CONFIDENTIAL

Application Programming Interface

Version 9.0.2

CBOE API Volume 2 CMi Programmer's Guide to Interfaces and Operations

Programmer's Guide to Interfaces and Operations for the CBOE Market Interface (CMi)

CBOE PROPRIETARY INFORMATION

15 July 2011

Document #[API-02]

Front Matter

Disclaimer

Table of Contents

Change Notices

The following change notices are provided to assist users of the CMi in determining the impact of changes to their applications.

Date	Version	Description of Change
15 Jul 2011	V9.0.2	New section for CMi V10 interfaces
29 Apr 2011	V9.0.1	Updated the Quote Processing Enhancement section
14 Jan 2011	V9.0	New section for CMi V9 interfaces
05 May 2010	V7.0	Noted the ExchangeFirm as a required field for deleting a MMHH trade.
23 Mar 2010	V7.0	Updated cmiCallbackV5 in the CMi V8 section of this document
05 Feb 2010	V7.0	Add a new section for CMi V8 interfaces
08 Jan 2010	V7.0	New section for CMi V7 interfaces for synchronous order enter and short sale marking
07 Aug 2009	V6.1	Updated the Contingency table with BID_PEG_CROSS=29 and OFFER_PEG_CROSS=30
13 May 2009	V6.0	-New section for CBOE 2 (C2) -Updated the AIM section to include Directed AIM -New section for Market Maker Hand Held functionaltiy
21 Nov 2008	V5.3	Added the section: Quote Processing Enhancements Modified the sections: -Recommended Market Making Guidelines. -Quotes -Designing a Quote Application -QRM
03 Oct 2008	V5.2	Modified the Auction Event section to contain information about the extentions string in the AuctionStruct
23 Jul 2008	V5.1	Added a section for changes to restricted series cancels
24 Apr 2008	V5.0	Added a new section for Too Late to Cancel scenario

Date	Version	Description of Change
07 Mar 2008	V5.0	Added a new section for AIM AON
29 Feb 2008	V5.0	New sections for CMi V5:
		-CMi V5 Quote Functionality
		-Internalization Complex Order Entry
		-Retrieve the Current Rate Limit
18 Jan 2008	V4.2.4	Section for Remote TransactionTimeout exceptions (RTT)
		Updated the SAL functionality for non-hybrid index classes
		Added two new BilllingTypeIndicators
02 Nov 2007	V4.2.3	New sections for cross-product spreads and drop copies
01 Jun 2007	V4.2.2	New billing type indicators for W_STOCK
		Added new cross order types: CROSS_WITHIN and TIED_CROSS_WITHIN
11 May 2007	V4.2.1	Updated the “Considersation for Internalization” secton to include using A:AIR in the Optional Data field.
20 Feb 2007	V4.2.1	Updated the contingency mapping table for CBSX
15 Dec 2006	V4.2	Added a new section: Recommended Market Making Guidelines
		Included a section for the User Input Monitor (UIM)
08 Sept 2006	V4.1	Updated the Order Contingency table to include order contingency types for Stock
		New user role “Expected Opening Price (EOP)”
25 May 2006	V4.0	Updated documentation for Version 4.0 interfaces
		Referenced document API-08, CMi’s Programmer’s Guide to the Market Data Express (MDX) data feed.
06 Jan 2006	V3.2b	Updated the callback removal error message and created a new section
		Added a new section of CFE Options on Futures (COF)
		Added a new section for auction type AUCTION_ SAL
12 Aug 2005	V3.2	Added single acronym quote and QRM restrictions.

Date	Version	Description of Change
		Included a table of the Market Data Types provided
29 July 2005	V3.2	Added section to describe overlay mode
		Created a new section for PDPM
		Included AUCTION_HAL in the auction section
08 Apr 2005	V3.1a	Documentation errata release
17 Dec 2004	V3.1	Updates for internalization and auctions.
18 Jun 2004	V3.0	Updated with V3 examples and functionality
02 Jun 2004	V2.52	Updated the Interface section
28 Apr 2004	V2.52	New section “Special Consideration for CFE Orders”
06 Feb 2004	V2.63	New cancel/replace section
31 Oct 2003	V2.62	New example for Stock
		New sections for Quote Lock and Quote trigger
29 Aug 2003	V2.61	New quote throttling section.
31 Jul 2003	V2.6	Changes for market linkage and stock.
08 Jul 2003	V2.51	Document updates.
28 Mar 2003	V2.5	Add Stop and Stop Limit order functionality description
14 Mar 2003	V2.5	Support for Hybrid.
24 Jan 2003	V2.1	Production release update to support Linkage P orders.
22 Apr 2002	V2.0	Production Release
27 Feb 2002	V2.0b	Software Development Kit Beta 2
23 Jan 2002	V2.0a	Software Development Kit Beta 1
14 Dec 2001	V2.0	Updated documentation only for Version 2.0.
27 Apr 2001	V1.0b	Added Market Data role information.
16 Mar 2001	V1.0a	Error corrections and updated to reflect that Strategies will not be part of Version 1.0.
15 Jan 2001	V1.0	Production version.
15 Sep 2000	V0.9	Network testable version.
28 Apr 2000	V0.8	Includes revisions to the CMi API since the last update. Refer to the Release Notes for full details.
30 Sep 1999	V0.5	Includes revisions to API since last update.
3 July 1999	V0.4	API is substantially finished.

Date	Version	Description of Change
		Added diagrams that show the various interfaces and the relationship to the message structures used in the CMi.
8 May 1999	V0.3-1	Properly updated API reference sections
		Continued expand description of the API
30 Apr 1999	V0.3	Support for Strategies.
		Support for crossing notification and requests.
		Introduced an administrator interface to support those who administrate and operate trading rooms.
		Version reporting of the API now supported.
		Mandatory heartbeat between client and CAS.
		Trader Interface renamed to OrderEntry.
		Product Definition Interface added to permit definition of strategies and future definition of FLEX style and OTC instruments.
		Simplified market query interface.
		User is now able to change password through the SessionManager interface.

About This Document

Purpose

This document provides details on how to create an application to access CBOE Exchange Services using the CBOE Common Market Interface (CMi).

Intended Audience

Software developers using the CMi to develop applications that use CBOE Exchange Services.

Prerequisites

This document assumes that the reader has a working knowledge of CORBA and one of the programming languages supported by CORBA, such as C++ or Java. See the reference section of a list of books and web sites that can provide you with fundamentals on CORBA. Specifically, you should be familiar with CORBA modules, interfaces, structs, operations, IORs, exceptions, and callbacks.

Also assumed is your understanding of the basic components and architecture of the CBOEdirect electronic trading system as provided in Volume 1: Overview and Concepts.

Related Documents

Document Number	Document Description
Roadmap.doc	CBOE API and CAS Document Road Map
API-01	CBOE API Volume 1: Overview and Concepts
API-03	CBOE API Volume 3: CMi Programmer's Guide to Messages and Data Types
API-04	CBOE API Volume 4: CMi Dictionary of Attributes and Operations
API-05	CBOE API Volume 5: Using CMi with Specific Object Request Brokers
API-06	CBOE API Volume 6: Connecting to the CBOE Network
API-07	CBOE API Volume 7: CBOEdirect Certification and Testing Procedures
API-08	CBOE API Volume 8: CMi Programmer's Guide to Market Data Express (MDX) Data Feed
FIX-01	CBOE API FIX Protocol Support Volume 1: Overview
CAS-01	CBOE Application Server Volume 1: Overview and Concepts
CAS-02	CBOE Application Server Volume 2: CBOE Application

Document Number	Document Description
	Server Simulator for Stand Alone Testing

Programming Examples—Notation and Location

All programming examples are specified using Java package notation. For instance, the first example program is referenced in the documentation as:

com.cboe.examples.example1.example1.java

This example program can be found in the following directory for Java:

${INSTALL_DIR}\Examples\src\javapoa\com\cboe\examples\example1\example1.java

Where:

${INSTALL_DIR} is the install directory you selected during the installation of the CBOE Software Development Kit. The default location is *C:\CBOE\CMi4.0.*

The C++ version of the example program can be found in:

${INSTALL_DIR}\Examples\src\cpppoa

Note that all C++ code is kept in the same directory.

Scripts and IDL compilation for specific ORBS is provided in the Platforms directory, which is located in:

${INSTALL_DIR}\Examples\orb

Support and Questions Regarding the CBOE APIs

Questions regarding this document can be directed to The Chicago Board Options Exchange at 312.786.7300 or via e-mail: api@cboe.com.

The latest version of this document can be found at the CBOE web site http://systems.cboe.com.

Introduction

The CBOE is adding new interfaces to provide access to exchange services. These interfaces are designed to support both electronic and open outcry trading at the Chicago Board Options Exchange.

The first of these interfaces is an Application Programming Interface (API) that provides access to all exchange trading services and is targeted at firms making markets at CBOE. This API is known as the *CBOE Market Interface* (CMi). CMi is a distributed object interface based upon the CORBA (Common Object Request Broker Architecture) standard from the Object Management Group (OMG). The interface is defined using the Interface Definition Language (IDL), which is an OMG and ISO standard. Messages are transported using the Internet Inter-Orb Protocol (IIOP), which operates over standard Internet protocols (TCP/IP).

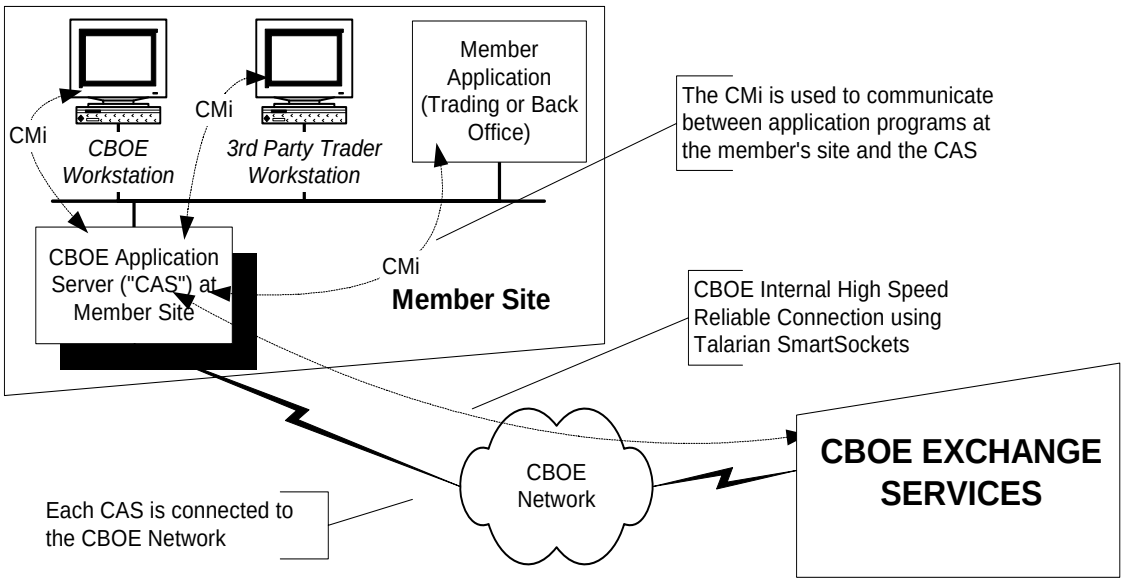
The second interface is a message-based protocol based upon the Financial Information Exchange (FIX) protocol. CBOE's implementation of the FIX protocol should be of particular interest to retail and institutional firms that require order routing to CBOE markets. The FIX protocol is implemented over TCP/IP. FIX is an important messaging protocol in the financial industry. CBOE continues to work with the FIX protocol organization and to participate in FIX working groups to help evolve the FIX protocol for wider use in exchange based derivatives markets.

Introduction to the CBOE Market Interface (CMi)

The CMi is a set of CORBA Interfaces that expose services provided by the CBOE Application Server. Care has been taken to use CORBA as a transport layer, as opposed to a naïve approach of partitioning an object-oriented application's objects across multiple processes and networks. Services are exposed and messages are passed between a client application and the CBOE Application Server (CAS). The messages are in the form of CORBA structs and sequences of CORBA structs.

Additional background on the CMi interface can be found in the API-01 Volume 1: Overview and Concepts.

The following diagram shows the main components involved in the communication between a trading application and the CBOE Exchange.



Getting Started

Choose your development environment

CBOE has adopted the use of CORBA IDL as an interface to exchange services because there are a myriad of alternatives CORBA provides to users, in terms of development environment. Users of CMi are able to select the language that best suits their own needs, they can choose the development and deployment platforms that best fit within their environment, and they are able to use CORBA middleware that best fits their internal needs.

CBOE recommends that you use one of the following languages:

- C++
- Java

Select a Development Platform:

- Microsoft Windows NT 4.0 Service Pack 6 or later
- Sun Solaris Version 2.6 or later

Select a CORBA middleware product:

- Java 1.4 JDK from Sun Microsystems (only for testing, not production)
- omniORB from AT&T Laboratories Cambridge
- Tao Orb from Object Computing or Washington University
- JacORB 2.2.3 from www.jacorb.org

Acquire the CMi software

Retrieve the CMi Software Developer's Kit from the CBOE Internet Web site http://systems.cboe.com. This site requires you to contact CBOE to obtain a User ID and password. The CMi is available in zip and tar formats. The CMi software includes:

1. CMi IDL
2. CAS Simulator
3. Documentation, including this programmer's guide
4. Example programs in Java and C++
5. Scripts for supported development environments

Design your application using this guide

Using this CMi Programmer's Reference and Guide—begin designing your trading application. It is recommended that prior to programming, you should understand:

- the functionality that is available using the CMi
- what functionality is required by your particular trading application

Implement your application using the software developer's kit

Use the CBOE CMi Software Developer's Kit to begin developing your trading application. This is done via the following steps:

- Compile the CMi IDL provided with the CMi Software Developer's Kit using the CORBA compiler that you have selected and procured separately.
- Determine how to integrate CMi within your existing or planned trading application(s). Examples are provided.
- Select the interfaces and operations required to support your trading applications.
- Use the CAS Simulator to test your application in a stand-alone testing mode.
- Follow the procedures for certifying your application as outlined in API-01: Volume 1: Overview and Concepts.

Using the CMi

The CMi is composed of a set of CORBA modules. There are a four types of modules provided with CMi

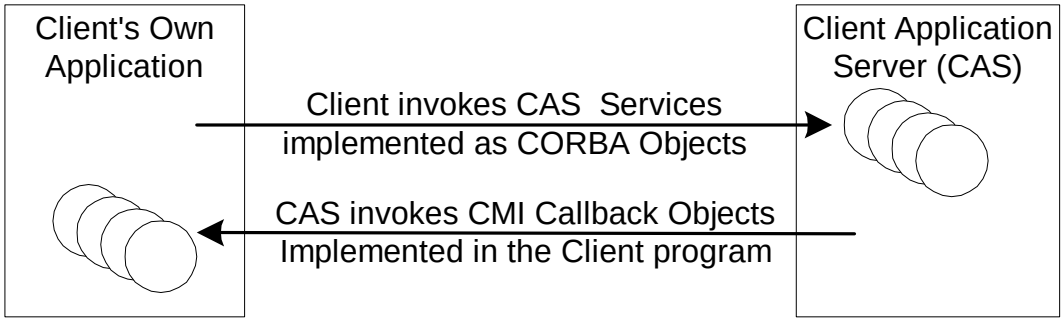
- The first type of module contains the interfaces and operations (or methods) provided or required by the CBOE Application Server (CAS).
- The second are CORBA structs that are the messages that are passed between the client and the CAS.
- The third are modules containing constant values that are defined in the CMi.
- The fourth are modules that contain interfaces for exceptions and error codes.

Definition: A CORBA Operation is a function or behavior that is provided by a distributed object. An operation is the CORBA terminology for an object oriented method.

It is best to start with the interfaces provided in the CMi to understand the functionality provided by the CAS through the CMi. The operations, provided by these interfaces, constitute the functionality of the CAS. The interfaces and operations are contained in the `cmi.idl` and `cmiCallback.idl` files.

Interfaces

The programming model provided through the CMi interface supports access to services that are implemented as CORBA objects on the CAS and CORBA callback objects that you implement in your application program. These callback objects are registered with the CAS via operations available in the various CMi interfaces.



Communication between the CAS and the client programs is done by passing messages. These messages are implemented as CORBA structs. The CORBA struct is very similar to a struct in C or C++. Anyone familiar with any higher level language (C, C++, COBOL) should be able to read and understand the structs (or messages) that are used with the CMi. The structs for a particular interface are placed in CORBA modules. The CORBA module name that contains the structs are named to correspond to the interfaces provided by the CMi. For instance the *Quote interface* primarily operates on structs from the *cmiQuote module* and the *OrderEntry* and *OrderQuery interfaces* primarily reference structs defined in the *cmiOrder module*.

Operation Naming Conventions

A consistent naming convention has been used in naming operations. The following naming conventions have been used throughout the CAS.

Operation Prefix	Behavior
get* Example: cmi::ProductQuery::getProductClasses()	Query a CAS service for information. Information is returned as part of the call – often as a sequence of CORBA structs. Some get operations require the client to provide a callback (or consumer) object to receive subsequent updates to the information that was requested.
accept* Example: cmi::OrderEntry::acceptOrder() cmi::Quote::acceptQuote()	Accept operations are used to deliver information to an interface. CAS Services have `accept*` operations to accept messages from the client. Client callbacks have operations to accept messages from the CAS.
Subscribe* Example: cmi::MarketQuery::subscribeTicker()	Subscribe operations permit the client to subscribe to information to be published by the CAS. The subscription operation always requires a callback object to be provided as an argument. The CAS will call an `accept*` operation on the callback object to provide the information that was subscribed for by the client. *NOTE: You must register your callback with the client ORB to receive information.*
Unsubscribe* Example: cmi::Quote::unsubscribeTicker()	Unsubscribe operations permit the client application to unsubscribe for information that was previously subscribed for using either a get operation or a subscribe operation.
Cancel* Example: cmi::Quote::cancelAllQuotes()	Cancel is used to cancel a previous operation (usually an accept operation), such as canceling an order or a quote.

