling. Instead you can configure the list of AutoTrader required parameters along with the lists of valid values.



Using HTML

AutoTrader panel can be also automatically generated by TMS from the HTML file. You can use third party HTML editor tools to create and modify customized layouts of AutoTrader panel.



Configuring TMS to run custom algorithms

All custom analytic modules have to be registered with the TMS. To register the module one has enter an XML description of the module's implementation class and module's parameters in the special configuration file *[TMS_HOME]/Directory/ElTrader/TMS/Config/AnalyticModuleMetaData.xml.* Here is an example of the XML element registering the new custom module:

```
<AnalyticModuleMetaData
    type = "SampleModule"

    analyticModuleClassName = "SampleAnalytic"
    metaDataImplClassName =
         "com.inforeach.eltrader.tms.domain.portfolio.analytic.modules.TMSFieldListAnalyticModuleMetaData"
    specClassName =
         "com.inforeach.eltrader.tms.domain.portfolio.autotrader.modules.TMSFieldListAnalyticModuleSpec"
    editorClassName =
         "com.inforeach.eltrader.tms.view.portfolio.market.autotrd.modules.TMSVFieldListModuleEditor"

    classPath = "../../samples/java/SampleAnalytic"
>
   <FieldList>
     <Field
          name = "Timer1Interval"
          description = "Interval (in sec.) between evaluations for position entry"
          type = "integer"
     />
     <Field
          name = "Timer2Interval"
          description = "Interval (in sec.) between evaluations for position exit"
          type = "integer"
     />
   </FieldList>
```

```
</AnalyticModuleMetaData>
```

Description of the Analytic Module entry attributes:

- *type*
  Name of the module

- *isExposedToUsers*
  If true, the users will be able to instantiate the AutoTrader with this module from the GUI

- *analyticModuleClassName*
  The name of the class implementing the module (should extend ***com.inforeach.eltrader.tms.domain.portfolio.autotrader.modules.TMSAbstractAnalyticModule*****)**

- *metaDataImplClassName*
  In the simplest case is equal to ***com.inforeach.eltrader.tms.domain.portfolio.autotrader.modules.TMSFieldListAnalyticModuleMetaData.*** Can be overwritten in advanced cases. For example, if custom HTML form is used then this parameter should be equal to **com.inforeach.eltrader.tms.domain.portfolio.autotrader.modules.TMSHtmlFormAnalyticModuleMetaData**

- *specClassName*
  In the simplest case is equal to ***com.inforeach.eltrader.tms.domain.portfolio.autotrader.modules.TMSFieldListAnalyticModuleSpec.***
  Can be overwritten in advanced cases (but is the same for if custom HTML is used).

- *editorClassName*
  In the simplest case is equal to ***com.inforeach.eltrader.tms.view.portfolio.market.autotrd.modules.TMSVFieldListModuleEditor***
  Can be overwritten in advanced cases. For example, if custom HTML form is used then this parameter should be equal to **com.inforeach.eltrader.tms.view.portfolio.market.autotrd.modules.TMSVHtmlModuleEditor**

- *classPath*
  This parameter points to the location of the compiled analytic classes. **Note: i****t is important to keep the analytic classes in the location NOT pointed to by the CLASSPATH so that only custom TMS loader can load them from the classPath location. This way the analytics become re-loadable at runtime.**

- *ParameterList*
  Contains the type and the description of all parameters required by the module. At any given time multiple instances of the AutoTrader may be running in the system all using the same module but with different parameters.

Example:

1. Create an algorithm called SampleAnalytic.java and compile it as SampleAnalytic.class.

2. Create an HTML AutoTrader panel SampleAnalyticEditor.htm in [TMS HOME]\backend\Directory\ElTrader\TMS\Config folder.

3. Open [TMS HOME]/Directory/ElTrader/TMS/Config/AnalyticModuleMetaData.xml file and add a new AnalyticModuleMetaData block to <AnalyticModuleMetaDataList>. You can simply uncomment <AnalyticModuleMetaData type = "SampleAnalyticWithHTML" block in [TMS HOME]/Directory/ElTrader/TMS/Config/AnalyticModuleMetaData.xml as example. The classPath parameter should point to the location of the SampleAnalytic.class.

4. Save changes made to [TMS HOME]/Directory/ElTrader/TMS/Config/AnalyticModuleMetaData.xml

5. Start TMS backend and front-end and run your algorithm.

6. When changes are made to the algorithm and the classes are recompiled it is possible to reload the implementation without restarting any of the backend processes. Simply issue command "pfd at reload <analytic name>" on the ELTPortfolioSatellite process (See InfoReach TMS Operations Guide for the description of the AppAdmin tool).

NOTE: You can find sample algorithm in [TMS HOME]\backend\samples\java\SampleAnalytic folder.

Launch TMS server

Run Start_All.bat from [TMS HOME]\backend\win32\bin folder

Launch TMS GUI

Run ELTView.bat from [TMS HOME]\frontend\win32\bin folder

Launch AutoTrader

Invoke AutoTrader from Market Portfolio Tool menu AutoTrader-> New

Accessing resourcse from inside the AutoTrader module

All available TMS resources/events can be requested or subscribed for by calling methods of the ITMSAutoTrader interface. The handle to the object implementing this interface is obtained by calling getAutoTrader() method of the ***TMSAbstractAnalyticModule*** inside the analytic's code. Below all methods of the ITMSAutoTrader interface are described. Other methods of the ***TMSAbstractAnalyticModule*** include:

- *ITMSAutoTrader getAutoTrader()*
  Returns handle to ITMSAUtoTrader

- *ITMSAnalyticModuleMetaData getAnalyticModuleMetaData()*
  Returns handle to ITMSAnalyticModuleMetaData

- *String getType()*
  Returns the name of this AutoTrader

Subscribing AutoTrader for the TMS event flow

Upon initialization of the custom AutoTrader (in the onAutoTraderCreated() or onAutoTraderRecovered() method) the calls can be made to subscribe the module to the event flow generated by the TMS. Each of the *subscribe…()* methods below has one or more corresponding *on…()* methods that have to be implemented by the custom analytic. See section on listening to the TMS event flow for more details. To call the *subscribe()* methods use getAutoTrader() method to get a handle to the ITMSAutotTrader object.

- *void subscribeForOrderData() throws TMSException*
  Subscribes the module for events carrying the information about changes in order fields. For example changes in the field FillQty of the order. Only events from the orders generated from the targets of the portfolio managed by this AutoTrader instance will be passed to the module.

- *void subscribeForPortfolioData() throws TMSException*
  *S*ubscribes the module for events carrying the information about changes in portfolio fields that were caused by something other than market data updates or order flow (e.g. changes made through GUI or API).

- *void subscribeForTargetData() throws TMSException*
  Subscribes the module for events carrying information about changes in target fields. For example changes in the field ReleasedLeaves of the target. Only events from the targets of the portfolio managed by this AutoTrader instance will be passed to the module.

- *void subscribeForMarketData(String instrumentId) throws TMSException*
  Subscribes the module for events carrying market data records for specific instrument.

- *void subscribeForMarketData(String[] instrumentIds) throws TMSException*
  Subscribes the module for events carrying market data records for specific instruments.

- *void subscribeForCustomRecordData(String dataSourceName, String recordId) throws TMSException*
  Subscribes the module for events carrying data records from the custom data source.

- *void subscribeForCustomRecordData(String dataSourceName, String[] recordIds) throws TMSException*
  Subscribes the module for events carrying data records from the custom data source.

- *void subscribeForTableReportData(String domainManagerName, String reportName, String filter) throws RPTException*
  It is possible for an analytic module to subscribe for events from system or custom table reports. The first two parameters identify the report by name and by the name of the domain where this report was instantiated. By default, there is only one domain manager for reports in the TMS: "Portfolio report manager". Depending on the TMS configuration more domain managers may be present. The filtering criteria may be passed to this method in case only certain records from the report are needed. In the example below application subscribes to the event for all market targets for instrument 'IBM':

  ```
  subscribeForTableReportData("Portfolio report manager",
              "Market Targets",
              "\"Instrument\" = 'IBM'");
  ```

- *void subscribeForTreeReportData(String domainManagerName, String reportName, String level, String filter) throws RPTException*

It is possible for an analytic module to subscribe for events from system or custom tree reports. The first two parameters identify the report by name and by the name of the domain where this report was instantiated. By default, there is only one domain manager for reports in the TMS: "Portfolio report manager". Depending on the TMS configuration more domain managers may be present. The level parameter specifies the level in the tree whose nodes are requested. The filtering criteria may be passed to this method in case only certain nodes from the report level are needed. In the example below application subscribes to the event for all order transaction nodes for instrument 'IBM':

```
subscribeForTreeReportData("Portfolio report manager",
            "Single Order System Report",
            "OrdTrnId",
            "\"Instrument\" = 'IBM'");
```

- *void unsubscribeFromAutoTraderData() throws TMSException*
  Unsubscribes the module from events carrying information about changes in AutoTrader status or about modifications performed on the AutoTrader instance.

- *void unsubscribeFromOrderData() throws TMSException*
  Unsubscribes the module from events carrying the information about changes in order fields.

- *void unsubscribeFromTargetData() throws TMSException*
  Unsubscribes the module from events carrying information about changes in target fields.

- *void unsubscribeFromPositionData() throws TMSException*
  Unubscribes the module from events carrying information about changes in position values.

- *void unsubscribeFromCustomRecordData(String dataSourceName, String recordId) throws TMSException*
  Unsubscribes the module from events carrying data records from the custom data source.

- *void unsubscribeFromCustomRecordData(String dataSourceName, String[] recordIds) throws TMSException*
  Unsubscribes the module from events carrying data records from the custom data source.

- *void unsubscribeFromMarketData(String instrumentId) throws TMSException*
  Unsubscribes the module from events carrying market data records for specific instrument.

- *void unsubscribeFromMarketData(String[] instrumentIds) throws TMSException*
  Unsubscribes the module from events carrying market data records for specific instruments.

- *void unsubscribeFromReportData(String domainManagerName, String reportName) throws RPTException*
  Unsubscribes the module from report events.

Scheduling timers

Upon initialization of the custom AutoTrader (in the onAutoTraderCreated() or onAutoTraderRecovered() method) the calls can be made to schedule one or more timers (repeating or non-repeating) that will trigger the implementation of the module's *onTimer()* method.

- *long scheduleNonRepeatingTimer(int delayInSeconds)*
  Schedules non-repeating timer that will trigger the module's onTimer() method N seconds

from now. Returns the id of this timer to be used in the onTimer() method to distinguish this timer form others scheduled for this AutoTrader.

- *long scheduleNonRepeatingTimer(Date time)*
  Schedules non-repeating timer that will trigger the module's onTimer() method at time T. Returns the id of this timer to be used in the onTimer() method to distinguish this timer form others scheduled for this AutoTrader.

- *long scheduleRepeatingTimer(int delayInSeconds, int periodInSeconds)*
  Schedules repeating timer that will trigger the module's onTimer() method starting from K seconds from now and then every N seconds. Returns the id of this timer to be used in the onTimer() method to distinguish this timer form others scheduled for this AutoTrader.

- *long scheduleRepeatingTimer(int delayInSeconds, int periodInSeconds, int deviationInSeconds)*
  Schedules repeating timer that will fire after specified number of seconds and continue firing repeatedly with a certain interval with the interval being a random number uniformly distributed between (*periodInSeconds – deviationInSeconds)* and (*periodInSeconds + deviationInSeconds)*. Returns the id of this timer to be used in the onTimer() method to distinguish this timer form others scheduled for this AutoTrader.

- *void cancelTimer(long timerId)*
  Cancels timer with specified id.

- *void cancelAllTimers()*
  Cancels all timers for this AutoTrader instance.