CMi Interfaces

The CAS provides the following interfaces to the client. Not all services are available to all users. The availability of service is defined by the user role. The interfaces defined by the CAS are defined in the *cmi module* (file: `cmi.mdl`).

Interface	Description
cmi::Administrator	The Administrator Service provides access to the CAS Administrator object that will be used to provide access to Adminstration and operational information. This service provides an operation to send an e-mail type message to the exchange.
cmi::MarketQuery	The Market Query Service provides access to market data, market data history (see the list of operations for the full list of available) for market data retrieval. The query methods (signified by the prefix get) are used to retrieve data in snapshots - the currently available information being returned upon invocation. The MarketQuery service also is used to subscribe to market data, such as ticker data (last sale), recap (last sale and details: high, low, close, open interest, volume), current market (top of the book), and book depth. This interface is not allowed for new development. The MarketQueryV2 interface should be used instead.
cmi::OrderEntry	The Order Entry Service accepts orders, updates to orders and requests to cancel orders. The interface also provides the ability to submit request for quotes.
cmi::OrderQuery	The Order Query Service provides multiple methods for the retrieval of orders.
cmi::ProductDefinition	The Product Definition Service provides the ability to request the system to create new products and complex instruments, such as option strategies (spreads, combos, etc.)
cmi::ProductQuery	The Product Query service provides for the retrieval of product and class information.

Interface	Description
cmi::Quote	The Quote Service is used to submit and cancel quotes. Quotes can be submitted individually or in mass by class. This interface is not allowed for new development. The QuoteV2 interface should be used instead.
cmi::TradingSession	The Trading Session Service provides users with information on trading sessions at the exchange, including the classes that are traded during that session. The service provides the ability to subscribe for updates to the state of the trading session.
cmi::UserAccess	The UserAccess Service provides the capability to logon to the CAS. This object is accessed across the network by making an HTTP Get query to the CAS to retrieve the object reference (as a stringified IOR). A successful logon to the CAS will result in an object reference to the client's UserSessionManager Service running on the CAS being returned.
cmi::UserHistory	The UserHistory Service retrieves trader activity for a specified period of time. USE OF THIS INTERFACE IS NOT REQUIRED TO TRADE ON CBOE MARKETS.
cmi::UserPreferenceQuery	The User Preference Query Service provides configuration information for the system and the user.
cmi::UserSessionManager	The User Session Manager Service provides operations to return object references to the other services provided by the CAS. The UserSessionManager also returns information on the current logged in user and provides an operation to change the password for the currently logged in user.
cmi::UserTradingParameters	Interface for users to set and get the risk parameters for trading.
cmi::Version	Returns the version of the IDL that was used to create the CAS.

V2 Interfaces

Interface	Description
cmi::MarketQuery	The Market Query Service provides access to market data, market data history (see the list of operations for the full list of available) for market data retrieval. The query methods (signified by the prefix get) are used to retrieve data in snapshots - the currently available information being returned upon invocation. The MarketQuery service also is used to subscribe to market data, such as ticker data (last sale), recap (last sale and details: high, low, close, open interest, volume), current market (top of the book), and book depth. All of the Market Query V2 (subscribing to market data) is optional for all users, but if any market query is used at all, it must be from the V2 interface, instead of the non-V2 interface. The only exception is that subscribeExpectedOpeningPriceForClassV2 or subscribeExpectedOpeningPriceForProductV2 is required for options DPMs.
cmi::OrderQuery	The Order Query Service provides multiple methods for the retrieval of orders. If any Order Query is used at all, it must be from the V2 interface, instead of the non-V2 interface. These messages are required for order routing. They are not required for market-makers who enter quotes only.
cmi::Quote	The Quote Service is used to submit and cancel quotes. Quotes can be submitted individually or in mass by class. If any quote messaging is uses al all, it must be from the V2 interface, instead of the non-V2 interface. These messages are required for market-making.
cmi::UserAccessV2	The UserAccess Service provides the capability to logon to the CAS. This object is accessed across the network by making an HTTP Get query to the CAS to retrieve the object reference (as a stringified IOR). A successful logon to the CAS will result in an object reference to the client's UserSessionManager Service running on the CAS being returned.

Interface	Description
cmi::UserSessionManagerV2	The User Session Manager Service provides operations to return object references to the other services provided by the CAS. The UserSessionManager also returns information on the current logged in user and provides an operation to change the password for the currently logged in user.

V3 Interfaces

Interface	Description
cmi::MarketQuery	The Market Query Service provides access to market data, market data history (see the list of operations for the full list of available) for market data retrieval. The query methods (signified by the prefix get) are used to retrieve data in snapshots - the currently available information being returned upon invocation. The MarketQuery service also is used to subscribe to market data, such as ticker data (last sale), recap (last sale and details: high, low, close, open interest, volume), current market (top of the book), and book depth. The CMi V3.0 market query provides new subscription methods to access customer and/or professional size at the Top of the Book. Refer to API-02 for details on V3.0 functionality.
cmi::OrderEntry	The Order Entry Service accepts orders, updates to orders and requests to cancel orders. The interface also provides the ability to submit request for quotes.
cmi::OrderQuery	The Order Query Service provides multiple methods for the retrieval of orders. If any Order Query is used at all, it must be from the V2 interface, instead of the non-V2 interface. These messages are required for order routing. They are not required for market-makers who enter quotes only.
cmi::Quote	The Quote Service is used to submit and cancel quotes. Quotes can be submitted individually or in mass by class. Refer to API-02 for details on V3.0 functionality.

Interface	Description
cmi::UserAccessV3	The UserAccess Service provides the capability to logon to the CAS. This object is accessed across the network by making an HTTP Get query to the CAS to retrieve the object reference (as a stringified IOR). A successful logon to the CAS will result in an object reference to the client's UserSessionManager Service running on the CAS being returned. The User Access V3 object will be made available as an alternative IOR link on the HTTP port that the CAS is publishing on.
cmi::UserSessionManagerV3	The User Session Manager Service provides operations to return object references to the other services provided by the CAS. The UserSessionManager also returns information on the current logged in user and provides an operation to change the password for the currently logged in user. Reference to the User Session Manager V3 can be obtained through login using the User Access V3 interface.

V4 Interfaces

Interface	Description
cmi::MarketQuery	The Market Query Service provides access to market data, market data history (see the list of operations for the full list of available) for market data retrieval. The query methods (signified by the prefix get) are used to retrieve data in snapshots - the currently available information being returned upon invocation. The MarketQuery service also is used to subscribe to market data, such as ticker data (last sale), recap (last sale and details: high, low, close, open interest, volume), current market (top of the book), and book depth. The CMi V4 MarketQuery interface is required for users who want to subscribe to the MDX data feed.

Interface	Description
cmi::UserAccessV4	The UserAccess Service provides the capability to logon to the CAS. This object is accessed across the network by making an HTTP Get query to the CAS to retrieve the object reference (as a stringified IOR). A successful logon to the CAS will result in an object reference to the client's UserSessionManager Service running on the CAS being returned. The User Access V4 object will be made available as an alternative IOR link on the HTTP port that the CAS is publishing on.
cmi::UserSessionManagerV4	The User Session Manager Service provides operations to return object references to the other services provided by the CAS. The UserSessionManager also returns information on the current logged in user and provides an operation to change the password for the currently logged in user. Reference to the User Session Manager V4 can be obtained through login using the User Access V4 Interfaces.

V5 Interfaces

Interface	Description
cmi::OrderEntry	The Order Entry Service for V5 accepts strategy orders for internalization.
cmi::Quote	The V5 Quote Service is used to cancel quotes and to specify if a cancel report is needed.
cmi::UserAccessV5	The UserAccess Service provides the capability to logon to the CAS. This object is accessed across the network by making an HTTP Get query to the CAS to retrieve the object reference (as a stringified IOR). A successful logon to the CAS will result in an object reference to the client's UserSessionManagerV5 Service running on the CAS being returned.
cmi::UserSessionManagerV5	The User Session Manager Service provides operations to return object references to the other services provided by the CAS. The UserSessionManager also returns information on the current logged in user and provides an operation to change the password for the currently logged in user. Reference to the User Session Manager V5 can be obtained through login using the User Access V5 interface.

Interface	Description
cmi::UserTradingParameters	Interface for users to set and get the risk parameters for trading.

V6 Interfaces

Interface	Description
cmi::FloorTradeMaintenanceService	This interface is used to add, delete, subscribe and unsubscribe for trading floor trades (i.e. Market Maker Hand Held trades).
cmi::OrderQuery	The cmiV6 Order Query interface is used to participate in the Directed AIM auction process. It references cmiV3::OrderQuery. These messages are required for order routing. They are not required for market-makers who enter quotes only.
cmi::ProductQueryV6	The Product Query service provides for the retrieval of product and class information.
cmi::UserAccessV6	This interface is used to logon to the CMi V6 module.
cmi::UserSessionManagerV6	References the FloorTradeMaintenanceService.

V7 Interfaces

Interface	Description
cmi::OrderEntry	The Order Entry Service for V7 accepts orders that supports both a synchronous call and an asynchronous call.
cmi::Quote	The V7 Quote Service is used to submit both single quotes and mass quotes. This method returns an asynchronous acknowledgement report.
cmi::UserAccessV7	This interface is used to logon to the CMi V7 module.
cmi::UserSessionManagerV7	The User Session Manager Service provides operations to return object references to the other services provided by the CAS. The UserSessionManager also returns information on the current logged in user and provides an operation to change the password for the currently logged in user. Reference to the User Session Manager V7 can be obtained through login using the User Access V7 interface.

V8 Interfaces

Interface	Description
cmi::TradingClassStatusQuery	This interface provides the list off all trading groups and the list of classes traded in each group. It also has two different subscriptions for trading class status. The subscription methods take a callback, CallbackV5, to notify clients of any outage.
cmi::UserAccessV8	This interface is used to logon to the CMi V8 module.
cmi::UserSessionManagerV8	The User Session Manager Service provides operations to return object references to the other services provided by the CAS. The UserSessionManager also returns information on the current logged in user and provides an operation to change the password for the currently logged in user. Reference to the User Session Manager V8 can be obtained through login using the User Access V8 interface.

V9 Interfaces

Interface	Description
cmi::OrderEntry	The Order Entry Service for V9 accepts light orders and cancel requests for light orders.
cmi::UserAccessV9	This interface is used to logon to the CMi V9 module for light order entry.
cmi::UserSessionManagerV9	The User Session Manager Service provides operations to return object references to the other services provided by the CAS. The UserSessionManager also returns information on the current logged in user and provides an operation to change the password for the currently logged in user. Reference to the User Session Manager V9 can be obtained through login using the User Access V9 interface.

V10 Interfaces

Interface	Description
cmi::OrderEntry	The Order Entry Service for V10 accepts cancel replace requests for light orders.

Interface	Description
cmi::Quote	The V10 Quote Service is used to subscribe for and unsubscribe from receipt of quote fill messages.
cmi::UserAccessV10	This interface is used to logon to the CMi V10 module for light order cancel replace and quote fill subscription changes.
cmi::UserSessionManagerV10	The User Session Manager Service provides operations to return object references to the other services provided by the CAS. The UserSessionManager also returns information on the current logged in user and provides an operation to change the password for the currently logged in user. Reference to the User Session Manager V10 can be obtained through login using the User Access V10 interface.

The CAS objects listed above are made available via the following steps. The *UserAccess Interface* permits you to login. A successful login request via the *UserAccess Interface* results in the return of the reference to the *UserSessionManager Interface* for your application. You then use the *UserSessionManager* to access references to the remaining interfaces provided on the CAS.

CMi Callback Interfaces

The CORBA Callback mechanism is used to implement a publish-subscribe mechanism between the CAS and your program. There are two common approaches to subscribing for information from the CAS.

The first is using an explicit subscribe method available via one of the CAS interfaces. For instance, if you want to subscribe for ticker data for a particular product, the following operation from the MarketQuery interface would be invoked:

```
MarketQuery::subscribeTicker(sesionName,productKey,clientListener);
```

Where:

`sessionName` is a `cmiSession::TradingSessionName` that specifies from which trading session the ticker data should be sent.

`productKey` is a `cmiProduct::ProductKey` that specifies the product whose ticker data you want to receive.

`clientListener` is the `Callback::CMITickerConsumer` object you have implemented within your program to process ticker information. The operation that the CAS will invoke on your `CMITickerConsumer` is:

```
acceptTicker(ticker)
```

Where:

`ticker` is a `cmiMarketData::TickerStructSequence`

which is a sequence of `cmiMarketData::TickerStructs`

This pattern for subscription to information provided by the CAS is frequently repeated.

To subscribe to information from the CAS using a subscribe operation available through one of the CAS interfaces, you must:

- Implement the appropriate CORBA Callback object in your client application.
- **Register that callback with the CAS by invoking the appropriate subscribe operation on the appropriate CAS interface.**
- Implement the appropriate accept operation in the CORBA Callback object

The second pattern implemented for subscription by the CMi is the use of a query operation that returns a snapshot of information and simultaneously registers a client callback object to receive subsequent updates published by the CAS.

For instance, if you wanted to retrieve current market data for a product, you would invoke the following operation on the `ProductQuery` interface:

```
ProductQuery::getProductsForSession(sessionName, classKey,
includeActiveOnly, clientListener)
```

which returns a sequence of cmiProduct::ProductStructs

Where:

`sessionName` is a `cmiSession::TradingSessionName` that specifies from which trading session the list of products eligible for trading is to be retrieved.

`classKey` is a `cmiProduct::ClassKey` that indicates the class whose products (series) are to be returned.

`clientListener` is a `cmiCallback::CMIProductStatusConsumer` callback object that you implemented in your client program to accept updates to product status that are published by the CAS.

Which interfaces you implement will depend on what information to which you chose to subscribe or are required to subscribe.

Callback Performance Design Issues

Threading is an extremely important aspect of callbacks. Instances of callback objects, e.g. CMIOrderBookConsumer, are registered with the CMi to receive status and market data, e.g.

> MarketQuery.subscribeBookDepth(in cmiSession::TradingSessionName sessionName, in cmiProduct::ProductKey productKey, in cmiCallback::CMIOrderBookConsumer orderBookConsumer)

The CAS guarantees that for a given callback object, messages will be delivered in the order that they are received (by the CAS). To accomplish this, the CAS maintains a queue for each callback object registered, and delivers each message after the client has successfully processed the previous message. Each queue will be processed independently.

In theory, the client could use the same callback object for each product or class-based subscription. For example, the client application could instantiate a callback object, and subscribes for products for the classes IBM, AOL, and GE using this callback. If the callback in the client application is activated with a multithreaded POA, it has the ability to handle concurrent calls for all three classes. However, since the CAS sees this as a single callback, and associates the callback with a single queue, the calls would actually be serialized in the CAS. This is a particularly bad choice for high volumes of messages, especially as will be found in options trading.

The solution for this problem is to balance the load across multiple callback objects. Two different ways of accomplishing this might be to use a pool of callback objects or to have a distinct callback object per product class. It should be noted that each user session has a limited number of threads associated with it. If the CAS is trying to service too many callback objects it is possible for some high volume messages to become thread starved. CBOE suggest that you subscribe to market data by class for recap and current market. Note that book depth must be by product (not class) and ticker should be subscribed by class for futures and options by class or by product for underlying. Also note that CBOE currently does not permit book depth subscriptions for options. CBOE's recommendation currently is to use a callback object per product class and message type. This would allow the CAS to invoke the callbacks on distinct threads, without the overhead of managing an excessive number of queues or the delay of one subject's messages behind another subject's messages.

Creating multiple callback objects in the client application does not necessarily mean creating a large number of servant objects. The CORBA object model clearly separates the concept of an object reference from that of a servant. Thus, multiple references could be created by the client, but associated with a single servant, or set of servants as long as the servants are thread safe and don't inadvertently synchronize threads against each other:

```
CMIOrderBookConsumer callbackServant = new CMIOrderBookConsumerImpl();

for ( int i = 1 ; i <= NUM_CLASSES ; i++ ) {

        byte[] servantId = ("Callback" + i).getBytes(); // create a new ID
                           myPOA.activate_object_with_id(servantId, allbackServant);

                           }
```

Given a high rate of messages, a user will need to design to allow for a configurable number of callback objects. The specific number will most likely be determined during your integration and load testing. The key points are:

The CAS associates a thread with each unique callback.

An excessive number of threads can affect performance due to the overhead of managing the threads

Too few callbacks can affect performance due to the serialization of messages to a single callback.

If your implementation associates multiple callback objects to a single servant instance, it is important that the servant be thread safe and not contain any code that could inadvertently synchronize the threads against each other.

When you register a callback object you become a CORBA server. It is important to make sure that your Orb has been configured with enough threads in its POA threadpool

to service all the incoming messages. One performance aspect often overlooked is the available CORBA ORB resources. Depending on your ORB the default setting can be limiting. Review your ORB documentation for limitations with regard to thread usage and adjust the settings to your use. While we do not provide a performance lab it is simple enough to create a simulator to drive your ORB (and your own application) to assist in the tuning.

Another consideration is to limit the processing on each callback thread to the absolute minimum. The faster a callback returns the faster the CAS can send the next message for that subscription. The CAS message performance is directly related to how efficient the client listeners are.