Listening to the TMS event flow

As was mentioned above, all custom analytic modules will extend class ***com.inforeach.eltrader.tms.domain.portfolio.autotrader.modules. TMSAbstractAnalyticModule.***

This class implements a set of interfaces and provides empty implementations for each of the interfaces' methods. The custom analytic module will provide the actual implementation for those methods pertinent to the module's algorithm. It is inside these methods where the logic of the algorithm will be coded.

All of the methods described below are guaranteed to be called from the same thread for each AutoTrader instance. Thus, only one method can be executing at any given time.

The analytic will not be notified via callback methods of changes that it itself initiated. For example, onOrderUpdate() callback method will not be invoked for a modify request that the analytic module sends. However, onOrderUpdate() will be called when the resulting execution reports (e.g. confirmation, reject and/or replaced) are received from the counter-party.

The order of precedence for callbacks are as follows:

onAutoTraderUnassignedFromTarget()
onPortfolioUpdate()
onTargetRemoved()
onTargetAdded()
onTargetUpdate()
onTargetPaused()/onTargetResumed()/onTargetTerminated()
onOrderAdded()
onOrderUpdate()

onMarketDataUpdate()
onAutoTraderCreated()
onAutoTraderModified()
onAutoTraderPaused()
onAutoTraderResumed()
onAutoTraderAssignedToTarget()

Interface ITMSMarketPortfolioEventListener

The module's methods from this interface will always be called by the TMS if subscribeForPortfolioData() call was issued at initialization time.

- *void onPortfolioUpdate(ITMSLocalMarketPortfolio portfolio)*
  Called when any of the portfolio fields are changed by something other than market data updates or order flow (e.g. changes made through GUI or API).

Interface ITMSAutoTraderEventListener

The module's methods from this interface will be called by the TMS when AutoTrader status or any of its parameters change. No subscribe..() calls are necessary.

- *void onAutoTraderCreated()*
  Called when analytic module is instantiated and it is OK to initialize its state.
  For example, all subscribe calls and timer scheduling can be done here.

- *void onAutoTraderRecovered()*
  Called when analytic module is reovered as a result of the TMS restart.
  All subscriptions and timer scheduling should be re-requested at this point.

- *void onAutoTraderModified();*
  Called when AutoTrader is modified (for example targets are modified through API or through GUI).

- *void onAutoTraderPaused()*
  Called when AutoTrader's algorithm is paused (through API or through GUI).

- *void onAutoTraderResumed()*
  Called when AutoTrader's algorithm is resumed (through API or through GUI).

- *void onAutoTraderTerminated()*
  Called when AutoTrader's instance is terminated (through API or through GUI).

- *void onAutoTraderAssignedToTarget(ITMSMarketTarget target)*
  Called when target is assigned to AutoTrader's instance (through API or through GUI).

- *void onAutoTraderUnassignedFromTarget(ITMSMarketTarget target)*
  Called when target is unassigned from AutoTrader's instance (through API or GUI). It is also called when target is terminated while still associated to the AutoTrader's instance.

- *void onAutoTraderAction(Configurator parametersHolder)*
  Called when a custom instruction is sent to the AutoTrader's instance from the GUI. Each such instruction can contain the set of attribute-value pairs passed to this method as a Configurator. It is up to the module to interpret each instruction depending on the set of

attributes passed to it.

Interface ITMSOrderEventListener

The module's methods from this interface will be called by the TMS if subscribeForOrderData() call was issued at initialization time.

- *void onOrderAdded(ITMSOrder order)*
  This method of the AutoTrader is called when an order is issued for the target outside the algorithm (e.g. by manual sending of the wave from GUI).

- *void onOrderUpdate(ITMSOrder order)*
  Called when any of the order's fields change as a result of any TMS processing. See sample module for a detailed example.

- *void onOrderSendError(ITMSOrder order, ELTFieldGroup fieldsToBeSent, Throwable exception*)
  Reserved for future use (for the case when the orders are sent on a separate thread).

Interface ITMSMarketTargetEventListener

The module's methods from this interface will be called by the TMS if subscribeForTargetData() call was issued at initialization time.

- *void onTargetAdded(ITMSMarketTarget target)*
  Called when target is added as a result of modifying AutoTrader (through API or through GUI).

- *void onTargetPaused(ITMSMarketTarget target)*
  Called when AutoTrader's target is paused (through API or through GUI).

- *void onTargetResumed(ITMSMarketTarget target)*
  Called when AutoTrader's target is resumed (through API or through GUI).

- *void onTargetTerminated(ITMSMarketTarget target)*
  Called when AutoTrader's target is terminated (through API or through GUI).

- *void onTargetUpdate(ITMSMarketTarget target)*
  Called when any of the targte's fields change as a result of any TMS processing. See sample module for a detailed example.

- *void onTargetRemoved(ITMSMarketTarget target)*
  Called when target is removed as a result of modifying AutoTrader (through API or through GUI).

- *void onTargetAction(ITMSMarketTarget target, Configurator params)*
  Called when a custom instruction is sent to the AutoTrader's instance from the GUI. Each such instruction contains that target that will be affected by the instruction ann the set of attribute-value pairs passed to this method as a Configurator. It is up to the module to interpret each instruction depending on the set of attributes passed to it.

Interface ITMSMarketDataEventListener

The module's methods from this interface will be called by the TMS if one of the subscribeForMarketData() calls was issued at initialization time.

- *void onMarketDataUpdate(String instrumentId, IRDRecord record)*
  called when the TMS processes a market data tick for an instrument for which there was a subscription.

- *void onMarketDataFeedDisconnected()*
  called when the TMS detects the disconnect from the market data feed.

- *void onMarketDataFeedReconnected ()*
  called when the TMS detects the connect from the market data feed.

Interface ITMSCustomRecordDataEventListener

The module's methods from this interface will be called by the TMS if one of the subscribeForCustomRecordData() calls was issued at initialization time.

- *void onCustomRecordDataUpdate(String dataSourceName, String recordId, IRDRecord record)*
  called when the TMS processes a custom record update for a record id for which there was a subscription.

- *void onCustomRecordDataFeedDisconnected()*
  called when the TMS detects the disconnect from the custom data feed.

- *void onCustomRecordDataFeedReconnected()*
  called when the TMS detects the connect from the custom data feed.

Interface ITMSTimerHandler

The module's methods from this interface will be called by the TMS if one or more timers were scheduled at initialization time.

- *void onTimer(long timerId)*
  This method is called when the scheduled timer with id 'timerId' triggers.

Interface IRPTEventListener

- *void processEvent(RPTEvent event)*
  When the AutoTrader subscribes to the report data with either *subscribeForTableRpeort* or *subscribeForTreeReport* methods it is notified about the report events through this callback method. The record object is obtained form the event by calling getNewRecord() method. Once the record is obtained the fields of the record are retrieved by calling either getNumericFieldValueAt(int index) or getStringFieldValueAt(int index) methods. See Listing 1 of this document for an example of examining the report events. To find out what report the event came from use *((IRPTReport)event.getSource()).getName().*

Requesting Info from the TMS

AutoTrader Info

- *ITMSAutoTraderSpec getSpec()*
  Returns the spec of this AutoTrader instance. See the section on ITMSAutoTraderSpec for details.

- *int getStatus()*
  Returns the status of this AutoTrader instance.
  The status values are

  INITIALIZING = 0
  EXECUTING = 1
  PAUSED = 2
  UNINITIALIZING = 3
  TERMINATED = 4
  ERROR = 5

- *boolean isExecuting()*
  Returns true if status is EXECUTING, false otherwise.

Portfolio Info

- *String getMarketPortfolioName();*
  Returns the name of the portfolio with which this instance of the AutoTrader is associated

Orders Info

- *Iterator<ITMSOrder> getOpenOrdersForTarget(ITMSMarketTarget target)*
  Returns an iterator of open orders currently outstanding for the specified target. The iterator is re-used and therefore should not be retained by the analytic.

- *int getOpenOrderCountForTarget(ITMSMarketTarget target)*
  Returns number of open orders currently outstanding for the specified target.

- *Iterator<ITMSOrder> getAllOrdersForTarget(ITMSMarketTarget target)*
  Returns an iterator of all orders currently outstanding for the specified target. The iterator is re-used and therefore should not be retained by the analytic.

- *void runAlgorithmOnOrdersForTarget(ITMSMarketTarget target, ITMSOrderAlgorithm algorithm) throws TMSException*
  Performs a pre-defined action on each order outstanding for the target.

- *ITMSOrder getOrder(String transactionId)*
  Returns an order given its id (unique id is assigned to each order by the TMS).

- *ITMSWave getWave(String waveId)*
  Returns a wave object given the wave id.

Targets Info

- *Iterator getTargets()*
  Returns Iterator of *ITMSTarget* objects. This object represents the current set of targets of given AutoTrader instance. All parameters specified for the target can be obtained from ITMSTarget (see section below). The iterator is re-used and therefore should not be retained by the analytic.

- *Iterator getTargetsForInstrument(String instrumentId)*
  Returns Iterator of ITMSTarget objects for a given instrument (each AutoTrader can have multiple targets for the same instrument which differ in some parameter, e.g. OrderType). The iterator is re-used and therefore should not be retained by the analytic.

- *ITMSMarketTarget getTargetForOrder(ITMSOrder order)*
  Returns an ITMSMarketTarget object for a given order.

- *void runAlgorithmOnTargets(ITMSMarketTargetAlgorithm algorithm) throws TMSException*
  Performs a pre-defined action on each target of the portfolio.

- *void runAlgorithmOnTargetsForInstrument(String instrumentId, ITMSMarketTargetAlgorithm algorithm) throws TMSException*
  Performs a pre-defined action on each target of the portfolio with a given instrument.

Position Info

- *ITMSCategoryPosition getGlobalPosition()*
  Returns global (TMS-wide) position.

- *ITMSCategoryInstrumentPosition getGlobalInstrumentPosition(String instrumentId)*
  Returns global (TMS-wide) position value for the given instrument.

- *ITMSCategoryPosition getPortfolioPosition()*
  Returns this portfolio position value.

- *ITMSCategoryInstrumentPosition getPortfolioInstrumentPosition(String instrumentId)*
  Returns position value for the given instrument within the portfolio managed by this instance of the AutoTrader.

- *ITMSCategoryPosition getCategoryPosition(String categoryType, String categoryName)*
  Returns category position value.

- *ITMSCategoryInstrumentPosition getCategoryInstrumentPosition(String categoryType, String categoryName, tring instrument)*
  Returns category position value for the given instrument.

- *String[] getCategoryNames*(*String categoryType)*
  When retrieving position data for some categories an ananlytic may first retrieve all category names for specific category type, for example all Account names.

Market Data Info

- *IRDRecord getMarketDataRecord(String instrumentId)*
  Returns market data record for specific instrument. This record contains values from the latest market data update processed by TMS for this instrument.

Static Data Info

- *ITMSStaticDataFacility getStaticDataSource()*
  Returns an interface to the TMS' static data facility. Through this interface static information for each instrument in the TMS' security master can be obtained.

Custom Data Info

- *IRDRecord getCustomDataRecord(String dataSouceName, String recordId)*
  Returns a record from the custom data source that could be plugged into TMS. See section on custom data sources.

Report Info

- *void requestTableReportData(String domainManagerName, String reportName, String filter)*
  Same as subscribeForTableReportData() but will only subscribe the analytic for the current state of the table report and once the state is received as a collection of state events no further event notifications will arrive.

- *void requestTreeReportData(String domainManagerName, String reportName, String level, String filter)*
  Same as subscribeForTreeReportData() but will only subscribe the analytic for the current state of the tree report and once the state is received as a collection of state events no further event notifications will arrive.

Instructing the TMS to perform an action

Order Actions

- *void startWave() throws TMSException*
  If it is desirable to mark orders with the wave id this method can be issued before any sendOrder() methods are called. Each time this call is made the wave id is increased by 1.

- *ITMSOrder sendOrder(ITMSMarketTarget target, TMSNewOrderMessage message) throws TMSException*
  Sends an order for a specific target. TMSNewOrder message object will contain field values required for this order. The remaining field values will be taken from the target itself. For example, it is enough to specify an order quantity and price in the TMSNewOrderMessage and the Side, Instrument and other things will be copied from the target (if they exist). It is the responsibility of the module implementer to make sure that all information required for the order can be obtained from the TMSNewOrderMessage object or from the target itself. This method returns an instance of the sent ITMSOrder object. The onOrderAdded()method will not be called as a result of this call.

- *ITMSOrder sendNonTargetOrder(TMSNewOrderMessage message) throws TMSException*
  Sends an order not associated with a market target. TMSNewOrder message object will

contain field values required for this order. It is the responsibility of the module implementer to make sure that all information required for the order can be obtained from the TMSNewOrderMessage object or from the target itself. This is useful for use cases such as sending uncommitted probe messages to liquidity pools as we do not want messages of this nature to affect order related fields in a target.
This method returns an instance of the sent ITMSOrder object. The onOrderAdded()method will not be called as a result of this call however it will receive onOrderUpdate() notifications the same way as target associated orders.

- *void modifyOrder(ITMSOrder order, TMSModifyOrderMessage message) throws TMSException*
  Issues a Modiy request for the existing ITMSOrder order with parameters specified in TMSModifyOrderMessage message. This method will throw exception if there is an attempt to modify an order that is already being modified or canceled or is closed. The onOrderUpdate() method will not be called as a result of this call.

- *void cancelOrder(ITMSOrder order, TMSCancelOrderMessage message) throws TMSException*
  Issues a cancel request for the existing ITMSOrder order. This method will throw exception if there is an attempt to modify an order that is already being modified or canceled or is closed. The onOrderUpdate()method will not be called as a result of this call.

- *void cancelOpenOrders(ITMSMarketTarget target) throws TMSException*
  Issues a cancel request for all orders open for specific target. The onOrderUpdate()method will not be called as a result of this call.

- *void cancelOpenOrders() throws TMSException*
  Issues a cancel request for all open orders for all targets of the portfolio. The onOrderUpdate()method will not be called as a result of this call.

- *ITMSStagedPortfolioSystem getStagedPortfolioSystem()*
  Get reference to staged portfolio system which contains the staged portfolio operations

- *void cancelWave(String waveId) throws TMSException*
  Issues a cancel request for all open orders of the specific wave. The onOrderUpdate() method will not be called as a result of this call.

Target Actions

- *void addTarget(FieldsContainer targetFields) throws TMSException*
  Adds new target to a portfolio. All target fields will be set in the FieldsContainer. It is the responsibility of the user to make sure that all required fields are set. The onTargetAdded() method will not be called as a result of this call.

- *void modifyTarget(ITMSMarketTarget target, FieldsContainer newtFields) throws TMSException*
  Modifies specified target by setting target fields to new values specified in the newFields parameter. The onTargetUpdate() method will not be called as a result of this call.

- *void removeTarget(ITMSMarketTarget target) throws TMSException*
  Removes the target from the portfolio. The onTargetRemoved() method will not be called as a result of this call.

- *void removeAllTargets() throws TMSException*
  Removes all targets from the portfolio. The onTargetRemoved() method will not be called as a result of this call.

AutoTrader Actions

- *void pauseAutoTrader() throws TMSException*
  Changes the status of the AutoTrader to PAUSED. All the *on…()* methods will still be called while the AutoTrader is paused. It is the responsibility of the implementer to examine the status of the AutoTrader and to make a decision on what actions should or should not be taken. The method onAutoTraderModifed() method will not be called as a result of this call.

- *void resumeAutoTrader() throws TMSException*
  Changes the status of the AutoTrader to EXECUTING. The onAutoTraderModifed() method will not be called as a result of this call.

- *void terminateAutoTrader() throws TMSException*
  Changes the status of the AutoTrader to TERMINATED. The implementation has to do whatever cleanup it needs. No on...() methods will be called for the AutoTrader once its status becomes TERMINATED. The AutoTrader can never be resumed from the TERMINATED status. The onAutoTraderModifed() method will not be called as a result of this call.

- *void pauseTarget(ITMSMarketTarget target) throws TMSException*
  Changes the status of the target to PAUSED. The onAutoTraderModifed() method will not be called as a result of this call.

- *void suspendTarget(ITMSMarketTarget target) throws TMSException*
  Changes the status of the target to SUSPEND. The onAutoTraderModifed() method will not be called as a result of this call.

- *void resumeTarget(ITMSMarketTarget target) throws TMSException*
  Changes the status of the target to ACTIVE. The onAutoTraderModifed() method will not be called as a result of this call.

- *void terminateTarget(ITMSMarketTarget target) throws TMSException*
  Changes the status of the target to TERMINATED. The target can never be resumed from the TERMINATED status. The onAutoTraderModifed() method will not be called as a result of this call.

- *void loadTargetsFromResource(String resourceName, String formatResourceName, boolean removePrevTargets) throws TMSException*
  At runtime the AutoTrader can be instructed to reload the set of targets from the specified resource files (for example when targets are generated by external model). The removePrevTargets parameter controls whether all old targets will be removed or the existing will just be overwritten with new field values. The onAutoTraderModifed() method will not be called as a result of this call.

- *void modifyAutoTrader(ITMSAutoTraderSpec autoTraderSpec) throws TMSException*
  Modifies the AutoTrader parameters by overwriting them with values set in the new autoTraderSpec. The onAutoTraderModifed() method will not be called as a result of this call.

Portfolio Actions

- *void modifyPortfolio(FieldsContainer fields) throws TMSException*
  Modifies portfolio fields by overwriting them with values set in the new autoTraderSpec. The *onPortfolioUpdate* () method will not be called as a result of this call.

Auditing AutoTrader activity

The activity of the AutoTrader module may be monitored from the GUI through AutoTrader Auditing report. To send an event to be displayed on GUI in this report, use these methods:

- *void enableAuditSeverity(int severity, boolean value)*
  Enable an audit severity level.

- *boolean isAuditSeverityEnabled(int severity)*
  Check if an audit severity is enabled. This is highly recommend prior to building an error message to be sent using the sendAuditRecord() method.

- *void sendAuditRecord(String text, int severity) throws TMSException*
  Send a text containing information to be displayed in the Auditing report on GUI.

- *void sendAuditRecord(ITMSMarketTarget target, String text, int severity) throws TMSException*
  Send a text containing information to be displayed in the Auditing report on GUI. The associated target is included for further clarification.

- *postAlertMessageToOwner(String type, String description, String details, boolean isUrgent)*
  This method is used to post an alert into portfolio owner's Alert Console on the GUI.

The meaning of a given severity level is open to the interpretation of the analytic module implementer. The available levels are:

- AUDIT_SEVERITY_ERROR
- AUDIT_SEVERITY_WARNING
- AUDIT_SEVERITY_ACTIVITY_DETAILS_HIGH
- AUDIT_SEVERITY_ACTIVITY_DETAILS_MEDIUM
- AUDIT_SEVERITY_ACTIVITY_DETAILS_LOW

In addition to having viewable records of the audit messages in the GUI, the audit messages can also be persisted in a file. This is accomplished by specifying the name of a logging facility in the loggingFacilityName attribute of the Auditor configuration element in the backend\Directory\ElTrader\TMS\Config\tms.xml configuration file.

```
<Auditor
   impl = "com.inforeach.eltrader.tms.domain.ataudit.TMSAuditorSpec"
   reportManagerList = "AutoTrader/Basket report manager"
   reportSpecResource = ElTrader/TMS/Config/SystemReports/AutoTraderAuditReport.xml"
   loggingFacilityName = "AutoTraderAuditLogFacility"
/>
```

If specified, the default auditor implementation will write messages of the AUDIT_SEVERITY_ERROR severity using the error logger, AUDIT_SEVERITY_WARNING using the warning logger, and the remaining in the info logger.

ITMSStagedPortfolioSystem

Actions affecting staged portfolio entities.

- *ITMSStagedTarget getStagedTarget(long stagedTgtId)*
  Get reference to ITMSStagedTarget with the numeric long identifier stagedTgtId.

- *ITMSStagedTarget getStagedTarget(Long stagedTgtId)*
  Get reference to ITMSStagedTarget with the object long identifier stagedTgtId.

- *IRPTRecordContext getStagedTargetReportContext()*
  Get record context of the staged target report.

- *void confirmIfNecessary(UserTicket userTicket, ITMSMarketTarget marketTarget)*
  Confirm staged target associated with the given ITMSMarketTarget, if it is in a pending confirmation state, otherwise do nothing.

- *void confirmIfNecessary(UserTicket userTicket, ITMSStagedTarget stagedTarget)*
  Confirm staged target associated with the given ITMSMarketTarget if it is in a pending confirmation state, otherwise do nothing.

- *void applyPendingUpdate(UserTicket userTicket,*
  ITMSMarketTarget marketTarget,
  ITMSPendingTargetUpdate pendingUpdate)
  Apply pending update to the staged target associated with the given market target.

- *void rollbackPendingUpdate(UserTicket userTicket,*
  ITMSMarketTarget marketTarget,
  ITMSPendingTargetUpdate pendingUpdate,
  boolean rejectClientCancelRequest)
  Roll back pending update to the staged target associated with the given target.

- *void statusTarget(UserTicket userTicket,*
  ITMSStagedTarget stagedTarget,
  TMSStatusOrderMessage message)
  throws TMSException
  Send status update for staged target.

Entity Objects and Interfaces

The methods of the AutoTrader often return objects to the caller or require objects to be passed as parameters. The interfaces to these objects that represent orders/targets/portfolios/messages in the TMS are described below. Whenever an order, target or portfolio field name is required as a parameter use constants from the ***com.inforeach.eltrader.tms.api.ITMSConstants*** interface. For example ITMSOrder.getStringFieldValue(ITMSConstants.ORDFLDNAME_CURRENCY) will return the currency of the order. Also, some of the returned values may also be defined as constants, e.g. getSideTyoe() method of the ITMSORder interface will return either ITMSCOnstants.ORDFLDVAL_SIDE_TYPE_BUY or ITMSCOnstants.ORDFLDVAL_SIDE_TYPE_SELL.

The ITMSMarketTarget and ITMSOrder interfaces contain methods that can be used to retrieve particular values of a target or an order. However, target and order objects can contain many field values for which there are no accessor methods in these interfaces. The set of possible fields is as big as the set of fields defined by the FIX protocol or custom metadata. Only most commonly used fields have the accessor methods defined for them. To retrieve the value of other fields the generic

methods *double getNumericFieldValue(String fieldName)* and *String getStringFieldValue(String fieldName)* should be used. These methods come from the ITMSRecord interface, which both ITMSMarketTarget and ITMSOrder extend.

Interface ITMSLocalMarketPortfolio

- *String getName ()*
  Returns portfolio name.

- *int getType()*
  Returns portfolioType. One of the

  ```
  INDEX_MARKET_PORTFOLIO = 0;
  PURE_MARKET_PORTFOLIO = 1;
  LINKED_MARKET_PORTFOLIO = 2;
  CORRELATED_MARKET_PORTFOLIO = 3;
  ```

This interface extends IRecord and additionally has the same API as IRecord.

Interface ITMSMarketTarget

- *Object getIdentifier()*
  Returns target's unique identifier.

- *String getInstrumentId()*
  Returns instrument id of the target.

- *String getSymbol()*
  Returns exchange symbol of the target.

- *boolean isActive()*
  Returns true if the target is active (not paused and not terminated).

- *String getStatus()*
  Returns status value (see ITMSConstants. PFFLDVAL_TARGET_STATUS_... constants).

- *double getOrderQuantity()*
  Returns the quantity.

- *double getTargetValue()*
  Returns cash target value.