As we have stated previously, we suggest using a unique callback per class wherever possible. Market data, Quote and Order Status messages are subscriptions where a class level object design still results in a fairly manageable number of threads. While this can result in 2000 plus threads in the options or equity world any other allocation will potentially result in the queuing of one classes messages behind another's. A case can be made for using fewer threads when dealing with low volume classes but that requires manual adjustments as the market changes.

One of the features provided is a Queue Depth parameter on the callbacks. This value will inform you of the depth of any queue your CAS server currently has for your process. You will be able to set a threshold and an action to take to eliminate the queue.

Oneway CORBA Calls for V4.0

The cmiCallbackV4 callbacks are used for CBOE's Market Data Express (MDX) data feed. The callbacks are implemented as oneway CORBA calls. These are high-performance calls that function differently than a standard CORBA call. Normally a CORBA function call will wait for the full round trip to complete before continuing. A oneway CORBA call will not wait for any acknowledgement from the server before continuing.

The exact effect this has on your application will depend on your ORB's POA threading policy. If your POA is single-threaded then things will work as they were before with calls queuing up on a single thread. If your POA is multi-threaded then there is no guarantee what order the calls will be received. A sequence number is provided in the method signatures as an aid to determine whether a message was received out of order. Regardless of the threading policy it is essential to keep track of the current sequence number and compare incoming messages to it.

Refer to document API-08, the CMi Programmer's Guide for the Market Data Express (MDX) Data Feed, for complete details on the use of oneway CORBA calls for MDX.

CMi Callbacks
CmiCallback::CMIClassStatusConsumer Callback interface to receive class status for classes trading during a specific trading session.

CMi Callbacks
CmiCallback::CMICurrentMarketConsumer Callback interface to receive current market data.
CmiCallback::CMIExpectedOpeningPriceConsumer Callback interface to receive expected opening price data.
CmiCallback::CMINBBOConsumer Callback to receive National Best Bid and Offer (NBBO)
CmiCallback::CMIOrderBookConsumer Callback interface to receive order book information.
CmiCallback::CMIOrderBookUpdateConsumer Callback interface to receive updated order book information.
CmiCallback::CMIOrderStatusConsumer Callback interface to receive order status for all order activity, including fills, cancels, queries, busts.
CmiCallback::CMIProductStatusConsumer Callback interface to receive product status
CmiCallback::CMIQuoteStatusConsumer Callback interface to receive order status
CmiCallback::CMIRecapConsumer Callback interface to receive recap information.
CmiCallback::CMIRFQConsumer Callback interface to receive request for quotes
CmiCallback::CMIStrategyStatusConsumer Callback interface to receive option strategies as they are created on the system.
CmiCallback::CMITickerConsumer Callback interface to receive ticker information

CMi Callbacks
CmiCallback::CMITradingSessionStatusConsumer Callback interface to receive changes in trading session status.
CmiCallback::CMIUserSessionAdmin The CMIUserSessionAdmin interface is required to be implemented in all client applications accessing the CAS. There are four operations that must be implemented. The first is acceptHeartBeat(), which is used by the CAS to actively monitor the state of the connection between the CAS and the client. The second operation is acceptLogout() that is used by the CAS to request the client application to logout. The third, acceptTextMessage() is used by the trading system and trading operations personnel to send messages to users of the system. The fourth, acceptAuthenticationNotice() is used by the CAS to request that the client reauthenticate. This is used during periods of inactivity. This is a security mechanism that is intended primarily for interactive applications. Once the acceptAuthenticationNotice() is received, the client must respond within a short amount of time with an invocation of the UserSessionManager.authenticate(userLogonStruct) message on the CAS.

V2 Callback Interfaces

CMi Callbacks
CmiCallback::CMICurrentMarketConsumer Callback interface to receive current market data.
CmiCallback::CMIExpectedOpeningPriceConsumer Callback interface to receive expected opening price data.
CmiCallback::CMILockedQuoteStatusConsumer Callback interface to receive locked quote status.
CmiCallback::CMINBBOConsumer Callback to receive National Best Bid and Offer (NBBO)
CmiCallback::CMIOrderBookConsumer Callback interface to receive order book information.
CmiCallback::CMIOrderBookUpdateConsumer Callback interface to receive updated order book information.

CMi Callbacks
CmiCallback::CMIOrderStatusConsumer Callback interface to receive order status for all order activity, including fills, cancels, queries, busts.
CmiCallback::CMIQuoteStatusConsumer Callback interface to receive order status.
CmiCallback::CMIRecapConsumer Callback interface to receive recap information.
CmiCallback::CMIRFQConsumer Callback interface to receive request for quotes.
CmiCallback::CMITickerConsumer Callback interface to receive ticker information.

V3 Callback Interfaces

CMi Callbacks
CmiCallback::CMIAuctionConsumer Callback interface to receive auction data.
CmiCallback::CMICurrentMarketConsumer Callback interface to receive current market data.

V4 Callback Interfaces

CMi Callbacks
CmiCallback::CMICurrentMarketConsumer Oneway CORBA callback interface to receive current market data
CmiCallback::CMIRecapConsumer Oneway CORBA callback interface to receive recap and last sale information.

CMi Callbacks
CmiCallback::CMITickerConsumer Oneway CORBA callback interface to receive ticker information.

V5 Callback Interfaces

CMi Callbacks
CmiCallback::CMITradingClassStratusQueryConsumer Callback interface to receive product group status.

Example of cmiCallbackV3::CMICurrentMarketConsumer::acceptCurrentMarket

cmiCallbackV3::CMICurrentMarketConsumer

acceptCurrentMarket

cmiMarketData::CurrentMarketStructSequence bestPublicMarkets

CurrentMarketStruct

MarketVolumeStructSequence bidSizeSequence

MarketVolumeStructSequence askSizeSequence

MarketVolumeStruct

VolumeType volumeType

cmiConstants::VolumeTypes

AON

CUSTOMER_ORDER

FOK

IOC

LIMIT

NO_CONTINGENCY

ODD_LOT

PROFESSIONAL_ORDER

QUOTES

bestMarkets - All orders and all quotes at the top of the book (best bid, best ask). This is exactly what CBOE publishes today already in the old Current Market message. Contingency types will show up here. No origin types will show up here.

bestPublicMarkets - If customer or professional orders are in the top of book, then they'll show up in the bestPublicMarkets. Market-maker quotes and ICM orders will not show up in bestPublicMarkets but they will be included in bestMarkets. Contingency types will not show up here. Origin types will show up here. The only possible constants values in this field will be either:

PROFESSIONAL_ORDER is defined as all order origins other than I, C, and D. Does not include market-maker quotes.

CUSTOMER_ORDER is defined as order origins C and D (D=CUSTOMER_FBW).

Callback Removal Error Message

To allow user's to re-register a single removed callback without logging out and resubmitting all callbacks, CBOE has reworked the callback removal error message.

The error message will now contain a standard IIOP reference. This will allow the user to resolve the individual object being removed and reapply only that object instead of all listener objects. Please note - **we have not changed the IDL.** Only the contents of the fields in the return struct have changed.

The easiest manner in which to test this feature is for the client application to throw an uncaught exception when receiving a callback from the CBOE. Any exception returned to CBOE will force a callback deregistration by the CBOE process.

Existing IDL:

cmiCallback.idl:

```
void acceptCallbackRemoval( in cmiUtil::CallbackInformationStruct
callbackInformation,
                    in string reason,
                    in exceptions::ErrorCode errorCode);
};
```

cmiUtil:

```
struct CallbackInformationStruct
{
  string subscriptionInterface;
  string subscriptionOperation;
  string subscriptionValue;
  string ior;
};
```

CBOE used to send the following information in the callback removal:

subscriptionInterface

Callback interface (example: CMICurrentMarketConsumer)

subscriptionOpereration

The name of the failed method(acceptCurrentMarket)

subscriptionValue "=" + string representation of object (com.cboe.idl.CurrentMarketStruct[] @16d58b)

ior

CBOE ORB IOR object digested.

CBOE will now send:

SubscriptionInterface

Callback object type id (example: ZDL:cmiCallback/CMICurrentMarketConsumer:1.0)

subscriptionOperation

Method name that failed. Example: acceptCurrentMarket

subscriptionValue

Session class and product key information this object was subscribed for delimited by "\u0001" and the IOR.

example: = 69206141^A69220449^A89128963^A69216556

ior

CORBA standard regular ior representation of the CORBA callback object.

The following is a sample of a test application error log:

TestCallback::acceptCallbackDeregistration::subscriptionInterface =

"IDL:cmiCallbackV3/CMICurrentMarketConsumer:1.0"

subscriptionOperation = "acceptCurrentMarket"

subscriptionValue = "69206035_69213922"

ior =

"IOR:000000000000002f49444c3a636d6943616c6c6261636b56332f434d4943757272656e7

44d61726b6574436f6e73756d65723a312e300000000000210ca10000000005800000000000

0000a4348454e4a2d57324b000000789234f70000003c000000010000000b44656661756c745

04f4100b1ebcdcc0000000e0010f6d30000010530abb0f880060000000e0010f6d3000001053

0abb0f880010000000000000000078000102000000000a4348454e4a2d57324b00200a0000003

c000000010000000b44656661756c74504f4100b1ebcdcc0000000e0010f6d30000010530abb

0f880060000000e0010f6d30000010530abb0f880010000000002000000020000000800000000

0000000000000000000000000800000000cb0e0001"

TestCallback::acceptCallbackDeregistration::reason::Connection lost.

TestCallback::acceptCallbackDeregistration::errorCode::1

Heartbeat Callback

CBOE sends the firm's application a heartbeat every 2 seconds. The Orb on which the CAS is now based uses a timeout for heartbeat. Currently this timeout is set to be 20 seconds. This means when a CMi client application doesn't respond to a single heartbeat request within 20 seconds for any reason, the connection will be considered dead and the CAS will logoff the connection and clean up its session. As a result, care must be taken to make sure that the heartbeat thread in your application never becomes starved or isn't serviced or your application may be logged off on a false detection.

Guaranteed Message Delivery (GMD)

The CMi provides an optional guaranteed delivery mechanism for messages that are critical in the trading process. A GMD type message is a message that will continue to be delivered until it has reached its final destination. A GMD consumer is considered a final destination for a GMD message.

Messages for which GMD is available

A GMD flag has been added to the following callback interfaces:

- Text Messaging
- New Order
- Order Fill Reports
- Order Cancel Reports
- Order Bust Reports
- Order Bust with Reinstate Reports
- Quote Fill Reports
- Quote Bust Reports

GMD Usage

- Interfaces that support GMD have an additional parameter on the subscription method for a boolean flag to indicate if GMD should be used on that callback object.
- If the GMD flag is set to false the CAS will not acknowledge any messages delivered to the CBOEdirect server.
- If the GMD flag is set to true, the CAS will acknowledge each message delivery to server.
- The order status NEW message will no longer by GMD. If a CMi user is disconnected for any reason, at re-login the user will not receive the undelivered NEW order status messages.
- The order status message with the BOOKED state will no longer be published. The state of a new order will be reflected in the copy that is returned in the acceptNewOrder message. For all new order messages, the state will be BOOKED except for STOP orders. So, if an order is fully filled upon entry, the user would

receive a new order message, containing a copy of the order with the state BOOKED and status reason as NEW, and a separate fill report with the state FILLED. If the order is booked, the user would receive a new order message, containing a copy of the order with the state BOOKED. If the order was partially filled, a separate fill report would also be received.

- The GMD flag on the login method is used to indicate whether or not the user wants this particular Admin callback object to be used as the "GMD" consumer for text messaging.

  In the case of text messaging, every single time a user logs on, any messages that have not been delivered to the GMD consumer will be published. The specific impact of this is that any non-GMD consumers will continue to receive all text messages that haven't been officially received by the GMD text messaging consumer ever time that they register. Once a GMD consumer receives a particular message, it will no longer be sent to any text message consumers when they subscribe to the CMi. There may only be one active "GMD" consumer subscribed to the CAS of a given type (text messaging, order, quote) at a time.

  The impact of putting "true" for this flag is that all text messages delivered to the Admin consumer supplied at login will be considered "delivered" by our system and we will never try to send them again. Additionally all subsequent logins under the same ID that try to use "true" for GMD text messaging will be rejected since there may only be one active "GMD" consumer at a time. The impact of using false for this flag is that the consumer will receive all old text messages every time that it is subscribed during login until a GMD flagged consumer has received the message.

- A user is allowed to register multiple GMD consumers for orders and quotes, but can only register one per class if the user is using the methods – subscribeOrderStatusByClass and subscribeQuoteStatusByClass.
- You are not required to use the GMD service. A user does not need to have a GMD consumer registered first in order to register other callback consumers in non-GMD mode.
- A user can have multiple non-GMD consumers.
- GMD can be used with login session that are in either Primary or Secondary mode.

As part of the API testing, we will require one subscription to be a GMD for each type. These primary GMD callbacks may be spread across applications.

Designing a Quote Application

When designing a quote application against CBOEdirect, it is helpful to understand the implications of some design choices, particularly threading choices.

It is important to understand that CBOEdirect is highly distributed. As a result, a given product class can be trading on one of many trade servers. The most important implication of this architecture is that any problem with a particular trade server or product class does not mean that any other product class will necessarily be effected. As a result, it is recommend that you design your system so that problems, such as latency, with a particular product class do not effect your ability to timely quote any other product class. One way to achieve this might be to dedicate threading resources for each each product class being quoted.

Quotes

The quote status subscription now allows multiple GMD callbacks to be registered. Instead of only allowing one GMD callback consumer per user, the CMi now allows the user to specify a separate GMD callback consumer per class. Additionally, the indication that a quote has been cancelled or deleted will no longer be communicated through the quote status message. Instead it will be published to the acceptQuoteDeleteReport method on the new quote status callback consumer.

A new quote callback interface was also added to notify market makers when their quotes were locked against other quotes. This is on a separate interface to allow the user to service this callback independent of the fill messages.

The decision on whether to publish existing quotes on the subscription call to the consumer has been changed to a Boolean parameter called publishOnSubscribe. Finally, a new parameter "includeUserInitiatedStatus" was added to allow the user to indicate to the CAS whether or not to publish the "booked" message through the quote status consumer. A successful call to acceptQuote with no exceptions (or a successful call to acceptQuotesForClass with no exceptions) with ErrorCodes all equal to 0 indicates the quote message has been processed and booked by the server. The booked status message was designed to be used by position management systems, which usually are separate from the quoting application. Set the "includeUserInitiatedStatus" to true if you need the message.

Previously, if a user (one user ID, one user session) sent a number of mass quote or quote messages at the same time to the CAS for the same class, the CAS sent the quotes one at a time to the back-end server in the order in which the CAS received them.–In order to increase peak quote throughput for a given class, CBOE has redefined its quote processing.

Quote Processing Enhancements

CBOE has improved its quote processing time by allowing quotes to be submitted concurrently even for a given class. The concurrent thread model is now the standard. The redefined quote process allows multiple concurrent quote messages up to a configurable limit. Please note that, unless explicitly mentioned, all quote messages referred to in the following section are messages acting on the same trading class. A quote method is any of the following requests : acceptQuote, acceptQuotesForClass, cancelQuote, cancelQuotesByClass and cancelAllQuotes.

The following behavior should be considered by the quoting Firm:

Due to the asynchronous nature of the concurrent quote message calls; it is very important that the same series not be included in multiple quote blocks. While CBOE will not enforce any restriction with regard to this check, the order of processing of the individual calls is not guaranteed. The quote processing changes are as follows:

1. The CAS will allow multiple concurrent quote messages up to a configurable limit. These requests include *acceptQuote and acceptQuotesForClass*. The limit at this time is 10.
2. Immediately before the quote message is to be dispatched the quote rate limits are checked. If the call or quote rate limits are not exceeded the message is dispatched.

3. When the number of concurrent quotes in-flight equals the maximum allowed, any new *acceptQuote* or *acceptQuotesForClass* calls will be **rejected** with a NotAcceptedException.

   The error code in this case will be *EXCEEDS_CONCURRENT_QUOTE_LIMIT (4160*) and error text will say, "*Concurrent Quotes exceeded the limit (<limit>)"*.

4. The cancelQuotesByClass and cancelAllQuotes requests will be forwarded to the server immediately without regard to the number of concurrent quote messages currently in progress.

5. While any cancelQuotesByClass is in flight to the server, any new quote request calls (includes *acceptQuote, acceptQuotesForClass and cancelQuote*) will be **rejected** with a NotAcceptedException.

   The error code in this case will be *QUOTE_CANCEL_IN_PROGRESS (4170*) and error text will say, *"Quote Cancel by class is in progress"*.

6. Block quote cancels (block of 0-0 quotes coming through the acceptQuotesForClass calls) are considered to be a acceptQuotesForClass calls and could be rejected if the number of in-flight quote requests to the server exceed the limit or if a cancelQuotesByClass is in-flight.

7. A cancelAllQuotes request is always forwarded to the server. Due to the asynchronous nature of the cancel all quotes request, the CAS will not prevent any new quotes message or quote cancel message (including another cancelAllQuotes) from being forwarded to server while a previous cancel all quotes request is in progress.

Quote Risk Monitor

The Quote Risk Monitor (QRM) feature is designed to allow the user to limit the risk associated in quoting. Using the *struct QuoteRiskManagementProfileStruct* in the CMi quote interface, the user can set a threshold limit per class after which the CBOE systems will pull all quotes for the user in the class. Once the threshold the user sets is crossed, the QRM process issues system cancel message for all remaining quotes in the class. The QRM cancel message is out of process from the trading process and the cancel actions are best effort. Meaning it is possible for the user to be filled past the set limit.