- *double getPrice()*
  Returns limit price.

- *int getSideType()*
  Returns side type value (see ITMSConstants.FLDVAL_SIDE_TYPE_BUY or ITMSConstants.FLDVAL_SIDE_TYPE_SELL constants).

- *char getSide()*
  Returns side value (see ITMSConstants.SIDE_... constants).

- *char getType()*
  Returns target's order type value (see ITMSConstants.ORDTYPE_... constants).

- *long getLastFillTime()*
  Returns time of the last fill that was received for any of the target's orders. (long)Double.NaN if not applicable.

- *double getLastFillQty()*
  Returns quantity of the last fill that was received for any of the target's orders. Double.NaN if not applicable.

- *double getLastFillPrice()*
  Returns price of the last fill that was received for any of the target's orders. Double.NaN if not applicable.

- *double getAvgFillPrice()*
  Returns average price of all fills received for all orders issued for the target.

- *double getFillValue()*
  Returns aggregated value of all fills received for all orders issued for the target.

- *double getFillQty()*
  Returns aggregated quantity of all fills received for all orders issued for the target.

- *double getReleasedQty()*
  Returns aggregated quantity of all orders issued for the target. In order to avoid over-releasing, analytic module can make sure that released quantity does not exceed target's size.

- *double getReleasedLeaves()*
  Returns aggregated leaves quantity of all orders issued for the target.

- *double getBuyFillValue()*
  Returns aggregated value of all fills received for all Buy orders issued for the target.

- *double getBuyFillQty()*
  Returns aggregated quantity of all fills received for all Buy orders issued for the target.

- *double getBuyReleasedLeaves()*
  Returns aggregated leaves quantity of all Buy orders issued for the target.

- *double getSellFillValue()*
  Returns aggregated value of all fills received for all Sell orders issued for the target.

- *double getSellFillQty()*
  Returns aggregated quantity of all fills received for all Sell orders issued for the target.

- *double getSellReleasedLeaves()*
  Returns aggregated leaves quantity of all Sell orders issued for the target.

- *double getNetFillVal()*
  Returns aggregated net value of all fills received for all orders issued for the target.

- *double getNetFillQty()*
  Returns aggregated net quantity of all fills received for all orders issued for the target.

- *double getNetReleasedLeaves()*
  Returns aggregated net leaves quantity of all orders issued for the target.

- *double getTargetLeaves()*
  Returns target leaves, i.e. non-filled portion of the target. If target does not have size specified, this method returns Double.NaN.

- *double getUnreleasedQty()*
  Returns unreleased quantity, i.e. un-issued portion of the target. If target does not have size specified, this method returns Double.NaN.

- *ITMSMarketPosition getPosition()*
  Returns position information.

- *ITMSOrder getFirstMarketOrder()*
  Returns first order released for the target.

- *ITMSOrder getLastMarketOrder()*
  Returns last order released for the target.

This interface extends IRecord and additionally has the same API as IRecord.

Interface ITMSOrder

- *Object getTransactionId ()*
  Returns order's unique transaction identifier.

- *boolean isOpen()*
  Returns true if leaves are not equal to zero, false otherwise.

- *String getInstrumentId()*
  Returns instrument id of the order.

- *String getTargetId()*
  Returns id of the target associated with the order.

- *String getSymbol()*
  Returns symbol of the order.

- *char getType()*
  Returns order type as described in the FIX specification (see ITMSConstants.ORDTYPE_... constants).

- *char getSide()*
  Returns order side as described in the FIX specification (see ITMSConstants.SIDE_... constants).

- *int getSideType()*
  Returns type of the order side (see ITMSConstants.FLDVAL_SIDE_TYPE_BUY or ITMSConstants.FLDVAL_SIDE_TYPE_SELL constants).

- *long getOrderTime()*
  Returns time the order was created.

- *double getOrderQuantity()*
  Returns original size of the order.

- *double getActiveOrderQuantity()*
  Returns current size of the order given all replace requests.

- *double getPrice()*
  Returns price at which the order was issued.

- *char getStatus()*
  Returns order status as described in the FIX specification (see ITMSConstants.PFFLDVAL_TARGET_STATUS… constants).

- *double getLeaves()*
  Returns number of shares yet to be filled.

- *double getFillQuantity()*
  Returns number of shares filled.

- *double getFillValue()*
  Returns number of shares filled * average fill price.

- *double getAvgFillPrice()*
  Returns average price of all fills for the order.

- *double getNetFillQuantity()*
  Returns number of shares filled signed based on side type, negative if sell, positive if buy.

- *double getNetFillValue()*
  Returns number of shares filled * average fill price signed based on side type, pos if sell, negative if buy.

- *long getLastFillTime()*
  Returns time of last fill received for this order. (long)Double.NaN if not applicable.

- *double getLastFillQty()*
  Returns quantity of last fill received for this order. Double.NaN if not applicable.

- *double getLastFillPrice()*
  Returns price of last fill received for this order. Double.NaN if not applicable.

- *String getAccountId()*
  Returns account id of the order. Null if not applicable.

- *String getAccountHierarchyId()*
  Returns account hierarchy id of the order. Null if not applicable.

- *String getTransactionDestination()*
  Returns transaction destination of this order.

- *char getTransitionState()*

Returns state of transition of this order between New, Modify, and Cancel. An order cannot be modified if it is not in a TRANS_STATE_READY and it cannot be canceled if it is in a TRANS_STATE_CANCEL_SEND_PENDING or a TRANS_STATE_CANCEL_PENDING state. The valid values are as follows (see ITMSOrder.TRANS_STATE_... constants):

TRANS_STATE_NEW_SEND_PENDING
New order message is pending to be sent through the engine.

TRANS_STATE_MODIFY_SEND_PENDING
Modify message is pending to be sent through the engine.

TRANS_STATE_MODIFY_PENDING
Expecting replace report to be received from the counterparty.

TRANS_STATE_CANCEL_SEND_PENDING
Cancel message is pending to be sent through the engine.

TRANS_STATE_CANCEL_PENDING
Expecting cancel report to be received from the counterparty.

TRANS_STATE_READY
Ready for new transition.

This interface extends IRecord and additionally has the same API as IRecord.

Interface ITMSMarketPosition

- *String getInstrumentId()*
  Returns instrument id.

- *double getTargetPositionShares()*
  Returns holdings in shares of this target (combined NetFillQty of all target's orders).

- *double getTargetPositionValue()*
  Returns holdings in $ of this target (combined NetFillVal of all target's orders).

- *double getPortfolioPositionShares ()*
  Returns combined holdings in shares of the instrument from all targets of this portfolio (combined NetFillQty of all target's orders for all targets with the given instrument).

- *double getPortfolioPositionShares ()*
  Returns combined holdings in $ of the instrument from all targets of this portfolio (combined NetFillVal of all target's orders for all targets with the given instrument).

- *double getGlobalPositionShares ()*
  Returns global holdings in shares of this instrument.

- *double getGlobalPositionShares ()*
  Returns global holdings in $ of this instrument.

Interface ITMSWave

- *String getWaveId()*
  Returns the id of this wave.

- *Iterator getOrders()*
  Returns an iterator of ITMSOrder object from this wave. The iterator is re-used and therefore should not be retained by the analytic.

- *String getPortfolioName()*
  Returns the name of the portfolio to which this wave belongs.

Interface IRecord

- *double getNumericFieldValue(String fieldId)*

- *String getStringFieldValue(String fieldId)*

- *char getCharFieldValue(String fieldId)*

- *boolean isFieldNumeric(String fieldId)*

- *boolean getBooleanFieldValue(String fieldId)*

- *long getTimeFieldValue(String fieldId)*

TMSNewOrderMessage

Objects of this type are passed to methods sending/posting new individual orders/slices through TMS. This object extends ELTFieldGroup and the ETLTFieldGroup API should be used to set any of the required FIX fields. The field names and tags can be looked up in the configuration file *Directory/ElTrader/FIXTMSMetaData/FIXFieldsSuperset.dtd* (see "INFOREACH FIX Java Programmers Guide.doc" section 2.2.1 for ELTFieldGroup API). The constants from ITMSConstants file starting with MSGFLDTAG_ prefix may be used in place of integer tags. The TMSNewOrderMessage also extends TMSBaseMessage and the TMSBaseMessage's method *setCommandFlag(int flag)* can be used to instruct the TMS to treat the new order message in several different ways:

- if *setCommandFlag(ITMSContanst.COMMANDFLAG_POST_MESSAGE*) is called then the message will be "posted" into the TMS rather than "sent". The difference between posting and sending is that in case of posting all the same processing of the message is done except that the message is not physically sent to the counterparty.

- if *setCommandFlag(ITMSContanst.COMMANDFLAG_IGNORE_ROUND_TO_LOT*) is called then the message will be sliced from the target and it is necessary to instruct the system to ignore round-to-lot instructions set in the target. In case of non-slice orders the round to lot flag has no effect as there is no rounding don anyway.

- if *setCommandFlag(ITMSContanst.COMMANDFLAG_DONE_AWAY*) is called then the message will posted into the system together with the artificial confirm and the Fill message with the fill quantity equal to the order quantity set in the message and the fill price equal to the limit price set in the message. Used for "tallying" market orders.

TMSModifyOrderMessage

Same as TMSNewOrderMessage only is used to modify existing order. In the simplest case will only contain fields that have to be modified. Additional fields may be set to be sent in the modify request.

TMSCancelOrderMessage

Same as TMSNewOrderMessage only is used to cancel existing order. Additional fields may be set to be sent in the cancel request.

Interface ITMSAutoTraderSpec

To modify AutoTrader parameters from the AutoTrader module itself one can issue a modifyAUtoTrader() call that takes an object of type ITMSAutoTraderSpeec as a parameter. To obtain this object getAutoTrader().getSpec() call can be made and then from the returned object the analytic module spec can be retrieved. It can be cast in most cases to TMSFieldListAnalyticModuleSpec. In advanced cases when implementer provided its own spec implementation different cast should be made. Once the spec is obtained its set..() methods can be called to set the new parameters and then the spec can be passed to the modifyAutoTrader() call:

```
moduleSpec = (TMSFieldListAnalyticModuleSpec)
getAutoTrader().getSpec().getAnalyticModuleSpec();
moduleSpec.set...();
getAutoTrader().modifyAutoTrader(spec);
```

ELTFieldGroup

See "INFOREACH FIX Java Programmers Guide.doc" section 2.2.1

FieldsContainer

Objects of this type are passed to methods adding/modifying portfolios/targets in TMS. This object's API should be used to set any of the required portfolio/target fields. The field names and tags can be looked up in the configuration file *Directory/ElTrader/TMS/Config?ElementSettings/TableElement/PortfolioTableFieldsMetadata.xml*. The constants from ITMSConstants file starting with PFFLDNAME_ prefix may be used in place of field ids.

- *setFieldValue(Object fieldId, String value)*
- *void setFieldValue(Object fieldId, double value)*
- *void setFieldValue(Object fieldId, boolean value)*
- *void setFieldValue(Object fieldId, char value)*
- *void setFieldValue(Object fieldId, long value)*
- *void clear()*
- *boolean containsStringField(Object fieldId)*
- *boolean containsNumericField(Object fieldId)*
- *String getStringFieldValue(Object fieldId)*
- *double getNumericFieldValue(Object fieldId)*
- *Iterator getStringFields()*
- *Iterator getNumericFields()*

Using AutoTrader Helpers

InfoReach TMS comes with few simple analytic modules that can be used out of the box for automatic portfolio trading. Along with the basic analytic modules there are also few *AutoTrader helper* modules that can be used by any analytic. For example, "order pegger" or "order ager" helpers can be used by a custom analytic to peg each outgoing order to the desired limit price or monitor outstanding orders and replace them according to the 5-min. rule. Thus, the implementer of the custom analytic module can always reuse the helpers instead of implementing the logic every time new AutoTrader is developed.

Sample Analytic Module

The code for the sample algorithms can be found in [TMS HOME]\samples\java\SampleAnalytic.

XML for registering the sample module

```
   <AnalyticModuleMetaData
        type = "SampleAnalytic"
        analyticModuleClassName = "com.mycompany.SampleAnalytic"

         metaDataClassName =
"com.inforeach.eltrader.tms.domain.portfolio.autotrader.modules.TMSFieldListAnalyticModuleMetaData"
         editorClassName =
"com.inforeach.eltrader.tms.view.portfolio.market.autotrd.modules.TMSVFieldListModuleEditor"
         specClassName =
"com.inforeach.eltrader.tms.domain.portfolio.autotrader.modules.TMSFieldListAnalyticModuleSpec"

          classPath = "../../samples/java/SampleAnalytic"
     >
       <ParameterList>
         <Parameter
              name = "pxMove"
              description = "Amount the price of the order differs from market before being
modified"
              type = "double"
         />
         <Parameter
              name = "delay"
              description = "Amount of time to delay start of trading in seconds"
              type = "integer"
         />
       </ParameterList>
    </AnalyticModuleMetaData>
```

Developing custom report fields

Using default TMS classes for custom fields

TMS comes with a comprehensive set of business fields already defined and described in field metadata files (Appendix B). It also has an extensive set of classes that can be used without any modification to describe new fields required for a specific TMS-based application. The list below describes all generic classes that can be used when defining new business field (for tree report). Once the field is defined then XML structure containing the field definition is added to one of the filed metadata files.

Order Report Fields

If field was developed for reports based on the Single Order Domain then its definition should be placed in file
/ElTrader/TMS/Config/ElementSettings/TreeElement/SingleOrderTreeFieldsMetaData.xml;

Class name:
com.inforeach.eltrader.tms.fields.TMSTreeRecordValueField

Metadata class name:
com.inforeach.report.tree.fields.RPTTreeRecordValueFieldMetaData

Description:

Use this class when you need a field that takes its value directly from the message that was last attached to the node. If the message does not have the corresponding field the previous value of the report field will be retained. Fields of this type can only be used in the leaf nodes because messages come to the transaction leafs only. These fields, thus, should always be added to the base domain reports because all other reports' structures re-use domain report levels. The *recordFieldId* in the fields' definition specifies the FIX fields from which the value should be taken. The type of the report field should match the type of this FIX field.

XML definition example:

```
<Field
    className = "com.inforeach.eltrader.tms.fields.TMSTreeRecordValueField"
    metaDataClassName =
"com.inforeach.report.tree.fields.RPTTreeRecordValueFieldMetaData"
    identifier = "OrdRqstId"
    type = "String"
    recordFieldId = "OrdRqstId"
/>
```

Class name:
com.inforeach.eltrader.tms.fields.TMSFirstChildBasedField

Metadata class name:
com.inforeach.report.tree.fields.RPTTreeRecordValueFieldMetaData

Description:

For the leaf nodes it works exactly like `TMSTreeRecordValueField`. On higher-level nodes the value of this field is taken from the same field of the first child that was added to the node. Then it is never recalculated. The *recordFieldId* in the fields' definition specifies the FIX fields from which the leaf node will take the value. If this field is present on some level of the report then it must be present on the level below.

XML definition example:

```
<Field
    className = "com.inforeach.eltrader.tms.fields.TMSFirstChildBasedField"
    metaDataClassName =  "com.inforeach.report.tree.fields.RPTTreeRecordValueFieldMetaData"
    identifier = "OrdTrnId"
    type = "String"
    recordFieldId = "OrdTrnId"
/>
```

Class name:
com.inforeach.eltrader.tms.fields.TMSGrouperOrFirstChildBasedField

Metadata class name:
com.inforeach.report.tree.fields.RPTTreeRecordValueFieldMetaData

Description:

Same as TMSFirstChildBasedField only on higher-level nodes it tries to take its value first from the actual level grouper name if it is the same as the field name and then, if it is not the same, it will take it from the first child.

XML definition example:

```
<Field
    className = "com.inforeach.eltrader.tms.fields.TMSGrouperOrFirstChildBasedField"
    metaDataClassName =  "com.inforeach.report.tree.fields.RPTTreeRecordValueFieldMetaData"
    identifier = "Symbol"
    type = "String"
    recordFieldId = "Symbol"
/>
```

Class name:
com.inforeach.eltrader.tms.fields.TMSParentLevelDependentField

Metadata class name:
com.inforeach.eltrader.tms.fields.TMSParentLevelDependentFieldMetaData

Description:

On the lowest two levels of the reports (transaction and request levels) this field is the same as `TMSFirstChildBasedField`. On higher level the value of this field is taken from the grouper value of the level above whose grouper id the same as this field's id. If such level does not exists then the value is N/A. For example, all nodes of the tree below the instrument level will have the same instrument field value set to the value of their parent on the instrument level.

XML definition example:

```
<Field
  className = "com.inforeach.eltrader.tms.fields.TMSInstrument"

metaDataClassName="com.inforeach.eltrader.tms.fields.TMSParentLevelDependentFieldMetaData"
    identifier = "Instrument"
     type = "String"
     recordFieldId = "Instrument"
     parentLevelId = "Instrument"
/>
```

Class name:
com.inforeach.eltrader.tms.fields.TMSSideTypeBasedSummationField

Metadata class name:
com.inforeach.eltrader.tms.fields.TMSNodeValueFieldMetaData

Description:

On the lowest two levels this field takes the value of some other field and then negates it if the value of SideType field is 'Sell'. Then on higher level this value is just an aggregation of the field values of the children. For example, if we have Buy and Sell orders for IBM and they are filled at 1000 shares then the value of NetFillQty field for the 'Sell' transaction will be –1000.0 and on higher levels the value of NetFillQty field will be 0. The *nodeFieldId* in the fields' definition specifies the id of the report field whose value will be negated (or left the same).

XML definition example:

```
<Field
  className = "com.inforeach.eltrader.tms.fields.TMSSideTypeBasedSummationField"
  metaDataClassName="com.inforeach.eltrader.tms.fields.TMSNodeValueFieldMetaData"
  identifier = "NetFillQty"
  type = "float"
  nodeFieldId = "FillQty"
/>
```

Class name:
com.inforeach.eltrader.tms.fields.TMSNodeValueDependentMarketDataField

Metadata class name:
com.inforeach.eltrader.tms.fields.TMSNodeValueDependentMarketDataFieldMetaData

Description:

This class is used for all market data fields for both tree and table reports. The *recordFieldId* parameter specifies from which field of the market data record the value will be taken. The *reportComparissonFieldId* parameter specifies the field from which the subscription id to market data resource will be taken. The example below described OpenPx field. The value of this field will be taken from the OpenPx field of the market data record that arrived from the MarketData source. The record arriving in this node is for the instrument specified in the Instrument field of this report.

XML definition example:

```
<Field
  className="com.inforeach.eltrader.tms.fields.TMSNodeValueDependentMarketDataField"

metaDataClassName="com.inforeach.eltrader.tms.fields.TMSNodeValueDependentMarketDataFieldMetaData"
  identifier = "OpenPx"
  type = "Price"
  recordFieldId = "OpenPx"
  recordComparissonFieldId = "ExchSymbol"
  reportComparissonFieldId = "Instrument"
  targetingSourcesDependencies="MarketData,Instrument"
/>
```

Class name:
com.inforeach.report.tree.fields.RPTTreeExprField

Metadata class name:
com.inforeach.report.tree.fields.RPTTreeExprFieldMetaData

Description:

This is a field that is calculated according to the formula based on other fields in the node. Naturally, the fields participating in the formula must exist. In the example below field PctTargetQtyFilled is calculated as a ratio of FillQty to TargetQty

XML definition example:

```
<Field
  className = "com.inforeach.report.tree.fields.RPTTreeExprField"
  metaDataClassName="com.inforeach.report.tree.fields.RPTTreeExprFieldMetaData"
  identifier = "PctTargetQtyFilled"
  type = "float"
  expression="FillQty/TargetQty"
/>
```

Class name:

com.inforeach.report.tree.fields.TMSTreeExprSummationFromDesignatedLevelField

Metadata class name:

```
com.inforeach.report.tree.fields.TMSTreeExprFromDesignatedLevelFieldMetaData
```

Description:

This is a field that is calculated on the designated level according to the formula based on other fields in the node. On all levels above the designated level the value of this field is an aggregation (by summation) of its children's values. Below the designated level this field is N/A. In the example below the designated level is TargetInstr.

XML definition example:

```
<Field
  className =
"com.inforeach.eltrader.tms.fields.TMSTreeExprSummationFromDesignatedLevelField"
  metaDataClassName =
      "com.inforeach.eltrader.tms.fields.TMSTreeExprFromDesignatedLevelFieldMetaData"
  identifier = "TargetFilledSlippage"
  type = "Price"
  expression="NetTargetFillQty * (TargetAvgFillPx - TargetOrdPx)"
  grouperIdentifierOfDesignatedLeaf = "TargetInstr"
/>
```

Class name:

com.inforeach.report.tree.fields.TMSTreeExprMultiplicationFromDesignatedLevelField

Metadata class name:

```
com.inforeach.report.tree.fields.TMSTreeExprFromDesignatedLevelFieldMetaData
```

Description:

This is a field that is calculated on the designated level according to the formula based on other fields in the node. On all levels above the designated level the value of this field is an aggregation (by multiplication) of its children's values. Below the designated level this field is N/A. In the example below the designated level is TargetInstr.

XML definition example:

```
<Field
  className =
"com.inforeach.eltrader.tms.fields.TMSTreeExprMultiplicationFromDesignatedLevelField"
  metaDataClassName =
      "com.inforeach.eltrader.tms.fields.TMSTreeExprFromDesignatedLevelFieldMetaData"
  identifier = "TargetFilledSlippage"
  type = "Price"
  expression="NetTargetFillQty * (TargetAvgFillPx - TargetOrdPx)"
  grouperIdentifierOfDesignatedLeaf = "TargetInstr"
```

```
/>
```

Class name:

com.inforeach.eltrader.tms.domain.singleorder.fields.TMSActiveRequestBasedField

Metadata class name:

com.inforeach.report.tree.fields.RPTTreeRecordValueFieldMetaData

Description:

Fields implemented through this class can be used only in Single Order Domain Report. The value of the field on the leaf ('request') level of the Single Order Domain Report will be taken from the corresponding field of the original request message. On the top ('transaction') level the value will be taken from the last child node (last request) whose status is active. In the example below the ClOrdId value on the single order transaction level will be equal to the ClOrdID value of the last active modify request.

XML definition example:

```
<Field
   className =
"com.inforeach.eltrader.tms.domain.singleorder.fields.TMSActiveRequestBasedField"
   identifier = "ClientOrdId"
   type = "alphaNumeric"
   recordFieldId = "ClOrdID"
/>
```

Class name:

com.inforeach.eltrader.tms.domain.singleorder.fields.TMSLastRecordBasedField

Metadata class name:

com.inforeach.report.tree.fields.RPTTreeRecordValueFieldMetaData

Description:

Fields implemented through this class can be used only in Single Order Domain Report. The value of the field on the leaf ('request') level of the Single Order Domain Report will be calculated as TMSTreeRecordValueField. On the top ('transaction') level the value will be taken from the last child node (last request) whose status is active. In the example below the ExecBroker value on the single order transaction level will be equal to the ExecBroker of the last active modify request. The request node's ExecBroker value will always be taken from the last message that arrived for this request.

XML definition example:

```
<Field
   className = "com.inforeach.eltrader.tms.domain.singleorder.fields.TMSLastRecordBasedField"
   identifier = "ExecBroker"
   type = "alphaNumeric"
   recordFieldId = "ExecBroker"
/>
```

Class name:

com.inforeach.eltrader.tms.domain.singleorder.fields.
TMSSingleOrderSideTypeBasedSummationField

Metadata class name:

```
com.inforeach.eltrader.tms.fields.TMSNodeValueFieldMetaData
```

Description:

Use this class rather than TMSSideTypeBasedSummationField for all reports based on Single Order Domain transactions.