QRM is setup at the acronym level. Therefore, multiple users that share the same acronym and exchange will be sharing the same QRM values. For example,

- User ID1 and user ID2 are sharing acronym ABC
- User ID1 is quoting IBM
- User ID2 changes QRM on his/her CBOE*direct* trading system for IBM
- User ID1's QRM values will change to reflect the changes made by User ID2

The QRM interface has always supported sequences of status messages in the response but prior versions of the QRM feature have only published single status messages. With the software release planned for January 2009, the QRM responses will contain multiple status messages.

Quote Entry Restrictions with Shared Acronym

Two users sharing the same acronym cannot quote the same class. For example,

(1) User ID1 and user ID2 share the same acronym

(2) User ID1 is quoting class, IBM. User ID2 tries to quote IBM but the quote is rejected. User ID2 will receive the error message **ErrorCode OTHER_USER_FOR_ACR_QUOTING_CLASS = 4150**

(3) If user ID1 logs off and all quotes are cancelled, user ID2 will be able to quote IBM.

(4) If user ID1 cancels all his/her quotes, without logging off, user ID2 will be able to quote IBM.

User Input Monitor (UIM)

This feature is awaiting regulatory approval and is not currently enabled.

CBOE's User Input Monitor (UIM) is a service that will run in the CBOE trade server to monitor user quote times. The UIM service will limit the CBOE's liability for a user who is unable, due to exchange system issues, to update or cancel their market quotes. An additional benefit is to protect Market Makers when their own system problems prohibit them from quoting.

The UIM service is intended to assist in the detection of problems with a user's quote input stream. The CBOEDirect system services discrete collections of underlying symbols per business server group. If one of these server group detects no inbound quotes for a user for any of the group's serviced underlying symbols during the configured time period, it will declare an error condition for the user. This will result in all of the user's quotes in that group to be removed by the systems. If a user is actively quoting in classes serviced by one business server group and stops streaming quotes in another business group there will be no impact to their quotes in the active group. Cancel reports will be sent to the user with the cmiConstant cancel reason, const cmiUtil::ActivityReason NO_USER_ACTIVITY. Refer to the Quote Update Control ID (also called Quote Token) section below for details on how CBOE handles messages that may cross in flight.

UIM behavior at the open

If a user has system problems before (or seconds after) the open, their quotes will be at risk and UIM will not save them. UIM is only effective during the trading hours (*n* seconds after the open and *m* seconds before the close, currently *n*=*m*=30sec).

Details:

The basic UIM behavior is to record a timestamp for each inbound quote or quote block per user per business server group. A background thread will cancel orders and quotes per business server group for users whose most recent business server timestamp is older than *m* seconds.

The timestamp for inbound quotes is *not* recorded for UIM if the current time is less than "*n* seconds after the first series in the class opens". This is to avoid cancelling quotes entered during preopen, when market makers may not be quoting very often.

The side effect of this is that UIM protection is not activated on a business server group for a user until that user has entered a quote for a class that has at least one open series on the business server.

Example:

- User JIM enters an IBM Jul 07 quote at 8:25 (Because this is before any IBM series is open, UIM does not record his quote time.)
- Due to system problems, he is unable to quote but has not logged out.
- IBM opens, and JIM's quote is at risk until he is logged out, *or* until he enters a quote on a class that has an open series.

CMi V3 Quote Functionality

There are several new quote functionalities.

Quote Update Control ID (also called Quote Token)

A new field has been added to the mass quote message to indicate that a new quote submission should be overridden. This is to solve the case when a market maker's quote system is in the process of sending in new quotes at the same time that CBOEdirect is removing the old quotes. The messages may cross in flight and the market maker will have replaced his old quotes. One particular example is QRM. The QRM is intended to protect a market maker from being hit multiple times across a large number of series. In one example, the market maker would be streaming its quotes into the market without knowing that the QRM has triggered. The Quote Update Control ID field will provide the user with the ability to protect against new quotes being processed before the user acts upon the QRM notice.

1. When a market maker begins sending quotes in the morning this new field will be initialized.
2. If the Quote Update Control ID field is set to zero (0) (in CMi, this is cmiConstants::QuoteUpdateControlValues:CONTROL_DISABLED), then the Quote Update Control function will be turned off.. Using this new method will have the exact same behavior as the old mass quote method, i.e. CBOE would not provide any protection against the new quotes when such race condition occurs.
3. When a QRM event or quote cancellation occurs, either user or system initiated, CBOEdirect will look at the value in the control field. CBOE will not accept any new quotes from the user as long as the new Control ID value remains the same as it was before the cancel event.
4. Once the user changes the value on a new quote, that quote will be accepted and that value becomes the new test condition.

CBOE recommends that if the market-maker uses the Quote Update Control ID, then the market-maker should increment the Quote Update Control ID every time the market-maker receives a quote cancel acknowledgement from CBOE (intentional or unintentional cancellation). CBOE discourages the market-maker from incrementing the Quote Update Control ID every time a quote cancel request is sent *and* every time a quote cancel acknowledgement is received from CBOE. The reason for this is that the market-maker would not be protected from the race condition.

In addition to incrementing the Quote Update Control ID when receiving a regular quote cancel confirmation from CBOE, the market-maker must increment the Quote Update Control ID every time a quote entry to CBOE rejects or is canceled with the following error codes:

1. Quote rejected
   User will get NotAcceptedException with error code:
   const exceptions::ErrorCode QUOTE_CONTROL_ID = 4110
2. Quote Cancel Report
   User will get a cancel report with reason:
   const cmiUtil::ActivityReason QUOTE_UPDATE_CONTROL =10;

The Quote Update Control ID is stored at CBOE in a short (16 bit signed integer, -32768 to 32767). Since the ID can be reused during a trading day by the same market-maker, the firm may rollover their numbers in any feasible range desired (1-99, 1-999, etc.).

The Quote Update Control ID must be submitted on a product-by-product basis (one ID per individual option or future, not one ID per class). It is likely that a market-maker will have different IDs for quotes for different options or futures products within the same class at the same time.

An extra protection of quotes from being in the market after the logout will be provided in this release. When CBOEdirect processes the user logout event, if there is any in flight quotes coming into the system, CBOEdirect will reject those quotes or send a cancel report for those quotes. If the quotes are rejected, the returned quote result will contain the AuthorizationCodes. INVALID_SESSION_ID. If the quote cancel report is sent, the cancel reason will be specified as ActivityReasons. **INVALID_SESSION_ID.**

Finally, a new cancelAllQuoteV3 method has been added. The parameter of this method remains the same as the old one. The difference in this new method is proper return of the DataValidationException when the trading session name that is entered is not valid.

CMi V5 Quote Functionality

The existing quote cancel methods: *cancelQuote, cancelQuoteByClass* and *cancelAllQuotes* result in asynchronous quote cancel reports for all the quotes canceled. New aspects of the above quote cancel methods have been added in this release. They take an extra boolean parameter to specify whether a cancel report needs to be sent or not. If set to false, no asynchronous cancel report will be generated for the requested cancel. If set to true, behavior will be same as that of the currently existing cancel methods. This feature will be available in April 2008.

```
interface Quote : cmiV3::Quote
{
   void cancelQuoteV5(in cmiSession::TradingSessionName
sessionName,
                      in cmiProduct::ProductKey productKey,
                         in boolean sendCancelReport)
      raises(
      exceptions::SystemException,
      exceptions::CommunicationException,
      exceptions::AuthorizationException,
      exceptions::DataValidationException,
      exceptions::TransactionFailedException,
      exceptions::NotAcceptedException,
      exceptions::NotFoundException
```

```
      );

   void cancelQuotesByClassV5(
                        in cmiSession::TradingSessionName
                     sessionName,
                        in cmiProduct::ClassKey classKey,
                        in boolean sendCancelReports)
   raises(
      exceptions::SystemException,
      exceptions::CommunicationException,
      exceptions::AuthorizationException,
      exceptions::DataValidationException,
      exceptions::TransactionFailedException,
      exceptions::NotAcceptedException,
      exceptions::NotFoundException
   );

   void cancelAllQuotesV5(in cmiSession::TradingSessionName
sessionName,
                                 in boolean sendCancelReports)
   raises(
      exceptions::SystemException,
      exceptions::CommunicationException,
      exceptions::AuthorizationException,
      exceptions::DataValidationException,
      exceptions::NotAcceptedException,
      exceptions::TransactionFailedException
      );
};
```

Recommended Market Making Guidelines

It is important to follow the market making guildelines listed below to effectively interact with CBOE's Hybrid Trading System.

1. Logout implications. The logout is a relatively expensive function. We have observed Market Makers logging out of the system and immediately logging back. A less severe method of removing quotes would be to send a ***cancel all quotes*** message.
2. Excessive quoting/thrashing. We have observed firms sending in mass quotes, then single cancels, followed by mass quotes again. If there is a perceived reason validating the need for cancels, then they should not be immediately followed by mass quotes. If the intent is to update the quote, then the cancel is superfluous.
3. There are many single quote blocks entered. It is more efficient to block multiple quotes together within a single message, when possible.
4. Status message handling. A minimum of a single thread per class for status callbacks pertaining to market data is recommended. Status messages should be handled as quickly as possible, in order to allow delivery of the next status message. We ask that each firm use ***class level callbacks for order and quote status***. We have observed that connections cannot keep up with order or quote status if it spans multiple classes with a single callback.

Remote Transaction Timeout (RTT) Exceptions

An Addition to CBOE inbound Order and Quote error handling is a quality of service feature wherein the processes will set a maximum service time for an inbound order or quote to process internally within the CBOEDirect system. The intention of this addition is to release control of the inbound processing back to the end user with an indication an error or slow down has occurred without waiting for the situation to clear. While the request will abort with an exception it does not mean the original request will not eventually complete. This is in essence a “maybe” condition and it will require the user to take action to with regard to the inbound quote or order stream at the time of the error.

Specifically, the user should:

1. Take immediate action to compensate for the ambiguous message failure. If a Quote message is involved, cancel the quote(s) in question or log out to cancel any resting quotes.
2. If an order origination request failed, a cancel request should be sent to ensure the order is not in the market before taking an alternate action regarding the order. If an order cancel is in question the cancel request could be resubmitted.

Please contact the CBOE helpdesk to help determine where and what the internal communications involved failed or succeeded if there is any doubt.

Qrder Query

The order status subscription now allows multiple GMD callbacks to be registered. Instead of only allowing one GMD callback consumer per user, the CMi now allows the user to specify a separate GMD callback consumer per class. Additionally the decision on whether to publish existing orders on the subscription call to the consumer has been changed to a Boolean parameter.

Modules that Contain the Message Formats

The following table contains the list of CORBA IDL modules that contain the messages between CAS and client that are implemented as CORBA structs. The CMi interface uses typedefs extensively to minimize maintenance.

Module	Description
cmiAdmin	Types and messages for use in administration of the CAS and Client interface.
cmiIntermarketMessages	Intermarket messages used in the CBOE Market Interface.
cmiMarketData	Order book depth information provides the total volume and contingency volume for the top N(5) best bid and ask prices. Order book depth information is available on a request not on a subscription basis. Access to this call will be limited on a per minute basis and by user type.
cmiOrder	The cmiOrder module contains messages for order

Module	Description
	routing. Message are available for submission of orders, cancel requests, request for quotes, and reports on order activity.
cmiProduct	Messages describing products within the system.
cmiQuote	The cmiQuote module contains structs and datatypes for Quotes and Request for Quotes. Messages are provided for quote status and quote filled reports.
cmiSession	Messages from the TradingSession service. The TradingSession service is used to determine which products trade in a given session. The service is also used to obtain session and product state information.
cmiStrategy	Contains structs and types required to support trading derivative strategies.
cmiTrade	Contains structs and types required to support external trades.
cmiTraderActivity	This is an optional interface - it is not needed for trading on CBOE markets. The cmiTraderHistory module contains messages used to report trader activity. An activity record is generated for each new quote and order and each subsequent activity on a quote or order. TraderHistory information is accessed using the cmi::UserHistory interface. Unlike the other interfaces, there are several different types of history records that are returned via the same UserHistory operation. Information is returned as a sequence of named value pairs. The UserHistory interface is provided to support CBOE applications. It is not necessary as part of a trading application. The UserHistory interface also does not replace existing CBOE interfaces for accessing matched trade information.
cmiUser	User preferences and privileges
cmiUtil	Common utility type used within the CBOE Market Interface

Market Data

The market data interfaces previously allowed the user to register callback objects per class. However not all methods returned sequences of responses. Almost all methods have been modified to return sequences to allow us to take advantage of blocking where appropriate. For instance a single block of quotes from any user can result in multiple updates to the current market quote for several products within a class. We will deliver the market updates in a single message where possible.

The new parameters available in the market data subscriptions are:

Queue Action.

Queue Depth.

A new book depth data struct will be added to allow for more granular information. This information will be available on a request *not* on a subscription basis. Access to this call will be limited on a per minute basis and by user type.

Market Data Types Provided

Market Data Type	Description	Trading Sessions	Request Type	Subscription Type	When Published
Current Market	Top of the book for products trading on CBOE markets. Includes: Best bid price Best ask price Best bid size Best ask size	W_MAIN, ONE_MAIN, CFE_MAIN, W_STOCK	Subscription plus initial snapshot at time of subscription	Class Level (recommended) or Product Level ** A mix of class level and product level subscriptions is not permitted.	Anytime there is a change in Best bid price Best ask price Best bid size Best ask size
Recap	**Options and Futures:** Last trade (last sale) including: Last sale price Last sale quantity Low price High price Total quantity traded on the day **Underlying stock or index:** Best bid price Best ask price Best bid size Best ask size Last sale price Last sale quantity Low price High price Total quantity traded on the day	W_MAIN, ONE_MAIN, CFE_MAIN, W_STOCK	Subscription plus initial snapshot at time of subscription	Class Level (recommended) for options and futures or Product Level for options, futures, or underlying ** A mix of class level and product level subscriptions in one class is not permitted.	**Options and Futures:** Published when there is a trade **Underlying stock or index:** Published when there is a trade *or* when any of the following change: Best bid price Best ask price Best bid size Best ask size
Ticker (Last Sale)	Last Sale for underlying and derivative products traded on CBOE	W_MAIN, ONE_MAIN, CFE_MAIN	Subscription	Product Level Only	Each trade (last sale)

Market Data Type	Description	Trading Sessions	Request Type	Subscription Type	When Published
	markets. Ticker is a subset of Recap. Includes: Last trade price Last trade quantity				
Expected Opening Price	The expected opening price for products traded on CBOE markets. This also announces when order imbalances occur.	W_MAIN, ONE_MAIN, CFE_MAIN	Subscription	Class Level or Product Level** A mix of class level and product level subscriptions is not permitted.	Published on a timer basis before market open
Book Depth	Query or subscription of the book depth for a product traded on a CBOE market	W_MAIN & W_STOCK via FIX, and ONE_MAIN & CFE_MAIN (via CFN only)	Snapshot query or subscription (*Hybrid: 1 Snapshot once every 1second & no subscriptions*)	Product Level Only	Each Time a change occurs to the book
NBBO	*National Best Bid and Offer – best of consolidated options markets*	*Not published at this time*	*N/A*	*N/A*	*N/A*

Underlying Market Data

For underlying market data, if any of the items in a recap message change, INCLUDING the bidPrice, bidSize, askPrice, and askSize, in addition to lastSale price and quantity, a recap is published. There will be no currentMarket messages published for underlying market data. If a firm cares about last sales, it should also subscribe to the ticker, which contains the underlying last sale information that is published when a trade takes place. Occasionally, if a trade is out of sequence, a trade will be published in the form of a ticker and no recap will be published. Indexes that are not actually traded do not have bid/asks, only last sale prices. CBOE publishes the underlying index market data for index classes (such as OEX, S&P500, Dow Jones, NDX etc.). CBOE does not publish the market data for the futures contract that mirrors the index. For example, CBOE does publish the underlying market data for the S&P 500 index, but CBOE does not publish the market data for the S&P 500 futures contract. Note: underlying ticker and recap are published following industry standard practices (CTS/CQS).

** A mix of class level and product level market data subscriptions is not permitted on an individual class. If a user is subscribed to receive IBM market data at the class level and would like to receive market data at the IBM product level, the user must unsubscribe from receiving market data at the class level before subscribing to receive market data at the product level.

For Hybrid market data, the firm must open up a range of IP addresses (subnet) on the firm's firewall and routers as described in the Hybrid Port document available for download at the API web site. This applies in the test and production environments

Market Data V3 Functionality

The existing API provides the top of the book information on the callbacks. V3 functionality provides a public market parameter on the new callbacks. This value will inform users of the Customer and Professional size if, and only if, there is Customer and/or Professional interest at the Top of the Book. To access this information, new subscription methods are provided in the CMi V3 MarketQuery interface.

Market Data V4 Functionality

The new ticker, recap and last sale callbacks now feature a sequence number in the unlikely event that a message arrives out of sequence. The existing recap callback is broken up into two separate callbacks: recap and last sale. New subscription methods are provided in the new MarketQuery interface.

Refer to document API-08, the CMi Programmer's Guide to the Market Data Express (MDX) Data Feed, for complete details on the use of market data for MDX.

Queue Actions

CBOE provides four types of queue actions to manage application backlog.

(1) NO_ACTION - CBOE currently does not support this designation. It will return the same resulting actions as DISCONNECT_CONSUMER.

(2) FLUSH_QUEUE - Allows the end user to continue building large queues on the CAS. Once the queue reaches a certain size, it will automatically be flushed then allowed to rebuild. The queue size is currently configured to 1,000.

(3) DISCONNECT_CONSUMER - Automatically unregisters the callback object when the queue size is exceeded. Currently, the queue size value for the Hybrid CAS is configured to 5,000.

(4) OVERLAY_LAST - The callback object receives the most recent market data information for a product, overlaying updates that are in progress. Below is an example of overlay mode.

Overlay Mode Example

Let's say there are 10 products for a given class.

1) The firm gets an update from CBOE for one of those products.

2) Before the firm finishes processing that one update, the other nine tick.

3) The next call the firm gets from us will be the remaining nine ticks on the remaining nine products all at once.

So, one of the features of overlay is to "group" up Current Market into portions. The portions will be small if the firm processes them fast and potentially larger as the firm's processing becomes slower.

The second feature of overlay is to only give the very latest Current Market.

1) The firm gets an update from CBOE for product X

2) While the firm is processing that update, Product X ticks eight more times.

3) CBOE would then send only the eighth of those updates for Product X to the firm. CBOE would not send the first seven messages in step #2 above.

The only messages that will ever be dropped in overlay mode are "old" current market messages that are no longer "current" for one particular product. Those old current markets will be overlaid on a product by product basis. Also, there will never be queuing in overlay because of this, thus it is impossible to disconnect because of large queues if the firm is using overlay mode.

If the firm wants to limit market data, it can limit the market data to a maximum of some number of calls per second. So once the firm gets a market data message from the MDCAS, it sleeps for X milliseconds, then returns the call. This has the effect of forcing overlay on the MDCAS side if the option ticks more then once per some number of seconds. This allows the firm's application to avoid "old" current market messages while at the same time "grouping" the current market into larger messages. It also has the effect of setting a "worst case scenario" for application throughput which steadies out the maximum CPU the application will try to use. If the firm needs larger chunks, this is the only way they can ensure it will be achievable. The firm may want to implement a strategy similar to this.

Session Management

The CMi V2 interfaces will be made available as extensions to the existing CMi on the existing CAS. A CMi user can gain access to the enhanced interfaces by getting a reference to the UserSessionManagerV2. This reference can be obtained in one of two ways. Both options are exposed on the UserAccessV2 object which will be made available as an alternate IOR link on the HTTP port that the CAS is publishing on.

The first option of accessing the new interfaces is to log into the old CMi and obtain a UserSessionManager reference. Using this reference, the CMi user can call getUserSessionManagerV2, which will return a SessionManagerStructV2. This struct will contain the reference to the original CMi's UserSessionManager as well as the new enhanced UserSessionMangerV2. The other option is to logon using the UserAccessV2 interface. The new logon method takes exactly the same parameters as the old CMi's logon method and returns a newly enhanced SessionManagerStructV2 that can then be used to access both the old and new CMi methods.

Session Management V3 and V4 Functionality

The CMi V3 and V4 interfaces will be made available as extensions to the existing CMi on the existing CAS. A CMi user can gain access to the enhanced interfaces by getting a reference to the UserSessionManagerV3 or UserSessionManagerV4. This reference can be obtained through logon using the UserAccessV3 and UserAccessV4 interfaces respectively. The new logon method takes exactly the same parameters as the old CMi's logon method and returns a newly enhanced SessionManagerStructV3 or SessionManagerStuctV4. The UserAccessV3 or UserAccesV4 objects will be made available as an alternate IOR link on the HTTP port that the CAS is publishing on.

Refer to document API-08, the CMi Programmer's Guide for the Market Data Express (MDX) Data Feed, for complete details on using the UserSessionManagerV4 for MDX.

Login IOR

The IOR is by standard a variable length message that can differ in implementation from ORB to ORB. The new release of the CAS and Simulator is built on a different ORB and returns a larger IOR string. This IOR will normally be split across multiple packets – use the supplied length field in the IOR string.

User Roles

This section provides a description of the user roles used in the CMi. Each UserID is assigned a User Role. The User Role defines the features that can be accessed for a user. User Roles exist for the various market participants, such as a broker-dealer or market maker. In addition, User Roles exist for special functions, such as firms to receive all trade reports, display of market data, etc. Users developing applications with the CMi may choose to create applications that use one or more of the following roles.

User Role	**Description**

Broker Dealer	• Main purpose is to enter orders acting as agent
	• Commonly referred to as the "Broker" role
	• The UserID for a Broker Dealer must be an actual membership acronym approved by the CBOE Membership Department
	• Not allowed to enter quotes
	• Allowed to enter orders for any approved clearing firm per the permissions given by the Membership Department
	• Allowed to enter orders for any order origin type (see cmiConstants::OrderOrigins)
	  o BROKER_DEALER
	  o CUSTOMER
	  o CUSTOMER_BROKER_DEALER
	  o FIRM
	  o MARKET_MAKER
	• Allowed to send RFQs but not allowed to receive Request For Quotations (RFQs)
	• Not allowed to access orders entered by other users clearing their trades through the same clearing firm
	• Has full access to viewing products, market data, strategies
	• Able to create strategy products

Customer Broker Dealer	This role is not currently being used by CBOEdirect
DPM	• Commonly referred to as the Lead-Market-Maker (LMM) role • Main purpose is to act as LMM in CBOEdirect platform • User ID must be an actual membership acronym approved by the CBOE Membership Department • Has access similar to the "Market-Maker" role

User Role	Description
	• Allowed to enter proprietary orders for own account • NOT allowed to enter orders acting as agent • Allowed to enter quotes • Only allowed to enter orders for default executing give up firm per the permissions given by the Membership Department • Allowed to enter orders only for MARKET_MAKER order origin type (see cmiConstants::OrderOrigins) • Allowed to send and receive RFQs • Not allowed to access orders entered by other users belonging to same clearing firm • Has full access to viewing products, market data, strategies • Able to create strategy products
Exchange Broker	Reserved role for exchange level order flow
Product Maintenance	For CBOE administrator use only.
Market Maker	• Main purpose is to act as Market-Maker (MM) in CBOEdirect platform • ID must be an actual membership acronym approved by the CBOE Membership Department • Has access similar to the “DPM” role • Allowed to enter proprietary orders for own account • NOT allowed to enter orders acting as agent • Allowed to enter quotes • Only allowed to enter orders for default executing give up firm per the permissions given by the Membership Department • Allowed to enter orders only for MARKET_MAKER order origin type (see cmiConstants::OrderOrigins) • Allowed to send and receive RFQs • Not allowed to access orders entered by other users belonging to same clearing firm • Has full access to viewing products, market data, strategies • Able to create strategy products
Firm	• Main purpose is to enter orders acting as agent on behalf of clearing firm • May only be used by approved CBOE clearing firms • User ID must be an actual membership acronym approved by the CBOE Membership Department that will be changed to the Firm role • Not allowed to enter quotes • Allowed to receive copies of all messages delivered to all users clearing their trades through the default clearing firm. • Allowed to enter orders for the default clearing firm per the permissions given by the Membership Department • Allowed to enter orders for any order origin type (see cmiConstants::OrderOrigins) o BROKER_DEALER o CUSTOMER o CUSTOMER_BROKER_DEALER o FIRM

User Role	Description
	○ MARKET_MAKER • Allowed to send RFQs but not allowed to receive RFQs • Allowed to access orders entered by other users belonging to same clearing firm • Has full access to viewing products, market data, strategies • Able to create strategy products
Firm Display	• Main purpose is to receive copies of all messages delivered to all users clearing their trades through the default clearing firm • User ID must be created and assigned by the CBOE Help Desk • Has access similar to the "Class Display" role • May only be used by approved CBOE clearing firms • Not allowed to enter quotes • Not allowed to enter orders • Not allowed to send RFQs and not allowed to receive RFQs • Allowed to access orders entered by other users belonging to same clearing firm • Has full access to viewing products, market data, strategies • Not able to create strategy products
Class Display	• Main purpose is to display market data • Renamed to "Class Display" from the "Market Data" role • User ID must be created and assigned by the CBOE Help Desk • Not allowed to enter orders • Not allowed to enter quotes • Not allowed to send RFQs and not allowed to receive RFQs • Has full access to viewing products, market data, strategies • Not able to create strategy products
Expected Opening Price	• Main purpose is to display expected opening price and expected opening size messages • User ID must be created and assigned by the CBOE Help Desk • Not allowed to enter orders • Not allowed to enter quotes • Not allowed to send RFQs and not allowed to receive RFQs • Has full access to viewing products, market data, strategies • Not able to create strategy products
Help Desk	For CBOE Administrator use only
Unknown Role	This is the default role when a user is created. No enabled and active CBOEdirect user should persist with this role.

Underlying Ticker and Recap

With respect to Underlying MarketData, if any of the items in a recap message including the bidPrice, askPrice, bidSize, askSize, lastSalePrice and lastSaleVolume are changed, then a recap will be published. There will be no currentMarket messages published for underlying marketData. If a firm cares about lastSale values, it should also subscribe to

the ticker, which contains the underlying lastSale information, which is published when a trade takes place. If a trade is out of sequence, a trade will be published in the form of a ticker but there will be no recap messages published.

Primary and Secondary Login Modes

Users can login multiple times at the same time using the same UserID. If a user ID wishes to logon with multiple user sessions, they all must be on the same CAS. A user ID cannot logon with multiple user sessions using different CASes. This is regardless of the logon session mode (primary, secondary), GMD or not GMD, the API used (FIX, CMi), etc. Each login session can be in a *Primary* or *Secondary* mode. CBOEdirect monitors "primary" login sessions for application termination. It does not matter if the application termination is normal via logout or abnormal due to an application or network failure. If a "primary" application terminates– ALL login sessions (primary and secondary) for that UserID are terminated using the forced logout capability of CBOEdirect. The major reason for using a Primary login session is to ensure that the quotes belonging to a user of the application are removed from the market if the application becomes unavailable. CBOE strongly encourages the firm to enter quotes from a user that is logged on with logon session mode Primary. Login sessions that are in Secondary mode are not monitored for application termination.

The following rules apply for Primary and Secondary login session modes:

- Multiple PRIMARY logins are allowed
- Multiple SECONDARY logins are allowed
- A SECONDARY login does not require a Primary logon session to be active.
- If a PRIMARY login logs out or unexpectedly terminates, all other logins for that user are forced logged out
- If a SECONDARY login logs out, no other login session for the UserID is affected.
- If a firm logs on with one Secondary logon session type and there are no Primary users and no other Secondary users logged on for that user ID, and if the Secondary user logs out, CBOE does remove the user's quotes from the book.

Exchange Qualified Firm Identification

In order to support multiple market places, such as the OneChicago exchange for trading single stock futures, CBOEdirect now requires an exchange to be specified as part of the identification for a member firm.

A particular user session in CBOEdirect may only access one exchange at a time. For example, a single user may not send quotes to CBOE options and CFE at the same time. The firm must use two separate user IDs: one will send quotes to CBOE and one will send quotes to CFE.

Cancel Functionality

Cancel quotes for a class is a synchronous call that directly removes the quotes for a user for a given class. The return of this call indicates that all quotes for the user have been removed. It is possible that there are in flight execution reports. It is also possible a quote

is involved in a trade at the moment of the cancel request. If so, any remaining quantity for that series will be canceled at the end of the trading process.

CancelAllQuotes is a best effort asynchronous request that instructs all CBOEDirect trade servers to cancel all resting quotes for a user. The return of this call does not indicate that all of the users resting quotes have been removed but that the process has been initiated on all trade servers.

Cancels on Restricted Series

The CBOE will begin a rollout of a change to the internal routing of orders. This software load will begin after July 2008 expiration. The external behavioral changes are:

(1) Orders that open positions for restricted series will no longer be accepted and routed to a booth for handling. They will be rejected immediately. The CMi cancel reason code is "const exceptions::ErrorCode ORDER_REJECTED_ON_RSS = 1909".

(2) When a series becomes restricted, any resting orders that would open a position will be canceled by the system. The cancel reports will be delivered to the originating user on login with a cancel report Activity Reason of " const cmiUtil::ActivityReason CANCEL_ON_RSS = 24.

(3) Currently cancel requests for partial quantity of a complex order cause the order to be canceled in its entirety. This behavior will remain if the order is resting on PAR or a booth. However, if the order is booked, the partial cancel request will be applied and any remaining quantity will remain in the market.

Blocked Cancel Requests

In order to allow users to cancel quotes more efficiently the CBOE has modified the accept quote for class processing to allow for blocked cancel requests. The feature is used much in the same way multiple series quotes for a class are submitted. Instead of sending a product key, price and quantity (for both bid and ask) the user sends in a product key and zero values for **both** the bid and ask quantity. The price for the bid and ask is ignored if the quantities are both zero. It is recommended that the price be set to zero as well but it is not required. This modification will allow the user to send in cancel requests for up to 400 series per method call. While you can mix quotes with cancels (for different products) in the same message, it is not recommended. If you exceed a quote rate limit the entire quote block message is rejected and no cancels will make it to the market. There is also no reason to send a cancel and a quote for the same product.

Cancel/Replace Functionality

Below are cancel replace scenarios that are supported through the CMi.

```
interface OrderCancelTypes
{
// This type of cancellation uses the quantity in the cancel request as the quantity of
//the order to be cancelled
```

```
const cmiOrder::CancelType DESIRED_CANCEL_QUANTITY    = 1;

// This type of cancellation uses the quantity as the desired remaining quantity of the
order
// after cancellation. It is not the quantity to be canceled. This type of cancellation is
// compatible with FIX order handling semantics
const cmiOrder::CancelType DESIRED_REMAINING_QUANTITY = 2;

// This type of cancellation cancels all the remaining quantity of an order. Available to
// support FIX style order cancel requests
const cmiOrder::CancelType CANCEL_ALL_QUANTITY        = 3;
};

struct CancelRequestStruct
{
cmiOrder::OrderIdStruct orderId;
cmiSession::TradingSessionName sessionName;
string userAssignedCancelId;
cmiOrder::CancelType cancelType;
long quantity;
};
```

Assume the original order had a quantity of 10.

DESIRED_CANCEL_QUANTITY

1) Desired Cancel Quantity = Remaining Quantity

Cancel Request (cancelType = DESIRED_CANCEL_QUANTITY, quantity = 10)

Original Order (remaining quantity = 10)

Original Order: Canceled w/ Cancel Report (tlcQuantity = 0, cancelledQuantity = 10, mismatchedQuantity = 0).

Replacement Order: Processed.

Exception: None.

2) Desired Cancel Quantity > Remaining Quantity [Race Condition]

a) Cancel Request (cancelType = DESIRED_CANCEL_QUANTITY, quantity = 10)

Original Order (remaining quantity = 5) [order partially traded]

Original Order: Unchanged (remaining quantity = 5).

Replacement Order: Ignored.

Exception: DataValidationException. [remaining quantity < the requested cancel quantity]

b) Cancel Request (cancelType = DESIRED_CANCEL_QUANTITY, quantity = 10)

Original Order (remaining quantity = 10/5) [order partially traded while the cancel request was being processed in the server]

Original Order: Unchanged (remaining quantity = 5) w/ Cancel Report (tlcQuantity = 0, cancelledQuantity = 0, mismatchedQuantity = 10).

Replacement Order: Canceled w/ Cancel Report (tlcQuantity = 0, cancelledQuantity = 0, mismatchedQuantity = 0, userAssignedCancelId = “”) [System Generated Cancel]

Exception: None.

Note: This case is due to a rare race condition in the server, and should not normally be encountered.

3) Desired Cancel Quantity < Remaining Quantity

Cancel Request (cancelType = DESIRED_CANCEL_QUANTITY, quantity = 7)

Original Order (remaining quantity = 10)

Original Order: Booked (remaining quantity = 3) w/ Cancel Report (tlcQuantity = 0, cancelledQuantity = 7, mismatchedQuantity = 0).

Replacement Order: Processed.

Exception: None.

Note: There are now two orders in the book for the user.

DESIRED_REMAINING_QUANTITY

1) Desired Remaining Quantity < Remaining Quantity (= Original Quantity)

Cancel Request (cancelType = DESIRED_REMAINING_QUANTITY, quantity = 0)

Original Order (remaining quantity = 10)

Original Order: Canceled w/ Cancel Report (tlcQuantity = 0, cancelledQuantity = 10, mismatchedQuantity = 0).

Replacement Order: Processed.

Exception: None.

2) Desired Remaining Quantity < Remaining Quantity (< Original Quantity)

Cancel Request (cancelType = DESIRED_REMAINING_QUANTITY, quantity = 0)

Original Order (remaining quantity = 5) [order partially traded]

Original Order: Canceled w/ Cancel Report (tlcQuantity = 0, cancelledQuantity = 5, mismatchedQuantity = 0).

Replacement Order: Processed.

Exception: None.

3) Desired Remaining Quantity = 0, Remaining Quantity = 0

Cancel Request (cancelType = DESIRED_REMAINING_QUANTITY, quantity = 0)

Original Order (remaining quantity = 0) [order fully traded]

Original Order: Fully Filled (remaining quantity = 0) w/ Cancel Report (tlcQuantity = 10, cancelledQuantity = 0, mismatchedQuantity = 0).

Replacement Order: **Processed**.

Exception: None.

Note: The replacement order is processed even though the original order was fully filled.

4) Desired Remaining Quantity (> 0) < Remaining Quantity (= Original Quantity)

Cancel Request (cancelType = DESIRED_REMAINING_QUANTITY, quantity = 5)

Original Order (remaining quantity = 10)

Original Order: Booked (remaining quantity = 5) w/ Cancel Report (tlcQuantity = 0, cancelledQuantity = 5, mismatchedQuantity = 0).

Replacement Order: Processed.

Exception: None.

Note: There are now two orders in the book for the user.

5) Desired Remaining Quantity > Remaining Quantity

Cancel Request (cancelType = DESIRED_REMAINING_QUANTITY, quantity = 7)

Original Order (remaining quantity = 5) [order partially traded]

Original Order: Unchanged (remaining quantity = 5).

Replacement Order: Ignored.