XML definition example:

```
<Field
   className
=com.inforeach.eltrader.tms.domain.singleorder.fields.TMSSingleOrderSideTypeBasedSummationField"
   metaDataClassName="com.inforeach.eltrader.tms.fields.TMSNodeValueFieldMetaData"
   identifier = "NetLeaves"
   type = "float"
   nodeFieldId = "Leaves"
/>
```

Class name:

com.inforeach.eltrader.tms.domain.autotrd.fields.
TMSTargetSideTypeBasedSummationField

Metadata class name:

```
com.inforeach.eltrader.tms.fields.TMSNodeValueFieldMetaData
```

Description:

Use this class rather than TMSSideTypeBasedSummationField for all reports based on AutoTrader Domain transactions.

XML definition example:

```
<Field
   className =
"com.inforeach.eltrader.tms.domain.autotrd.fields.TMSTargetSideTypeBasedSummationField"
   metaDataClassName="com.inforeach.eltrader.tms.fields.TMSNodeValueFieldMetaData"
   identifier = "NetTargetFillQty"
   type = "float"
   nodeFieldId = "TargetFillQty"
/>
```

Target Fields

If field was developed for the reports based on the portfolio targets then its definition should be placed in file
/ElTrader/TMS/Config/ElementSettings/TableElement/PortfolioTableFieldsMetaData.xml;

Class name:

com.inforeach.report.table.fields.RPTTableExprField

Metadata class name:

```
com.inforeach.report.table.fields.RPTTableExprFieldMetaData
```

Description:

This is a field that is calculated according to the formula based on other fields in the target record. Naturally, the fields participating in the formula must exist. In the example below field NetTgtVal is calculated as a difference of SellTgtVal and BuyTgtVal

XML definition example:

```
<Field
   type="Price"
   className="com.inforeach.report.table.fields.RPTTableExprField"
   metaDataClassName="com.inforeach.report.table.fields.RPTTableExprFieldMetaData"
   identifier="NetTgtVal"
   expression="SellTgtVal - BuyTgtVal"
/>
```

Class name:

com.inforeach.eltrader.tms.domain.portfolio.report.fields.TMSPortfolioOrderBasedSummationField

Metadata class name:

```
com.inforeach.eltrader.tms.domain.portfolio.report.fields.
                                        TMSPortfolioOrderBasedSummationFieldMetaData
```

Description:

The value of this field is a sum of values of the specified field from the orders released from the target. The example below defines field Released which is a sum of ActiveOrdQty of all releaed orders.

XML definition example:

```
<Field
  className =
"com.inforeach.eltrader.tms.domain.portfolio.report.fields.TMSPortfolioOrderBasedSummationField"
   metaDataClassName="com.inforeach.eltrader.tms.domain.portfolio.report.fields.

TMSPortfolioOrderBasedSummationFieldMetaData"
   identifier = "Released"
```

```
        orderFieldId = "ActivOrdQty"
        type = "Qty"
    />
```

Class name:

com.inforeach.eltrader.tms.domain.portfolio.report.fields.TMSPortfolioSideBasedSummationField

Metadata class name:

com.inforeach.eltrader.tms.domain.portfolio.report.fields.TMSPortfolioSideBasedSummationFieldMetaData

Description:

The value of this field is a sum of values of the specified field from the Buy/Sell orders released from the target. The example below defines field BuyReleasedQty which is a sum of ActiveOrdQty of all releaed Buy orders.

XML definition example:

```
      <Field
        className =
"com.inforeach.eltrader.tms.domain.portfolio.report.fields.TMSPortfolioSideBasedSummationField"
        metaDataClassName="com.inforeach.eltrader.tms.domain.portfolio.report.fields.

TMSPortfolioSideBasedSummationFieldMetaData"
        identifier = "BuyReleasedQty"
        orderFieldId = "ActivOrdQty"
        side="buy"
        type = "Qty"
    />
```

Class name:

com.inforeach.eltrader.tms.fields.TMSProductStaticGroupingTableField

Metadata class name:

com.inforeach.eltrader.tms.fields.TMSProductStaticGroupingTableFieldMetaData

Description:

The value of this field is calculated based on target's instrument and its specific grouping recorded in the security master. The example below defines field Industry which is based on the "Industry" grouping of the target's instrument.

XML definition example:

```
     <Field
        className="com.inforeach.eltrader.tms.fields.TMSProductStaticGroupingTableField"

metaDataClassName="com.inforeach.eltrader.tms.fields.TMSProductStaticGroupingTableFieldMetaData"
        identifier="Industry"
        type="alphaNumeric"
        groupingType="Industry"
```

```
/>
```

Class name:

com.inforeach.eltrader.tms.fields.TMSInstrumentStaticDataTableField

Metadata class name:

com.inforeach.eltrader.tms.fields.TMSInstrumentStaticDataTableFieldMetaData

Description:

The value of this field is calculated based on target's instrument and its specific attribute recorded in the security master. The example below defines field SEDOL.

XML definition example:

```
<Field
   className="com.inforeach.eltrader.tms.fields.TMSInstrumentStaticDataTableField"

metaDataClassName="com.inforeach.eltrader.tms.fields.TMSInstrumentStaticDataTableFieldMetaData"
   identifier="SEDOL"
   type="alphaNumeric"
   staticFieldId="SEDOL"
/>
```

Class name:

com.inforeach.eltrader.tms.fields.TMSProductStaticDataTableField

Metadata class name:

com.inforeach.eltrader.tms.fields.TMSProductStaticDataTableFieldMetaData

Description:

The value of this field is calculated based on target's instrument's underlying product and its specific attribute recorded in the security master. The example below defines field Issuer.

XML definition example:

```
<Field
   className="com.inforeach.eltrader.tms.fields.TMSProductStaticDataTableField"

metaDataClassName="com.inforeach.eltrader.tms.fields.TMSProductStaticDataTableFieldMetaData"
   identifier="Issuer"
   type="alphaNumeric"
   staticFieldId="Issuer"
/>
```

Developing new classes for custom fields

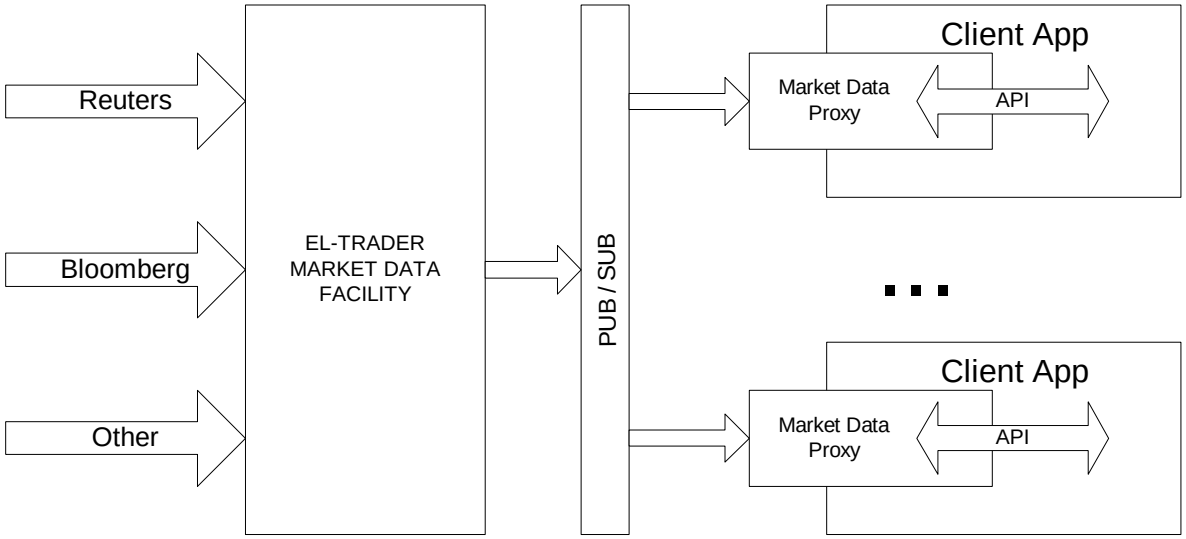
To be completed.

Using TMS Market Data API

Writing Market Data Listener components

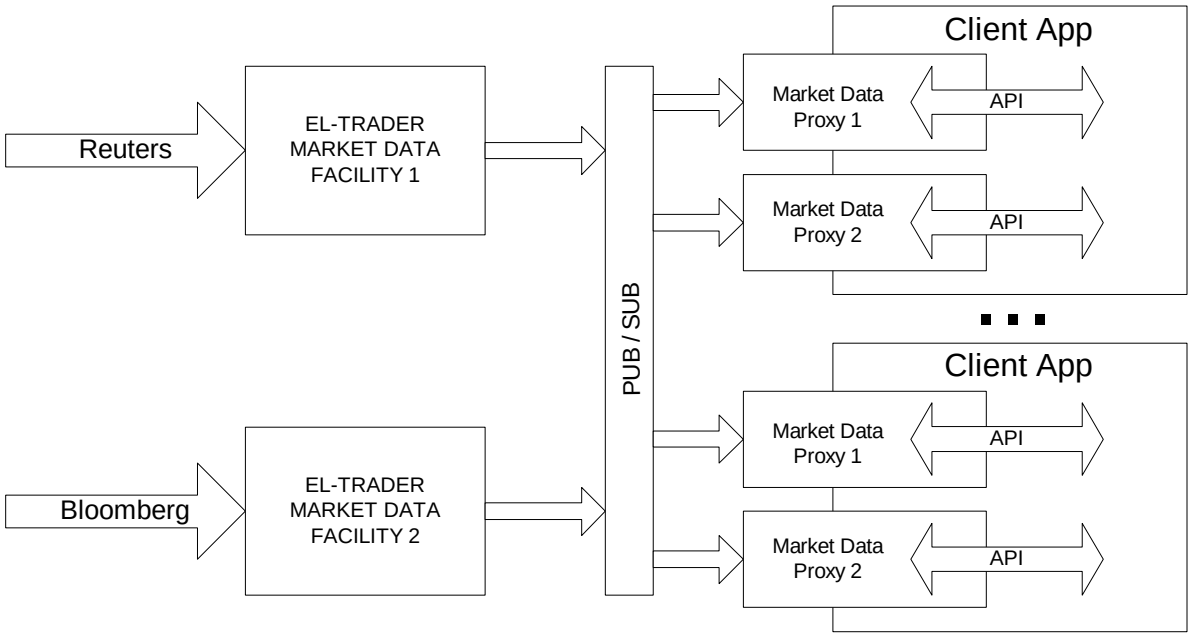
TMS's Market Data Facility (MDF) accepts the flow of market data records from one or several third party providers and then publishes the data in the form of homogeneous records to all TMS components. TMS components internally use MDF API to subscribe for info from the MDF. This same API can be used by custom applications to subscribe for the record flow from the TMS MDF. By using this API to receive market data info from the TMS MDF users can avoid coding to each third party provider and achieve faster time to market for their market-data-dependent apps.

As shown in the figure below the custom application has to be configured to contain Market Data Facility Proxy (MDF Proxy) in order to receive the records from MDF. All communication between the client app and the MDF is done through the MDF Proxy. Thus, the client app is completely oblivious the details of TMS deployment. The MDF can be running in any process on any machine. As long as the proxy is configured correctly to communicate with MDF the client app can receive the market data flow.



When MDF receives data from more than one third party provider it can create records that contain superset of the providers' fields. It makes sense to combine more than provider under the same MDF in case providers complement each other. Values from overlapping fields will overwrite each other in the MDF's record. For example, if MDF receives data from Reuters and Bloomberg and if both Reuters and Bloomberg provide LastPx then the resulting record published by MDF will have LastPx value equal to that of the provider that send its update last.