Exception: DataValidationException. INVALID_QUANTITY. [remaining quantity < the requested cancel quantity]

CANCEL_ALL_QUANTITY

1) Remaining Quantity = Original Quantity

Cancel Request (cancelType = CANCEL_ALL_QUANTITY) [quantity field ignored]

Original Order (remaining quantity = 10)

Original Order: Canceled w/ Cancel Report (tlcQuantity = 0, cancelledQuantity = 10, mismatchedQuantity = 0).

Replacement Order: Processed.

Exception: None.

2) Remaining Quantity < Original Quantity

Cancel Request (cancelType = CANCEL_ALL_QUANTITY) [quantity field ignored]

Original Order (remaining quantity = 5) [order partially traded]

Original Order: Canceled w/ Cancel Report (tlcQuantity = 0, cancelledQuantity = 5, mismatchedQuantity = 0).

Replacement Order: Processed.

Exception: None.

3) Remaining Quantity = 0

Cancel Request (cancelType = CANCEL_ALL_QUANTITY) [quantity field ignored]

Original Order (remaining quantity = 0) [order fully traded]

Original Order: Fully Filled (remaining quantity = 0) w/ Cancel Report (tlcQuantity = 10, cancelledQuantity = 0, mismatchedQuantity = 0).

Replacement Order: **Processed**.

Exception: None.

Note: The replacement order is processed even though the original order was fully filled.

Additional Behavior in Too Late to Cancel Scenarios

CBOE currently tracks all orders. In order to improve order flow, CBOE will no longer store inactive orders (i.e. orders with no remaining working quantity) nor process requests regarding any inactive order. Once an order has been filled or cancelled it will be removed from the active order process after a short duration. Currently, the minimum time an order could be referenced after being filled or cancelled is 30 seconds. No filled or cancelled order will exist for more than 1 minute. This functionality will be available with the late April 2008 release of CBOEdirect.

If no order is found a DataValidationException with the error code INVALID_ORDER_ID is thrown. If an order exists but its remaining quantity is zero a DataValidationException with the error code NO_WORKING_ORDER will be thrown instead.

CMI users should no longer expect to always receive a TLTC messages for cancel requests as the window for their creation has been greatly reduced.

The cancel reports for ExpressOrders are no longer GMD. If a user logs out before receiving one or more ExpressOrder cancel reports, they will not be re-published as poss-resend when the user logs in again.

There is an important consequence to be noted with regard to order entry. If the original order has not been accepted by the system, the cancel request for that order will be rejected with INVALID_ORDER_ID. So, it is important to wait for the order call to return before issuing a cancel request. Since an Express Order will be cancelled or filled, they do not warrant any cancel requests. Any cancel replace request on Express orders will be rejected.

Determining Trade Participants

The trade ID is one way to determine participants in a trade. The trade ID is unique, however, it will be the same for all orders and quotes that participated in the trade. Orders have a unique CBOE identifier (high+low) but they are not unique versus a quote ID. Currently, a given user can only have one quote per series, therefore, using the user ID is an option. For either an order or trade, you will need the transaction sequence number as there can be multiple fills generated for each.

- To search for order fills, you will need the trade ID, OrderID.high, OrderId.low and the TransactionSequenceNumber.
- To search for quote fills, you will need the trade ID, userID and TransactionSequenceNumber.

Creating and Submitting an In-Crowd Market-Maker (ICM, or I) Order

Market-Maker and DPM roles who intend to submit orders to Hybrid on the trading floor may do so by submitting an ICM order with the order origin “I”.

I orders are treated like quotes during trading but they are not entered or persisted in the same manner. Like quotes, “I” orders are canceled when a user logs out or is forced off the system.

The following rules apply for I orders:

- CBOE market makers and DPM market-makers may only send orders to CBOE using two order origin codes, "I" (In-Crowd Market-maker, or ICM) and "M" (CBOE Market-maker). Firms may use a Broker-Dealer ("BD") logon to enter orders on behalf of a CBOE market maker or DPM market-maker. "I" orders are effectively treated as one-sided quotes. Just as a quote that is locked in Hybrid, CBOE sends a quote locked notification message to the market-maker originator when an "I" order results in a locked market. CMi Broker-Dealer users can subscribe for quote locked messages. When a Broker-Dealer user logs on, CBOE will automatically subscribe the user to receive Quote Locked messages. Refer to the following section for Quote Lock information.
- "I" (ICM) orders should only be sent when the member is actually physically present in the trading crowd. If the member leaves the trading crowd, orders must be sent with order origin "M", not "I".
- "I" orders cannot be sent with a contingency or with a price of market (MKT).
- "M" orders can be sent with a contingency or with a price of market (MKT).
- "I" orders route to CBOE's back-end order routing system, and then come back to CBOEdirect, so performance on them will be slightly worse than it is for quotes.

- A market-maker can enter multiple "I" orders at different prices on one particular side of the market of a particular product.
- CBOE does cancel the user's "I" orders when the user logs out.
- "I" orders do not count toward a market-maker's Quote Risk Maintenance (QRM) profiles.
- ICM orders will only receive quote locked messages if the ICM order is locked against another market-maker's quote or ICM order.
- ICM orders must have a time in force of DAY. No Good 'Til Cancel (GTC) or Good 'Til Date (GTD) ICM orders will be accepted.
- ICM Strategy orders are not supported at this time.
- CBOE does not cancel "I" orders when the products close.

Preferred Market Maker

Firms that wish to give one DPM priority in participating in a trade use optional data. The firm would send "P:firm;" and can coexist with other data that may be present in this field. "firm" is the CBOE firm acronym that will be supplied by CBOE. Please note that the message must include the colon : and semi-colon ;

Market Maker Hand Held Functionality.

The August 2009 software release of CBOE*direct* will encompass Market Maker Hand Held (MMHH) functionality in the CMi V6 interfaces. The existing Market Maker Hand Held system gives Firm traders the ability to electronically enter their trades on a firm's handheld device and send them for processing to the MMHH System via the NCC interface. With the new CMi V6 interfaces, CMi users will have the ability to perform MMHH functionality using the CBOE*direct* platform instead of the NCC interface.

Login to CBOEdirect using the UserAccessV6 Interface

In order to perform a MMHH trade on CBOE*direct*, users must login using the UserAccessV6 interface. The UserAccessV6 interface references the UserSessionManagerV6, which references the FloorTradeMaintenanceService. The FloorTradeMaintenanceService is used to add, delete, subscribe and unsubscribe for MMHH trades.

```
interface UserAccessV6
{
  UserSessionManagerV6 logon
  (
        in cmiUser::UserLogonStruct logonStruct,
        in cmiSession::LoginSessionType sessionType,
        in cmiCallback::CMIUserSessionAdmin clientListener,
        in boolean gmdTextMessaging )
           raises(
              exceptions::SystemException,
              exceptions::CommunicationException,
              exceptions::AuthorizationException,
              exceptions::AuthenticationException,
```

```
                exceptions::DataValidationException,
                exceptions::NotFoundException
            );
};
interface UserSessionManagerV6 : cmiV5::UserSessionManagerV5
{
    FloorTradeMaintenanceService getFloorTradeMaintenanceService()
        raises(
            exceptions::SystemException,
            exceptions::CommunicationException,
            exceptions::AuthorizationException
        );
};
```

Generate a Market Maker Hand Held Trade

A MMHH trade is generated by calling the acceptFloorTrade method in the cmiV6:FloorTradeMaintenanceService interface. The acceptFloorTrade method takes the cmiTrade:FloorTradeEntryStruct as an argument and returns the cmiUtil:CboeIdStruct on successful trade generation. Considerations for generating a MMHH trade are:

- Only Market Makers (i.e. MarketMaker (M) and DPM_Role (D)) are allowed to submit a MMHH trade.
- If a MMHH trade is not generated successfully, the API will throw an exception specifying the general cause.
- The Market Maker entering the trade will not receive trade fill reports. Successful return of the method call should be considered as the confirmation.
- The CBOE*direct* trade engine has specific validation rules for allowing MMHH trades based on userId, executingAcronym, sessionName and ProductKey combinations. It maintains an internal mapping table for userId/acronym combinations. This table is configurable by the CBOE Help Desk. Based on passing parameters, the trade engine performs two look-ups in order to obtain the user information to carry on the trade. The first look-up looks for a match between the acronym corresponding to the user profile derived from the passed user id and the passed executing acronym. If this look-up does not return positive results then a second look-up is performed. The second look-up looks for an association between the executing acronym and the passed user id. Such association is entered and maintained as a firm property. If the second look-up does not find a match then the trade cannot proceed and an error message explaining the problem is sent as an exception.

```
interface FloorTradeMaintenanceService
{
        cmiUtil::CboeIdStruct acceptFloorTrade
                (
                in cmiTrade::FloorTradeEntryStruct floorTrade)
            raises(
                exceptions::SystemException,
                exceptions::CommunicationException,
                exceptions::AuthorizationException,
                exceptions::DataValidationException,
```

```
                    exceptions::NotAcceptedException,
                      exceptions::TransactionFailedException
                  );
module cmiTrade
{
struct FloorTradeEntryStruct
        {
        cmiSession::TradingSessionName sessionName;
        cmiProduct::ProductKey productKey;
        long quantity;
        cmiUtil::PriceStruct price;
        cmiUtil::Side side;
        string account;
    string subaccount;
        cmiUser::ExchangeFirmStruct cmta;
        cmiUser::ExchangeAcronymStruct executingMarketMaker;
        cmiUser::ExchangeFirmStruct firm;
        char positionEffect;
        cmiUser::ExchangeAcronymStruct contraBroker;
        cmiUser::ExchangeFirmStruct contraFirm;
        cmiUtil::DateTimeStruct timeTraded;
        string optionalData;
};
```

The following validations apply to the cmiTrade::FloorTradeEntryStruct.

Input Field Name	Mandatory or Optional Input	Valid Format Check	Valid Range Check	Existence Check
Trading Session Name	Mandatory			√
Product Key	Mandatory			√
Quantity	Mandatory	N/A	√	
Price	Mandatory	N/A	√	
Side	Mandatory			√
Executing Broker Acronym	Mandatory	√		√
Contra Broker	Mandatory	√		
Contra Firm	Mandatory	√		
Position Effect	Optional			√
Account	Optional			
Subaccount	Optional			
CMTA (Exchange + Firm)	Optional		**√ (Firm)**	**√ (Exchange)**
Executing Firm	Optional			
Optional Data	Optional			

N/A: Not Applicable
√ : Validated
Blank cell: Field is not validated for that specific check

Delete a Market Maker Hand Held Trade

The cmiV6:FloorTradeMaintenanceService is used to delete MMHH trades. A valid user, tradeId, tradingSession, productKey and ExchangeFirm must be specified to delete the

trade. If a trade is not deleted successfully an exception will be thrown specifying the general cause. TradeBust reports will not be sent for deleted trades.

```
interface FloorTradeMaintenanceService
{
        void deleteFloorTrade
                        (
                in cmiSession::TradingSessionName sessionName,
                in cmiProduct::ProductKey productKey,
                in cmiUtil::CboeIdStruct tradeId,
                in cmiUser::ExchangeAcronymStruct user,
                in cmiUser::ExchangeFirmStruct firm,
                in string reason)
        raises(
            exceptions::SystemException,
            exceptions::CommunicationException,
            exceptions::AuthorizationException,
            exceptions::DataValidationException,
            exceptions::NotAcceptedException,
            exceptions::NotFoundException,
            exceptions::TransactionFailedException
          );
```

Subscribe for a Market Maker Hand Held Trade

Whenever a PAR trade is received by the Order Handling Service (OHS) it generates a CMi based quote fill message (cmiQuote::QuoteFilledReportStruct). This message will be used for both Market Maker Trade Notifications (MMTN) and Floor Trade Reports.

- In order to distinguish between a regular quote fill message and a MMTN, the QuoteId is always set to 0 for MMTN.
- A user can subscribe for MMHH notifications using the cmiV6::FloorTradeMaintenanceService as shown below. If the classKey is zero, the user receives MMHH trade notifications for all classes. If the classKey has a valid non-zero value, the user gets MMHH trade notifications for the specified classKey only. If neither subscription exists, the CAS will drop the subscription.

  Note: It is highly recommended that the user subscribe for MMHH notifications for all classes.
- A user can specify either the same consumer or different consumers for their subscriptions. The CAS does not prevent users from using their regular QuoteStatus consumer as the FloorTradeReport consumer. The transaction sequence number in the report could be any number so assumptions should not be made on it.
- CBOE Help Desk has to turn on a specific property known as the “Firm Market Maker Trade Notification Parameter" in order to activate trade notifications. By default it is turned off. Firms have to contact the CBOE Help Desk for activation.
- CAS subscription settings do not go to the server. So, in order to have a user receive the MMTNs successfully, user should be setup correctly in both the places. Both side setups are independent and different. So, it is the user’s responsibility to send the correct subscriptions to the CAS.

```
interface FloorTradeMaintenanceService

    void subscribeForFloorTradeReportsByClass
    (
                    in cmiCallbackV2::
                    CMIQuoteStatusConsumer floorTradeReportConsumer,
                    in cmiProduct::ClassKey classKey,
                    in boolean gmdCallback)
        raises(
            exceptions::SystemException,
            exceptions::CommunicationException,
            exceptions::AuthorizationException,
            exceptions::DataValidationException
    );
```

Unsubscribe for a Market Maker Hand Held Trade

Users can turn off MMHH trade notifications by using the unsubscribeForFloorTradeReportsByClass method. If the classKey is zero, notification is turned off for all classes. If the classKey has a valid non-zero value, notification is turned off only for the specified classKey.

Note: It is highly recommended that the user unsubscribe for MMHH notifications for all classes.

```
interface FloorTradeMaintenanceService

    void unsubscribeForFloorTradeReportsByClass
    (
    in cmiCallbackV2::CMIQuoteStatusConsumer
            floorTradeReportConsumer,
    in cmiProduct::ClassKey classKey)
    raises(
            exceptions::SystemException,
        exceptions::CommunicationException,
        exceptions::AuthorizationException,
        exceptions::DataValidationException
            );
};
```

CMi V7 Interfaces

The CMi V7 interfaces support two new features; synchronous order entry changes and short sale marking functionality for orders and quotes.

Session Management

The new CMi V7 interfaces will be made available as extensions to the existing CMi on the existing CAS. A CMi user can gain access to the enhanced interfaces by getting a reference to the UserSessionManagerV7. This reference can be obtained through logon using the UserAccessV7 interface. The new logon method takes exactly the same parameters as the old CMi's logon method and returns a newly enhanced

SessionManagerV7. The UserAccessV7 object will be made available as an alternate IOR link on the HTTP port that the CAS is publishing on.

```
interface UserAccessV7
  {
    UserSessionManagerV7 logon(
      in cmiUser::UserLogonStruct logonStruct,
      in cmiSession::LoginSessionType sessionType,
      in cmiCallback::CMIUserSessionAdmin clientListener,
      in boolean gmdTextMessaging )
        raises(
          exceptions::SystemException,
          exceptions::CommunicationException,
          exceptions::AuthorizationException,
          exceptions::AuthenticationException,
          exceptions::DataValidationException,
          exceptions::NotFoundException
        );
  };
```

Synchronous Order Entry

The CMi V7 interfaces support a new paradigm for entering orders that will make order entry a true synchronous call. Current order entry interface methods return order id information in an OrderIdStruct, and they result in asynchronous new order acknowledgement reports being delivered. Orders entered through the CMi V7 order entry methods will return a full OrderStruct (or some variation of an OrderStruct wrapper) containing all the pertinent order information, including the embedded OrderIdStruct, and will not result in a new order acknowledgement being delivered.

New Order Entry methods

New V7 order entry methods have been added in this release. Both the NoAckV7 and the V7 for strategies methods provide the ‘Side’ field on the Strategy Orders to indicate possible short sale positions.

The NoAckV7 methods will differ in their return behavior from the current methods – these do not generate an asynchronous new order acknowledgement report. The new V7 methods retain current behavior, returning an asynchronous new order acknowledgement on the order entry. Behavior will remain the same for the currently existing order entry methods.

In addition, this release provides new methods for submission of new orders for strategies, cancel replaces for strategies and internalization orders for strategies.

New Quote Entry Methods

New V7 methods are provided in this release for submitting both single quotes and mass quotes. These methods use the new QuoteEntryStructV4, which allows users to submit the short sale position information. The V7 method will continue to maintain existing return behavior – the result of a successful mass quote submission will continue to be a ClassQuoteResultStructV3Sequence.

Short Sale Marking

The CMi V7 interface will provide support for indicating short sale positions on all orders, quotes and mass quotes.

- For simple order entry, a short sale can be indicated using the existing ‘Side’ indicator on the OrderEntryStruct, cmiUtil::Side side.

```
module cmiOrder
{
    struct OrderEntryStruct
            {
               cmiUser::ExchangeFirmStruct executingOrGiveUpFirm;
               string branch;
               long branchSequenceNumber;
               string correspondentFirm;
               string orderDate; // YYYYMMDD format
               cmiUser::ExchangeAcronymStruct originator;
               long originalQuantity;
               cmiProduct::ProductKey productKey;
               cmiUtil::Side side;
               cmiUtil::PriceStruct price;
               cmiOrder::TimeInForce timeInForce;
               cmiUtil::DateTimeStruct expireTime;
               cmiOrder::OrderContingencyStruct contingency;
               cmiUser::ExchangeFirmStruct cmta;
               string extensions;
               string account;
               string subaccount;
               cmiOrder::PositionEffect positionEffect;
               cmiOrder::CrossingIndicator cross;
               cmiOrder::OriginType orderOriginType;
               cmiOrder::Coverage coverage;
               cmiOrder::NBBOProtectionType orderNBBOProtectionType;
               string optionalData;
               string userAssignedId;
               cmiSession::TradingSessionNameSequence sessionNames;
            };
```

- For Complex Order Entry, the new LegOrderEntryStructV2 now provides a similar ‘Side’ indicator to indicate the short sale of a particular leg.

```
module cmiOrder
{
    struct LegOrderEntryStructV2
       {
          cmiOrder::LegOrderEntryStruct legOrderEntry;
          cmiUtil::Side side;
           string extensions;
       };
```

- For quote entry, a similar 'Side' field is now also provided on the QuoteEntryStructV4 to enable the indication of the Short Sale on the individual quote, or on each of the quotes of the mass quote.

```
module cmiQuote
        struct QuoteEntryStructV4
          {
            cmiQuote::QuoteEntryStructV3 quoteEntryV3;
            cmiUtil::Side sellShortIndicator;
            string extensions;
          };
        typedef sequence <QuoteEntryStructV4> QuoteEntryStructV4Sequence;
```

CMi V8 Interfaces

CMi V8.0 module provides a new interface to report outage on a trading class during trading hours. The new service is based on a subscription mechanism. Subscription could be made on a trading group (which is a list of classes) or at a class level within a trading group. Once a group/class is marked down. CBOE will not accept any new orders/Cancel Replace/ Quotes on that class. However order/quote cancels will be processed.

Session Management

The new CMi V8 interfaces will be an extension to the existing CMi on the existing CAS. A CMi user can gain access to "TradingClassStatusQuery" by getting a reference to the UserSessionManagerV8. This reference can be obtained through logon using the UserAccessV8 interface. The new logon method takes exactly the same parameters as the old CMi's logon method and returns the new SessionManagerV8. The UserAccessV8 object will be made available as an alternate IOR link on the HTTP port that the CAS is publishing on.

```
interface UserAccessV8
{
        UserSessionManagerV8 logon(
                in cmiUser::UserLogonStruct logonStruct,
                in cmiSession::LoginSessionType sessionType,
                in cmiCallback::CMIUserSessionAdmin clientListener,
                in boolean gmdTextMessaging )
                        raises(
                        exceptions::SystemException,
                        exceptions::CommunicationException,
                        exceptions::AuthorizationException,
                        exceptions::AuthenticationException,
                        exceptions::DataValidationException,
                        exceptions::NotFoundException
        );

        };

interface UserSessionManagerV8 : cmiV7::UserSessionManagerV7
{
        TradingClassStatusQuery getTradingClassStatusQuery()
raises(
```

```
        exceptions::SystemException,
        exceptions::CommunicationException,
        exceptions::AuthorizationException,
        exceptions::AuthenticationException,
        exceptions::NotFoundException
);
```

New Trading Class Status Query interface:

This new interface available through SessionManagerV8 provides the list off all Trading Groups and the list classes traded in each group. It also has two different subscriptions for Trading Class Status. The subscription methods take a callback to notify clients of any outage. There are four methods in TradingClassStatusQuery interface and a new CallbackV5 for clients to implement in this release.

```
interface TradingClassStatusQuery
{
        cmiProduct::ProductGroupSequence getProductGroups()
raises(
                exceptions::SystemException,
                exceptions::CommunicationException,
                exceptions::DataValidationException,
                exceptions::NotFoundException,
                exceptions::AuthorizationException
                );

        cmiProduct::ClassKeySequence getClassesForProductGroup(in
        cmiProduct::ProductGroup productGroupName)
                raises(
                exceptions::SystemException,
                exceptions::CommunicationException,
                exceptions::DataValidationException,
                exceptions::NotFoundException,
                exceptions::AuthorizationException
);

        void subscribeTradingClassStatusForProductGroup(
         in cmiSession::TradingSessionName sessionName,
        in cmiProduct::ProductGroupSequence roductGroupNames,
        in cmiCallbackV5::CMITradingClassStatusQueryConsumer clientListener)
                 raises(
                exceptions::SystemException,
                exceptions::CommunicationException,
                exceptions::AuthorizationException,
                exceptions::DataValidationException
);

        void subscribeTradingClassStatusForClasses(
in cmiSession::TradingSessionName sessionName,
in cmiProduct::ClassKeySequence classKeys,
        in cmiCallbackV5::CMITradingClassStatusQueryConsumer clientListener)
        raises(
        exceptions::SystemException,
```

```
        exceptions::CommunicationException,
        exceptions::AuthorizationException,
        exceptions::DataValidationException
};
```

Getting Groups/ Classes for Subscription:
The new Interface provides a method getProductGroups() which returns a list of all trading groups irrespective of trading session.

The method "getClassesForProductGroup" takes a group name as parameter and returns the list classes traded in that group.

Subscription based on Groups:
Subscriptions could be made on the group level. A group will have at least one or more trading classes. The method "subscribeTradingClassStatusForProductGroup" helps you do this. The method take in the Trading Session(like W_MAIN), the list group Name and a callback to notify end clients.

Subscription based on ClassKeys:
Subscriptions could also be made as granular as at the class level. A classes may or may not have products/series defined under them. The method "subscribeTradingClassStatusForClasses" helps you do this. The method take in the Trading Session(like W_MAIN), the list class keys and a callback to notify end clients.

Note: There are no unsubscribing methods in the new interface. All trading class status subscriptions are cancelled when user logs out.

New CmiCallBackV5 interface:

There are two methods in this new interface. Based on the subscription type (group or class) appropriate methods are called. This release introduces new CmiConstants to notify users of the trading class status.

```
interface CMITradingClassStatusQueryConsumer {
        void acceptTradingClassStatusUpdateforProductGroups(
        in cmiProduct::ProductGroupSequence listOfProductGroups,
        in cmiUtil::TradingClassStatusIndicator status);

void acceptTradingClassStatusUpdateforClasses(
        in cmiProduct::ClassKeySequence listOfClasses,
        in cmiUtil::TradingClassStatusIndicator status);
        };
```

New CMiConstants interface:

These two constant are send out as trading class status. Once a trading class closed due to outage. CBOE will not accept any new Orders, Cancel Replace or Quotes for that group or class. However Order/Quote/System generated Cancels will be processed.

```
interface TradingClassStatusIndicators

{

const short CLOSED_OUTAGE          = 1;
```

```
const short OPEN_AFTER_OUTAGE    = 4;
};
```

CMi V9 Interfaces

The CMi V9 interfaces support a new mechanism for entering orders that makes order entry light and rapid. CMi V9 provides an order entry interface that customers can use in lieu of quotes to take the added advantages of tiered quoting as well as one sided quoting.

Light order entry is accomplished by minimizing the order message size and by lessening the order status reports sent to the CMi user. CMi users, using the CMi V9 interfaces, will not receive NEW and Cancel reports. Instead, order details will be supplied as part of the return struct.

Session Management

The new CMi V9 interfaces will be made available as extensions to the existing CMi on the existing CAS. A CMi user can gain access to the enhanced interfaces by getting a reference to the UserSessionManagerV9. This reference can be obtained through logon using the UserAccessV9 interface. The new logon method takes exactly the same parameters as the old CMi's logon method and returns a newly enhanced SessionManagerV9. The UserAccessV9 object will be made available as an alternate IOR link on the HTTP port that the CAS is publishing on.

```
interface UserAccessV9
  {
        UserSessionManagerV9 logon(
        in cmiUser::UserLogonStruct logonStruct,
        in cmiSession::LoginSessionType sessionType,
        in cmiCallback::CMIUserSessionAdmin clientListener,
        in boolean gmdTextMessaging )
                        raises(
                        exceptions::SystemException,
                        exceptions::CommunicationException,
                        exceptions::AuthorizationException,
                        exceptions::AuthenticationException,
                        exceptions::DataValidationException,
                        exceptions::NotFoundException
                        );
  };

interface UserSessionManagerV9 : cmiV8::UserSessionManagerV8
        {
```

```
                cmiV9::OrderEntry   getOrderEntryV9()
        raises(
                exceptions::SystemException,
                exceptions::CommunicationException,
                exceptions::AuthorizationException,
                exceptions::AuthenticationException,
                exceptions::NotFoundException
                );
    };
```

Light Order Entry

A light order is generated by calling the acceptLightOrder method in the cmiV7:OrderEntry interface. The acceptLightOrder method takes the cmiOrder:LightOrderEntryStruct as an argument and returns the cmiOrder:LightOrderResultStruct on successful entry. A NEW report will not be generated. Light orders support only simple and IOC orders.

```
interface OrderEntry: cmiV7::OrderEntry
        {
                cmiOrder:: LightOrderResultStruct acceptLightOrder(
                   in cmiOrder:: LightOrderEntryStruct anOrder)
                        raises(
                           exceptions::SystemException,
                           exceptions::CommunicationException,
                           exceptions::AuthorizationException,
                           exceptions::DataValidationException,
                           exceptions::NotAcceptedException,
                           exceptions::TransactionFailedException,
                           exceptions::AlreadyExistsException
                        );

module cmiOrder
        {
        struct LightOrderEntryStruct
                        {
                           string branch;
                           long branchSequenceNumber;
```

```
        long originalQuantity;
        double Price;
        cmiProduct::ProductKey productKey;
        cmiUtil::Side side;
        cmiOrder::PositionEffect positionEffect;
        cmiOrder::Coverage coverage;
        boolean isNBBOProtected;
        boolean isIOC;
        cmiOrder::OriginType orderOriginType;
        cmiUser::Exchange cmtaExchange;
        string cmtaFirmNumber;
        string pdpm;
        string userAssignedId;
        cmiSession::TradingSessionName activeSession;
    };

struct LightOrderResultStruct
  {
    string branch;
    long branchSequenceNumber;
    long orderHighId;
    long orderLowId;
    cmiUtil::Side side;
    long leavesQuantity;
    long tradedQuantity;
    long cancelledQuantity;
    cmiUtil::ActivityReason reason;
    cmiUtil::DateTimeStruct time;
  };
   typedef sequence <LightOrderResultStruct>
LightOrderResultStructSequence;
```

Cancel a Light Order

Light orders can be canceled using the methods: acceptLightOrderCancelRequest or acceptLightOrderCancelRequestById. A valid user assigned cancel ID, branch, branch sequence number, product key and active session is required to cancel a Light order. A

successful cancel returns the cmiOrder:LightOrderResultStruct. A Cancel report will not be generated.

Light orders can be canceled only through the CMi V9 interfaces. Attempting to cancel a Light order through the existing CMi cancel interface will result in a cancel reject. In addition, the CMi V9 cancel interface is used to cancel only Light orders. Cancels targeting other orders will be rejected. Cancel replace of Light orders is not supported.

```
cmiOrder:: LightOrderResultStruct acceptLightOrderCancelRequest(
                        in string branch,
                        in long branchSequenceNumber,
                        in cmiProduct::ProductKey productKey,
                        in cmiSession::TradingSessionName activeSession,
                        in string userAssignedCancelId
                              )
                        raises(
                           exceptions::SystemException,
                           exceptions::CommunicationException,
                           exceptions::AuthorizationException,
                           exceptions::DataValidationException,
                           exceptions::NotAcceptedException,
                           exceptions::TransactionFailedException
                        );

cmiOrder:: LightOrderResultStruct acceptLightOrderCancelRequestById(
                        in long orderHighId,
                        in long orderLowId,
                        in cmiProduct::ProductKey productKey,
                        in cmiSession::TradingSessionName activeSession,
                        in string userAssignedCancelId
                              )
                        raises(
                           exceptions::SystemException,
                           exceptions::CommunicationException,
                           exceptions::AuthorizationException,
                           exceptions::DataValidationException,
                           exceptions::NotAcceptedException,
                           exceptions::TransactionFailedException
                        );
      };
```

Light Order Error Messages

CMi users need to be explicitly enabled to send Light orders by the CBOE Help Desk. Light orders are enabled at a user acronym level. Users that attempt to enter Light order but are not enabled or setup correctly will receive the following error messages.

interface DataValidationCodes

const exceptions::ErrorCode INVALID_USER_ID_FOR_LIGHT_ORDERS = 1950;

const exceptions::ErrorCode INVALID_ORIGIN_TYPE_FOR_LIGHT_ORDERS = 7200;

interface AuthorizationCodes

const exceptions::ErrorCode USER_NOT_ENABLED_FOR_LIGHT_ORDERS = 7051;

CMi V10 Interfaces

Cancel/Replace for Light Orders

The CMi V10 interfaces provide the mechanism for canceling and replacing Light orders.

Session Management

The new CMi V10 interfaces will be made available as extensions to the existing CMi on the existing CAS. A CMi user can gain access to the enhanced interfaces by getting a reference to the UserSessionManagerV10. This reference can be obtained through logon using the UserAccessV10 interface. The new logon method takes exactly the same parameters as the old CMi's logon method and returns a newly enhanced SessionManagerV10. The UserAccessV10 object will be made available as an alternate IOR link on the HTTP port that the CAS is publishing on.

For the logonV2 method, the userHeartbeatTimeout value should be set from 3 (seconds) to 20 (seconds). If the heartbeat timeout is set to less than 3, the system will default the timeout to 3 seconds. If the heartbeat timeout is set to greater than 20, the system will default the timeout to 20 seconds.

```
interface UserAccessV10
{
        UserSessionManagerV10 logon(
        in cmiUser::UserLogonStruct logonStruct,
        in cmiSession::LoginSessionType sessionType,
        in cmiCallback::CMIUserSessionAdmin clientListener,
        in boolean gmdTextMessaging )
                        raises(
                        exceptions::SystemException,
                        exceptions::CommunicationException,
                        exceptions::AuthorizationException,
                        exceptions::AuthenticationException,
                        exceptions::DataValidationException,
                        exceptions::NotFoundException
```

```
            );
        UserSessionManagerV10 logonV2(
            in cmiUser::UserLogonStruct logonStruct,
            in cmiSession::LoginSessionType sessionType,
            in cmiCallback::CMIUserSessionAdmin clientListener,
            in boolean gmdTextMessaging,
                in long userHeartbeatTimeout )
                    raises(
                    exceptions::SystemException,
                    exceptions::CommunicationException,
                    exceptions::AuthorizationException,
                    exceptions::AuthenticationException,
                    exceptions::DataValidationException,
                    exceptions::NotFoundException
                    );

interface UserSessionManagerV10 : cmiV9::UserSessionManagerV9
    {
            cmiV10::OrderEntry   getOrderEntryV10()
    raises(
            exceptions::SystemException,
            exceptions::CommunicationException,
            exceptions::AuthorizationException,
            exceptions::AuthenticationException,
            exceptions::NotFoundException
            );
```

Cancel/Replace Light Orders

A light order cancel/replace is created by calling the acceptLightOrderCancelReplaceRequest method in the cmiV9:OrderEntry interface. The acceptLightOrderCancelReplaceRequest method takes the cmiOrder: LightOrderEntryStruct as an argument and returns the cmiOrder:: LightOrderReplaceResultStruct on successful entry. If there is a quantity mismatch, CMi users will receive Activity Reason message, const cmiUtil::ActivityReason MISMATCHED_QUANTITY = 907. The user should be aware of the following scenarios:

- If a user tries to cancel a quantity that is greater than the remaining quantity (suggesting an in-flight fill), CBOEdirect will cancel the remaining quantity and cancel the replace order.
- If a user tries to cancel a quantity that is less than the remaining on the order and replace it, the cancel request quantity will be canceled and the replace order will also be canceled.
- Light Order Cancel Replace of IOC contingency orders is not supported. The Cancel/Replace request with this contingency type will be rejected.

```
interface OrderEntry: cmiV9::OrderEntry
{
        cmiOrder:: LightOrderReplaceResultStruct
acceptLightOrderCancelReplaceRequest(
```

```
            in long originalOrderHighId,
            in long originalOrderLowId,
            in cmiSession::TradingSessionName activeSession,
            in long quantityToCancel,
            in string userAssignedCancelReplaceId,
                        in cmiOrder::LightOrderEntryStruct replaceOrder
                        )
                        raises(
                           exceptions::SystemException,
                           exceptions::CommunicationException,
                           exceptions::AuthorizationException,
                           exceptions::DataValidationException,
                           exceptions::NotAcceptedException,
                           exceptions::TransactionFailedException
                        );
   };
module cmiOrder
            {
            struct LightOrderEntryStruct
                        {
                           string branch;
                           long branchSequenceNumber;
                           long originalQuantity;
                           double Price;
                           cmiProduct::ProductKey productKey;
                           cmiUtil::Side side;
                           cmiOrder::PositionEffect positionEffect;
                           cmiOrder::Coverage coverage;
                           boolean isNBBOProtected;
                           boolean isIOC;
                           cmiOrder::OriginType orderOriginType;
                           cmiUser::Exchange cmtaExchange;
                           string cmtaFirmNumber;
                           string pdpm;
                           string userAssignedId;
                           cmiSession::TradingSessionName activeSession;
                        };

struct LightOrderReplaceResultStruct
   {
```

```
            cmiOrder::LightOrderResultStruct originalOrder;
            cmiOrder::LightOrderResultStruct newOrder;
        };
        typedef sequence <LightOrderReplaceResultStruct>
    LightOrderReplaceResultStructSequence;

    };
```

Quote Fill Messages

In a later release of CBOEdirect, CMi users, accessing the CMi V10 interfaces, will be able to subscribe to receive only quote fill messages. Buy subscribing to receive only quote fill messages, the queuing of quote fill messages behind other quote status messages is avoided. New methods are available for the subscription of quote fill messages using the cmiV10.Quote which is part of the V10 Session Manager.