In case when it is necessary to distinguish Reuters' LastPx from Bloomberg's LastPx slightly different deployment may be used. See figure below.



To configure a process to contain an MDF Proxy it is necessary to specify RecordFacility section in the process' configuration (file *Directory/ElTrader/TMS/processes.xml*). The listing below shows an example of such process configuration. The MDFListener process will contain MDF proxy that would communicate with MDF called "Market Data" through RMI. Of course, there will have to be a process running on the network that actually hosts the MDF with name "Market Data".

```
<MDFListener>

    <GlobalSystem
        configDataResource = "/ElTrader/TMS/Config/system.xml"
        logFacilityName = "MDFListenerLogFacility"
    >

        <JMSSystem>
            <TopicConnection
                name = "MDFListenerEventBroker"
            />
        </JMSSystem>

        <RecordFacility
            name = "MarketData"
            fieldsMetadataResource = "/ElTrader/TMS/Config/MarketData/MarketQuoteFieldsMetaData.xml"
            contextAssignerClassName = "com.inforeach.recorddata.DefaultContextAssigner"
            propagateWholeRecord = "true"
            coalesceRecords = "false"
        >
            <AdjacentRecordFacility
                name = "MarketData"
            >
                <OutwardAdapter
                    kind = "RMI"
                />
            </AdjacentRecordFacility>
        </RecordFacility>
    </GlobalSystem>


  </MDFListener >
```

The following MDF API calls can be made inside the process containing an MDF Proxy:

- `public void`

```
IRDRecord getRecordAndSubscribe(IRDRecordFacilityListener facilityListener,
                                String name)
```

The client application calls this method whenever it needs to subscribe for market data updates for a specific instrument. This call returns an initial record right away. If there was a previous subscription for the same instrument through the same MDF proxy then the initial record will contain valid values, otherwise, if it the first subscription, the initial record will contain empty values except in the rare case when the update from MDF came in while the subscription was in progress. The implementation of MDF Proxy guarantees that updates are not lost during the subscription process. Once this call is made the implementer of the IRDRecordFacilityListener interface passed by the user in place of the first parameter will be invoked each time the MDF Proxy received an update from the MDF. The IRDRecordFacilityListener interface has only one method

```
void onUpdate(IRDRecord record);
```

Inside this method the values of the record may be examined and used in custom calculations. See sample code below for an example

- ```
  public void
  IRDRecord unsubscribe(IRDRecordFacilityListener facilityListener,
                        String name)
  ```

The client application calls this method whenever it needs to unsubscribe the listener from the market data updates for a specific instrument. The listener should be the same one previously passed to the getRecordAndSubscribe call.

The following is an example of the MDFListener process using MDF API. This example assumes the configuration described above.

```
// Import section
import com.inforeach.util.*;
import com.inforeach.util.log.*;
import com.inforeach.util.xml.*;
import com.inforeach.recorddata.*;

// Main application class:
public class MDFListenerApp
{
      public static void main(String args[])
      {
        // get main parameters including bootstrap file:
        String bootstrap_file_name = args[0];
        String processType = "MDFListener";

        try
        {
            // initialize GlobalSystem and TMS components
            XMLData bootstrapData = new XMLData(bootstrap_file_name);
            GlobalSystem.initialize(bootstrapData, processType);


            // get MDF proxy handle:
            RDRecordFacilityProxy proxy =
                  GlobalSystem. getRecordFacilityProxy("MarketData");

            // subscribe for IBM
```
```

```
            IRDRecord initialRecord =
                  IRDRecordproxy.getRecordAndSubscribe(new MDFListener(), "IBM");

            // do something with initial record or ignore it;
            // at this point method onUpdate() of the MDFListener object
            // will be called each time market data for IBM ticks

         }
         catch(Exception e)
         {
            e. printStackTrace(System.out);
         }
      } // end of main()

      // class implementing IRDRecordFacilityListener inerface
      private static class MDFListener implements IRDRecordFacilityListener
      {
            public void onUpdate(IRDRecord record)
            {
                  double ask = record.getDoubleFieldValue(1001);
                  double bid = record.getDoubleFieldValue(1004);
                  double mid = (ask + bid ) / 2.0;

                  // the tag numbers to pass to the record's get() methods
                  // and the type of the get() method to call can be looked
                  // up in file
                  // Directory\ElTrader\TMS\Config\MarketData\MarketQuoteFieldsMetaData.xml
            }
      }
}
```

Augmenting market data records with custom values

Sometimes it is necessary to augment market data records sent from Market Data Facility to backend processes and to GUIs with custom information. This information can be anything from custom calculated values to some static instrument info. TMS's Market Data Facility provides simple mechanism for plugging in a component that can enhance market data records right before they are published to all listeners.

Implementing custom component for record augmentation

A component that enhances records published from our Market Data Facility must implement interface **com.inforeach.recorddata.IRDProviderPostprocessor.** The sample code below shows basic example of a custom postprocessor that sets custom field with tag 2001 ("RecommendedOrderType") in each Market Data record and also print outs fields that have changed from the previous tick:

```
import com.inforeach.recorddata.IRDProviderPostprocessor;
```

```
public class SimulatorPostprocessor
     implements IRDProviderPostprocessor
{
      public void process(IRDRecordFacilityForProvider facility,
                          IRDRecord record,
                          IRDRecordFieldChecker fieldChecker)
      {
          System.out.println("\n=========================");

          IIntIterator iter = fieldChecker.getFieldIterator();

          while ( iter.hasNext() )
          {
              int tag = iter.next();

              if (fieldChecker.hasFieldChanged(tag))
              {
                  if (record.getContext().isFieldNumeric(tag))
                      System.out.print(" changed " + tag +
                      ": " + record.getDoubleFieldValue(tag));
                  else
                      System.out.print(" changed " + tag +
                      ": " + record.getStringFieldValue(tag));
              }
          }

          record.setStringFieldValue(2000, "Limit");
      }
}
```

Each field in Market Data record is described in file

Directory\ElTrader\TMS\Config\MarketData\MarketQuoteFieldsMetaData.xml.

Currently tags 1 – 2000 are reserved for the standard TMS set of fields. Therefore, the custom fields should have tag numbers greater than 2000. By adding the XML element describing the custom field to file

Directory\ElTrader\TMS\Config\MarketData\MarketQuoteFieldsMetaData.xml

one instructs TMS's Market Data Facility to reserve a place for the field in each record so that the user's postprocessor can set the field's value.

In our example the SimulatorPostprocessor from Listing 1 will only work if the following is added to

Directory\ElTrader\TMS\Config\MarketData\MarketQuoteFieldsMetaData.xml:

```
        <?xml version="1.0"?>

  <MarketMetadata
      defaultFieldsMetaDataName = "Marketdata MetaData"
  >
      <FieldList
         name="Marketdata MetaData"
         defaultFieldMetaDataClassName="com.inforeach.recorddata.RDFieldMetaData"
      >

           <!-- Added custom field for Recommended Order Type -->
```

```
<Field
      identifier = "2001"
      name = "RecommendedOrderType"
      type = "alphaNumeric"
/>
```

Of course, the name of the field and its type are absolutely arbitrary. The only restriction is that depending on the type of the custom field the specific set() method inside the postprocessor must be called. For example:

record.ngFieldValue() for alphanumeric field types
record.setIntFieldValue() for integer field types
record.setDoubleFieldValue() for double field types

In addition to setting custom field values the postprocessor can set any standard field in the record and it can also get any value from the record if it is required for calculations. To get a value from a record use get() methods of the record object. For example, to calculate custom mid price one can do this:

```
double ask = record.getDoubleFieldValue(1001);
double bid = record.getDoubleFieldValue(1004);
double mid = (ask + bid ) / 2.0;
record.setDoubleFieldValue(tagOfCustomMidPriceField, mid);
```

In order to plug in the custom postprocessor into the Market Data Facility one has to only add one attribute **postprocessorImplementationClass** to configuration file

Directory\ElTrader\TMS\Config\MarketData\MarketQuoteProviders.xml

For example:

```
<RecordFacilityProviderList
   delayedUpdating = "10"
   topicNameAssignerImplementationClass = "com.inforeach.eltrader.tms.recorddata.TMSTopicNameAssigner"
   cacheResourceFile="/ElTrader/TMS/Config/MarketData/SymbolsCache.cfg"
   postprocessorImplementationClass = "com.inforeach.recorddata.SimulatorPostprocessor"
>
```

Displaying custom values in Quote Reports

Once the postprocessor is implemented and plugged in, the records from Market Data Facility will have all the values filled by the postprocessor and they can be used for calculations on the backend including the AutoTrader analytics. However, in order to display the new values in Quote Reports more changes to configuration are necessary.

First, the file

Directory\ElTrader\TMS\Config\ElementSettings\TableElement\TableFieldsMetaData.xml

has to be changed to describe the new report field. In our example the following should be added:

```
<Field
    className = "com.inforeach.recorddata.report.RDTableRecordValueField"
    identifier = "RecommendedOrderType"
    type = "String"
```

```
        recordFieldId = "RecommendedOrderType"
    />
```

Then the report specification of the Stock Quotes Report has to change to include the new report field. File

Directory\ElTrader\TMS\Config\SystemReports\StockQuotesReport.xml

will have additional line:

```
<Fieldid="RecommendedOrderType"/>
```

Finally, the GUI specification of the Stock Quotes Report has to change to include the new report field. File

Directory\ElTrader\TMS\Users\all\StockQuotesReportViewSpec.xml

will have additional column:

```
<Column
      id="RecommendedOrderType"
      width="75"
/>
```

When all this is done erase *StockQuotesReportViewSpec.xml* from each user's profile so that it is picked up from Users/all.

Appendix A: Sample TMS accessing application

The sample file ELTClientApp.java is located in the folder backend/samples/java/ELTClientApp. In the same folder the JBuilder project can be found. To run the sample run backend/win32/bin/ELTClientApp.bat (the TMS server has to be running)

Appendix B: TMS fields

Order fields (used when working with ITMSOrder or IRecord objects representing TMS orders)

File [inforeach_home]/backend/Directory/ElTrader/TMS/Config/ElementSettings/TreeElement/SingleOrderTreeFieldsMetaData.xml contains definition of all TMS fields for Single Order Domain reports. When examining order events received in the external applications or inside the analytics the exact field names and their types may be looked up in this file. The table below describes some (but not all) fields. The fields marked by * are available only on order request level. The records from this level are only accessible to components subscribing to this order request level in the reports (see section "Using ITMSRemoteClient API for report data monitoring" of this document).

Field Name	Type	Representation	Description
OrdTrnId	String	String	The id assigned by TMS to every single order transaction
OrdRqstId*	String	**STRING**	**THE ID ASSIGNED BY TMS TO EVERY SINGLE ORDER REQUEST (ONE SINGLE ORDER TRANSACTION CONSISTS OF MULTIPLE SINGLE ORDER REQUESTS: THE ORIGINAL REQUEST AND ALL CONSEQUENT MODIFY, CANCEL REQUESTS)**
ConnName*	String	String	FIX engine connection name where request was originated
OrdRec*	int	Numeric	Index of the order message in the request node
AckRec*	int	Numeric	Index of the acknowledgement message in the request node
AckTime*	UTCTimestamp	Numeric	Time of the acknowledgement of the request
RecErrors*	int	Numeric	Errors detected in execution reports if consistency checking is turned on.
OrdStatus	char	Numeric	Valid values: A = UnAck 0 = New 1 = Partial 2 = Filled 3 = Done 4 = Canceled 5 = Replaced

			6 = Pending C/R 7 = Stopped 8 = Rejected 9 = Suspended B = Calculated C = Expired D = Accepted for bidding X = Implicitly Rejected
IsOrdActive*	Boolean	Numeric	Specifies whether this request is the current active request in the transaction (usually latest modify)
ClientOrdId*	String	String	Request's ClOrdID from the order message
Side	char	Numeric	Request's Side from the order message Valid values: 1 = Buy 2 = Sell 3 = Buy minus 4 = Sell plus 5 = Sell short 6 = Sell short exempt 7 = Undisclosed (valid for IOI and List Order messages only) 8 = Cross (orders where counterparty is an exchange, valid for all messages except IOIs) 9 = Cross short A = Cross short exempt
OrdQty	float	Numeric	Request's OrdQty from the order message. Is an aggregate on levels above the transaction level
ActiveOrdQty	float	Numeric	Part of request's order qty that was not modified or canceled. Is an aggregate on levels above the transaction level
FillQty	float	Numeric	Fill qty of the request. Is an aggregate on levels above the transaction level
FillVal	double	Numeric	Fill qty * avg. fill price. Is an aggregate on levels above the transaction level
1stFillRec*	int	Numeric	Index of the first arrived fill message
1stFillTime*	UTCTimestamp	Numeric	Time of the first arrived fill message
LastFillRec*	int	Numeric	Index of the last arrived fill message
LastFillTime	UTCTimestamp	Numeric	Time of the last arrived fill message
Leaves	float	Numeric	Unfilled qty of the request. Is an aggregate on levels above the transaction level
RqstState*	int	Numeric	Internal state if the request. Only used by auxiliary TMS components like validator, router, etc.

			Valid values: -1 = NA 1 = PendingValidation 2 = FailedValidationProcess 3 = PendingToBeSent 4 = Internalized 5 = PendingRelease 6 = BeingSent 7 = Sent 8 = UnderSpecialHandling
LatestActiveRqstId*	String	String	Id of the latest active request of the transaction
OrdType	char	Numeric	Request's OrdType from the order message
OrdPx	double	Numeric	Request's OrdPx from the order message
StopPx	double	Numeric	Request's StopPx from the order message
OrdVal	double	Numeric	OrdQty * OrdPx. Is aggregate on higher levels. Is an aggregate on levels above the transaction level
AvgOrdPx	double	Numeric	Has meaning on higher levels: OrdVal / OrdQty
OpenOrdVal	double	Numeric	Same as OrdVal only calculated based on leaves. Is an aggregate on levels above the transaction level
OrdTime	UTCTimestamp	Numeric	Time of the processing of the order request
OrdInternalTime	UTCTimestamp	Numeric	
NetFillQty	float	Numeric	FillQty adjusted by Side. Is an aggregate on levels above the transaction level
NetFillVal	float	Numeric	FillVal adjustedby Side. Is an aggregate on levels above the transaction level
AutoTrdId	String	Numeric	Id of the AutoTrader that initiated the order
SliceNumber	int	Numeric	Sif this order was a slice of a basket target then it will have the slice number field
Custodian	String	String	Request's Custodian from the order message
DestID	String	String	Request's DestID from the order message
SenderID	String	String	Request's SenderID from the order message
LastOrdTime	UTCTimestamp	Numeric	Time of the last request of the transaction
ExecBroker	String	String	Request's ExecBroker from the order message

AvgFillPx	double	Numeric	Avg. fill price.
NetLeaves	float	Numeric	Leaves adjusted by Side. Is an aggregate on levels above the transaction level
NetOrdQty	float	Numeric	OrdQty adjusted by Side. Is an aggregate on levels above the transaction level
NetOrdVal	float	Numeric	OrdVal adjusted by Side
NetOpenOrdVal	float	Numeric	OpenOrdVal adjusted by Side. Is an aggregate on levels above the transaction level
OpenOrClose	char	Numeric	For options. Request's OpenOrClose from the order message
ExpTime	UTCTimestamp	Numeric	For options. Request's ExpTime from the order message
OrdSender	String	String	ID of the originator of the order
MsgType	String	String	Request's MsType from the order message ('D' or 'G' or 'F')
OrigClientOrdId	String	String	Request's OrigClOrdID from the order message
LeavesSlippage	float	Numeric	Leaves * (LastPx - OrdPx). Is an aggregate on levels above the transaction level
FilledSlippage	float	Numeric	FillQty * (LastPx - OrdPx). Is an aggregate on levels above the transaction level
OrdSlippage	float	Numeric	LeavesSlippage + FilledSlippage. Is an aggregate on levels above the transaction level
TargetQty	float	Numeric	If order was originated by AutoTrader or Basket this field shows original target qty
PctTargetQtyFilled	float	Numeric	FillQty / TargetQty

* - *fields marked by * are only accessible from level OrdRqstId through methods subscribeForReportData or requestReportData.*

Portfolio target fields (used when working with ITMSMarketTarget, ITMSLocalMarketPortfolio, or IRecord objects representing TMS targets or TMS portfolios)

File [inforeach_home]/backend/Directory/ElTrader/TMS/Config/ElementSettings/TableElement/PortfolioTableFieldsMetaData.xml contains definition of all TMS fields for portfolio targets or portfolio summary records. When examining target/portfolio events received in the external applications or inside the analytics the exact field names and their types may be looked up in this file.

Market quote fields (used when working with IRDRecord objects)

File [inforeach_home]/backend/Directory/ElTrader/TMS/Config/MarketData/MarketQuoteFieldsMetaData.xml contains definition of all TMS fields for market data records. When examining market data events received in the external applications or inside the analytics the exact field names and their types may be looked up in this file.

Appendix C: Sample report resource

The following XML structure contains specification of a simple report based on the Single Order Domain base report. This tree report aggregates all single order transactions from the base report into a tree structure grouped by Account, SubAccount, and Industry. On each level the set of fields is specified. On Account, SubAccount, and Industry levels most field values are aggregations of the field values of the nodes from the levels below.

```
<?xml version='1.0'?>
<Report>
  <ReportSpecification
      type="Chain Report"
      name="Manual Orders by Account by Industry"
  >

    <ReportElement
      type="Grouping Tree"
      isEventModelReusable="false"
      purgingLevel="-1"
      reusingLevel="-1"
    >
      <Level grouper="Account">
        <Field id="OrdQty" />
        <Field id="FillQty" />
        <Field id="Leaves" />
        <Field id="NetOrdQty" />
        <Field id="NetFillQty" />
        <Field id="FillVal" />
        <Field id="NetOrdVal" />
        <Field id="NetFillVal" />
        <Field id="NetLeaves" />
      </Level>
      <Level grouper="SubAccount">
        <Field id="OrdQty" />
        <Field id="FillQty" />
        <Field id="Leaves" />
        <Field id="NetOrdQty" />
        <Field id="NetFillQty" />
        <Field id="FillVal" />
        <Field id="NetOrdVal" />
        <Field id="NetFillVal" />
        <Field id="NetLeaves" />
      </Level>
      <Level grouper="Industry">
        <Field id="OrdQty" />
        <Field id="FillQty" />
        <Field id="Leaves" />
        <Field id="NetOrdQty" />
        <Field id="NetFillQty" />
        <Field id="FillVal" />
```

```
  <Field id="NetOrdVal" />
  <Field id="NetFillVal" />
  <Field id="NetLeaves" />
</Level>
<Level grouper="OrdTrnId">
  <Field id="OrdTime" />
  <Field id="OrdStatus" />
  <Field id="Side" />
  <Field id="OrdQty" />
  <Field id="OrdPx" />
  <Field id="OrdType" />
  <Field id="FillQty" />
  <Field id="AvgFillPx" />
  <Field id="Leaves" />
  <Field id="FillVal" />
  <Field id="OrdVal" />
  <Field id="AcctId" />
  <Field id="TrnDest" />
  <Field id="OrdSender" />
  <Field id="Instrument" />
  <Field id="SideType" />
  <Field id="DestID" />
  <Field id="ExecBroker" />
  <Field id="Custodian" />
  <Field id="LastOrdTime" />
  <Field id="ActiveOrdQty" />
  <Field id="NetFillVal" />
  <Field id="NetOrdQty" />
  <Field id="NetFillQty" />
  <Field id="NetLeaves" />
  <Field id="NetOpenOrdVal" />
  <Field id="NetOrdVal" />
  <Field id="OpenOrdVal" />
  <Field id="Symbol" />
</Level>
<Level grouper="OrdRqstId">
  <Field id="MsgType" />
  <Field id="OrdTime" />
  <Field id="OrdStatus" />
  <Field id="Side" />
  <Field id="OrdQty" />
  <Field id="OrdPx" />
  <Field id="OrdType" />
  <Field id="FillQty" />
  <Field id="AvgFillPx" />
  <Field id="Leaves" />
  <Field id="FillVal" />
  <Field id="OrdVal" />
  <Field id="LastFillTime" />
  <Field id="AcctId" />
  <Field id="TrnDest" />
  <Field id="OrdSender" />
  <Field id="Instrument" />
  <Field id="SideType" />
  <Field id="OrigClientOrdId" />
  <Field id="ClientOrdId" />
  <Field id="SenderID" />
  <Field id="DestID" />
  <Field id="ExecBroker" />
```

```
        <Field id="Custodian" />
        <Field id="AckRec" />
        <Field id="RecErrors" />
        <Field id="OrdRec" />
        <Field id="1stFillRec" />
        <Field id="1stFillTime" />
        <Field id="LastFillRec" />
        <Field id="LastOrdTime" />
        <Field id="ActiveOrdQty" />
        <Field id="NetFillVal" />
        <Field id="NetOrdQty" />
        <Field id="NetFillQty" />
        <Field id="NetLeaves" />
        <Field id="NetOpenOrdVal" />
        <Field id="NetOrdVal" />
        <Field id="OpenOrdVal" />
        <Field id="IsOrdActive" />
        <Field id="Symbol" />
        <Field id="ConnName" />
      </Level>
    </ReportElement>
  </ReportSpecification>

  <ReportReferenceList>
    <ReportReference
       reportName="Single Order Domain Tree Report"
       managerNames="Single Order Domain Manager"
    />
  </ReportReferenceList>

</Report>
```

Appendix D: Getting Level 2 quotes through API

When Level 2 information is needed in external application or inside the analytic module the *getCustomDataRecord(String dataSouceName, String recordId)* call has to be used. For example, to obtain level 2 quotes for IBM the following sequence of steps should be performed:

1. Call ***getCustomDataRecord("QuoteMontageFacility", "IBM")***
   This call will result in the system subscribing to the level 2 IBM quotes from the market data feed and it will return the "master" record for the level 2 IBM info.

2. The master record will contain a field with the list of all "child" records for IBM from different sources. Call ***record.getStringFieldValue(ITMSConstants.RDFIELD_CHILD_RECORD_ID_LIST)*** to obtain the comma-separated list of child record names. For example, for IBM this may look like this: "IBM[XNYS],IBM[ARCA],IBM[ISLD],IBM[BRUT]".

3. Call ***getCustomDataRecord("QuoteMontageFacility", "IBM[source]")*** to obtain quote from each individual source. Use the child record names from the list obtained form the master record.

NOTE: when ***getCustomDataRecord()*** call is made the record returned may not contain the values because the delay between the subscription call and the initial record update from the market data feed. Therefore the calls may have to be made several times before the necessary values are obtained. To alleviate this problem the TMS may be instructed to pre-subscribe for the set of records on start-up.

Simply populate file [TMS_HOME]/backend/Directory/ElTrader/TMS/Config/MarketDataQuoteMontageSymbolsCache.cfg with the list of master record ids and make sure the configuration of the quote montage facility references this list --- add to file [TMS_HOME]/backend/Directory/ElTrader/TMS/Config/MarketQuoteMontageProviders.xml:

```
<RecordFacilityProviderList
     …

cacheResourceFile="/ElTrader/TMS/Config/MarketData/QuoteMontageSymbolsCache.cfg"
>
```