Session Management

A CMi user can gain access to the enhanced interfaces by getting a reference to the UserSessionManagerV10. This reference can be obtained through logon using the UserAccessV10 interface. The new logon method takes exactly the same parameters as the old CMi's logon method and returns a newly enhanced SessionManagerV10. The UserAccessV10 object will be made available as an alternate IOR link on the HTTP port that the CAS is publishing on.

```
interface UserAccessV10
  {
        UserSessionManagerV10 logon(
        in cmiUser::UserLogonStruct logonStruct,
        in cmiSession::LoginSessionType sessionType,
        in cmiCallback::CMIUserSessionAdmin clientListener,
        in boolean gmdTextMessaging )
                raises(
                exceptions::SystemException,
                exceptions::CommunicationException,
                exceptions::AuthorizationException,
                exceptions::AuthenticationException,
                exceptions::DataValidationException,
                exceptions::NotFoundException
                );

        UserSessionManagerV10 logonV2(
        in cmiUser::UserLogonStruct logonStruct,
        in cmiSession::LoginSessionType sessionType,
        in cmiCallback::CMIUserSessionAdmin clientListener,
        in boolean gmdTextMessaging,
                in long userHeartbeatTimeout )
                raises(
                exceptions::SystemException,
                exceptions::CommunicationException,
```

```
                                exceptions::AuthorizationException,
                                exceptions::AuthenticationException,
                                exceptions::DataValidationException,
                                exceptions::NotFoundException
                                );
};


    interface UserSessionManagerV10 : cmiV9::UserSessionManagerV9
            {
                    cmiV10::Quote getQuoteV10()
            raises(
               exceptions::SystemException,
               exceptions::CommunicationException,
               exceptions::AuthorizationException
            );
       };
```

Subscribing/Unsubscribing for Quote Messages

The following methods support subscribing and unsubscribing for quote messages. Firms should be aware that even if the CMiQuoteStatusConsumer object (which supports all quote status methods) is used, only the appropriate method will be called depending on the type of subscription. For example, if a user subscribes for the subscribeQuoteFillStatus method then the only messages the user should expect to receive would be quote fill messages. The remaining methods will never be called from this subscription.

```
    interface Quote: cmiV7::Quote
                {
                   void subscribeQuoteFillStatus(
                      in cmiCallbackV2::CMIQuoteStatusConsumer clientListener,
                      in boolean publishOnSubscribe,
                      in boolean includeUserInitiatedStatus,
                      in boolean gmdCallback)
                         raises(
                            exceptions::SystemException,
                            exceptions::CommunicationException,
                            exceptions::DataValidationException,
                            exceptions::AuthorizationException
                         );

                   void unsubscribeQuoteFillStatus(
                      in cmiCallbackV2::CMIQuoteStatusConsumer clientListener)
                         raises(
                            exceptions::SystemException,
                            exceptions::CommunicationException,
                            exceptions::AuthorizationException,
                            exceptions::DataValidationException
                         );
```

```
void subscribeQuoteFillStatusForClass (
   in cmiProduct::ClassKey classKey,
   in boolean publishOnSubscribe,
   in boolean includeUserInitiatedStatus,
   in cmiCallbackV2::CMIQuoteStatusConsumer clientListener,
   in boolean gmdCallback)
      raises(
         exceptions::SystemException,
         exceptions::CommunicationException,
         exceptions::DataValidationException,
         exceptions::AuthorizationException
      );

void unsubscribeQuoteFillStatusForClass(
   in cmiProduct::ClassKey classKey,
   in cmiCallbackV2::CMIQuoteStatusConsumer clientListener)
      raises(
         exceptions::SystemException,
         exceptions::CommunicationException,
         exceptions::AuthorizationException,
         exceptions::DataValidationException
      );

 void subscribeQuoteStatusWithoutFill(
   in cmiCallbackV2::CMIQuoteStatusConsumer clientListener,
   in boolean publishOnSubscribe,
   in boolean includeUserInitiatedStatus,
   in boolean gmdCallback)
      raises(
         exceptions::SystemException,
         exceptions::CommunicationException,
         exceptions::DataValidationException,
         exceptions::AuthorizationException
      );

void unsubscribeQuoteStatusWithoutFill(
   in cmiCallbackV2::CMIQuoteStatusConsumer clientListener)
      raises(
         exceptions::SystemException,
         exceptions::CommunicationException,
         exceptions::AuthorizationException,
         exceptions::DataValidationException
      );

void subscribeQuoteStatusWithoutFillForClass (
   in cmiProduct::ClassKey classKey,
   in boolean publishOnSubscribe,
   in boolean includeUserInitiatedStatus,
   in cmiCallbackV2::CMIQuoteStatusConsumer clientListener,
   in boolean gmdCallback)
```

```
        raises(
            exceptions::SystemException,
            exceptions::CommunicationException,
            exceptions::DataValidationException,
            exceptions::AuthorizationException
        );

    void unsubscribeQuoteStatusWithoutFillForClass(
        in cmiProduct::ClassKey classKey,
        in cmiCallbackV2::CMIQuoteStatusConsumer clientListener)
            raises(
                exceptions::SystemException,
                exceptions::CommunicationException,
                exceptions::AuthorizationException,
                exceptions::DataValidationException
            );
        };
```

Automated Improvement Mechanism (AIM)

Internalization and Automated Auction

A Firm that wants to trade orders against its own book must first expose the orders to the market. These orders participate in an automated auction to provide some assurance that they are executed at the best price.

Internalization Order Entry

Internalization order entry will be accomplished via the acceptInternalizationOrder method on interface com.cboe.idl.cmiV3.OrderEntry. The acceptInternalizationOrder method call must contain a primary (customer) order and a match (firm) order. The user submitting the two orders wishes to trade the match order with the primary order.

The acceptInternalizationOrder method call must contain a MatchType. Currently, LIMIT_PRICE and AUTO_MATCH are the only two MatchTypes supported. If the MatchType is LIMIT_PRICE, the match order price must be a limit price. If the MatchType is AUTO_MATCH, the match order price must be a market price. In both cases, the primary order price may be a limit price or a market price.

Both the primary and match orders are validated as regular orders. Additional validations pertaining to the validity of the fields in the acceptInternalizationOrder call are performed. If all validations succeed, the primary order will be auctioned and may trade, in full, or in part, with the match order, depending on business rules and auction outcome. The acceptInternalizationOrder method returns an InternalizationOrderResultStruct which contains two OrderResultStructs, one for the primary order and one for the match order. Each OrderResultStruct will contain a valid OrderIdStruct for the order with which it is associated. Additionally, OrderResultStructs for valid orders will contain errorCode values of 0 and empty strings for their errorMessage fields.

If a validation failure occurs while validating the primary order as a regular order a DataValidationException is thrown with an appropriate error code and message. No attempt is made to validate the match order if this occurs. In addition to standard order

validation rules, if the trading session for the primary order does not support internalization, or the trading session and class are not configured for auction, a DataValidationException will be thrown with DataValidationCode.INTERNALIZATION_NOT_ALLOWED.

If all of the above validations are successful, the method will not throw a DataValidationException. After primary order validation, the match type is validated, the match order is validated as an order, and additional validations are performed, including field compatibility between the two orders with respect to trading session, product key, quantity, and price.

If the match order or the additional internalization validations fail, the primary order will be considered valid as if it had been entered via an acceptOrder call. The acceptInternalizationOrder will return an InternalizationOrderResultStruct containing OrderResultStructs for the primary and match order. The fields of the OrderResultStruct for the primary order will be populated as described above; since the primary order will be exposed to the market, the OrderResultStruct for the primary order is used to communicate the OrderIdStruct to the user. The OrderResultStruct corresponding to the the match order is used to communicate any validation failure beyond one involving the primary order. The fields of the OrderResultStruct correspond to the fields of the exception that would have been otherwise thrown.

Internalization Complex Order Entry

Users can participate in a complex order auction using the new method, *acceptInternalizationStrategyOrder* in the CMi V5.0 module. This method has been added to the *OrderEntry* interface. The new method takes two additional parameters, *primaryOrderLegEntries* and *matchOrderLegEntries*. Similar to the existing method *acceptInternalizationOrder* used for entering AIM auctions for regular orders, the new parameters specify the leg order entry structs for the primary and matching order. This feature will be available in April 2008.

module CmiV5

```
interface OrderEntry : cmiV3::OrderEntry
{
   cmiOrder::InternalizationOrderResultStruct
      acceptInternalizationStrategyOrder(
         in cmiOrder::OrderEntryStruct primaryOrder,
         in cmiOrder::LegOrderEntryStructSequence
primaryOrderLegEntries,
         in cmiOrder::OrderEntryStruct matchOrder,
         in cmiOrder::LegOrderEntryStructSequence
matchOrderLegEntries,
         in cmiOrder::MatchType matchType )
      raises(
            exceptions::SystemException,
         exceptions::CommunicationException,
         exceptions::AuthorizationException,
         exceptions::DataValidationException,
         exceptions::NotAcceptedException,
         exceptions::TransactionFailedException
      );
```

Cancel Cancel/Replace Functionality

Any attempt to cancel or cancel replace a primary order or a match order while the auction is in progress will result in the generation of a Too Late to Cancel report at the end of the auction.

AIM Solicitation Mechanism (AIM AON)

AIM AON allows agents to electronically execute orders they represent against solicited orders.

The mechanics for entering orders into AIM AON is the same as for the existing AIM process with two main differences; 1) Both orders entered must have the contingency AON. 2) The orders must have a contract size of at least 500.

In AIM AON, the agency order will trade with the solicited order at the proposed price unless there are auction responses that improve the price of the auction for the total size of the agency order.

This process is available for simple, complex and cross product orders.

Considerations for Internalization

- Once the paired orders are submitted, they cannot be changed or cancelled.
- Internalizing Firms will be guaranteed a percentage of the trade if they are at the final price of the auction. The percentage will be a configurable parameter.
- Both the primary (customer) order and a match (firm) order will be cancelled if, for any reason, the AIM auction cannot be initiated (e.g. fewer than 3 quoters in the series; etc.).
- If a Firm does not wish to cancel the primary order when the auction expires, the Firm must enter A:AIR, instead of A:AIM, in the Optional Data field. This will designate the primary order to be returned to system and trade or book as a regular order.

Auction Events

Users may subscribe for Request For Price (RFP) auction events by trading session and class by calling the subscribeAuction and unsubscribeAuction methods on interface com.cboe.idl.cmiV3.OrderQuery. Any combination of supported auction types may be specified in the subscription. If the wrong session or class is submitted, a DataValidationException will be thrown and the subscription request will not be processed. Likewise, if any other exception is thrown during subscription, the subscription will have failed for all auction types. If no failures occur, but any auction type is invalid, the method will return. The result of the subscription for each auction type will be indicated in the corresponding OperationResultStruct for that auction type in the AuctionSubscriptionResultStructSequence. Each auction type the user submitted in the subscription call will be represented in this sequence, and any validation failure will be indicated by a nonzero error code and an error message in the OperationResultStruct. Subscriptions for any auction type that are successful will be indicated by an error code of 0. The user's callback will be subscribed for all auction types that were not marked invalid in the AuctionSubscriptionResultStructSequence.

Based on CBOE business rules, the announcement of the auction may be limited to certain users. Therefore, when an auction begins, the system will notify certain users of the auction, provided they are subscribed for the new events. Note that subscription for

auction events for a session and class does not guarantee delivery of events for products in that class; business rules may dictate that auction notification be limited to certain users. Auction events will contain product information as well as auction ID, auction type, auction state, and information pertaining to the order being auctioned, such as side, quantity, price, and contingency type. The extentions string in the AuctionStruct may contain multiple values concerning the auction (i.e. NBBO bid, NBBO ask, CMTA firm, executing firm or correspondent firm). AUCTION_INTERNALIZATION and AUCTION_STRATEGY (strategy auctions) are currently supported. Currently, STARTED will be the only auction state supported.

Auction Response

Users who wish to participate in an auction may respond to Auction events by calling acceptOrder on the interface com.cboe.idl.cmiV3.OrderEntry. The OrderEntryStruct must be populated as if it were a normal order with the following modifications.

- The OrderEntryStruct must contain an OrderContingencyStruct with type=ContingencyTypes.AUCTION_RESPONSE. The side of the auction response order must be tradable against the side indicated in the auction event.
- The OrderEntryStruct extensions field must contain a new field for auction response orders corresponding to the auction ID of the auction. The extensions field should contain the substring "auctionId=123:456" where 123 is the high CBOE Id and 456 is the low CBOE Id of the auction ID specified in the auction event. The standard field separator ("\u0001") must be used in between subfields of the extensions field.

The auction response order will be subject to standard order validation rules with the addition of the following. An auction response with an invalid auction ID will result in a DataValidationException with a DataValidationCode of INVALID_AUCTION_ID. An auction response for an auction that has completed will result in a NotAcceptedException with a NotAcceptedCode of AUCTION_ENDED.

Compatibility with the order being auctioned will be verified and may result in a DataValidationException.

Cancel Cancel/Replace Functionality

Auction responses may be cancelled or cancel/replaced during the auction period using the existing cancel and cancel/replace functionality.

Considerations for Auctions

- To start the auction, CBOE will disseminate a new RFP message to those quoters that are quoting any series in the underlying stock at the time the RFP is sent and the firm initiating the auction, if the order was for internalization.
- The auction will start immediately upon receipt of the order and will be live for a short period of time. The time is configurable.
- A response to the auction will not replace the user’s quote since the responses are one-sided. The responses will be treated like IOC orders and will expire after the auction is over. Users will be able to respond at multiple prices to the auction solicitation.
- Auction responses may be cancelled during the auction period.

- Auction responses may be cancel/replaced during the auction.
- Quote locks and quote triggers will end when the auction starts.

Hybrid Automated Liaison Auction Type

The Hybrid Automated Liaison (HAL) auction type (AUCTION_HAL) will allow users to participate in auctions for orders that are:

NBBO Reject: An incoming order is marketable but CBOE is not at the NBBO.

Tweener Lock: An incoming order that is between the market at CBOE but is marketable against an away market.

Tweener: An incoming order that is between the market at CBOE and does not lock or cross an away market.

Effective January 23, 2008, HAL will be activated in the Hybrid 3.0 (non-Hybrid index) Classes (SPX, OEX and MVR). As in all other Hybrid classes, the HAL flash period will be 700ms, and the allocation algorithm will be pro-rata. However, when the Exchange's BBO is represented by a manual quote on the same side as the incoming order, the limit order will automatically route to the electronic book instead of being processed by HAL and the manual quote will be canceled.

Special Considerations for the HAL Auction Type

- A Request for Price (RFP) flash will be used to place the order in an auction state. The RFP flash timer is configurable.
- If a decision to flash the order is taken then CBOEdirect will end any quote locks, quote triggers and any current non-flash auctions will be allocated prior to the start of order flashing.
- CBOEdirect will send RFP at the NBBO price (or order price for a Tweener) to all the market makers quoting in this product or class depending upon the settings.
- RFP will be of type HAL and have the NBBO price as the starting price and the size of the order.
- Response to the flash auction cannot be cancelled.
- Response to the flash auction will result in a Quote Trigger for a pre-configured time.

Simple Auction Liaison (SAL)

The auction type, AUCTION_SAL, is a mechanism that provides a price-improvement auction for simple (non-complex) orders.

SAL Details for Non-Hybrid Index Classes

SAL will be activated in the Hybrid 3.0 (non-Hybrid index) Classes (SPX, OEX and MVR). The SAL mechanism is an auction that allows for electronic price improvement on eligible simple (non-complex) marketable orders.

SAL details for Non-Hybrid Index Classes

1. Eligible marketable orders will be stopped at the LMM quote and exposed to a brief (300ms) electronic aution for price improvement. Customer orders of 250 contracts and less will be eligible.

2. The starting price for the auction will be the LMM quote. The auction increment in all three classes will be $.05 below $3.00 and $.10 above $3.00.
3. The auction message will be available to all market makers with appointments in the class and firms that have orders resting at the BBO.
4. At the end of the auction period, the order will be executed at the best price(s), including any customer book orders, auction responses and the LMM auto-quote.
5. The trade will be allocated using pro-rata, with each user's response capped at the size of the incoming order (i.e. "capped" pro-rata).
6. An auction will not begin if the incoming order size exceeds the LMM quote size or if a manual quote is present on the BBO on the opposite side of the incoming order.

SAL Details for Hybrid Classes

1. When CBOE is at the NBBO, an eligible marketable order will be stopped at the NBBO and exposed to a brief electronic auction for price improvement. The origins, order size and classes eligible for auction will be configurable.
2. Once the auction begins, quoters who were initially on the NBBO and those that subsequently join or improve the market will not be able to fade or reduce size until the process completes (similar to Quote Trigger).
3. The starting price for the auction will be one penny better than the NBBO.
4. The auction message will be available to all market makers quoting the class (and firms that have orders resting at the top of the market).
5. At the end of the auction period, the order will be executed at the best prices. At each execution price, those that were on the NBBO at the start of the auction will have priority, up to their original size, over those who were not at the NBBO. NBBO participants may also join on quantity in excess of their original size along with all other respondents using the CUMA allocation.
6. DPMs, eDPMs and/or Preferred DPMs will retain a participation right if they were initially on the NBBO and are on the final auction price.
7. An auction will not begin (i.e. the incoming order will simply auto-ex) if the displayed size includes quotes and resting book orders, and if the order size exceeds the quote size.
8. The auction will end early and trade against the existing auction responses and quotes for the following reasons:
   a. An incoming quote locks or crosses the displayed market.
   b. An opposite side order is received that is tradable against the SAL order with a quantity equal to or greater than that of the SAL order. If the new order is smaller than the SAL order, it will trade against the SAL order and the auction will continue for the remainder of the SAL order.
   c. A new marketable order is received on the same side as the SAL order. This order will trade against any remaining responses to the original auction (after the original SAL order is filled), and then a new auction will start for the new order, if appropriate.

Directed AIM

CBOE*direct's* Automated Improvement Mechanism (AIM) will be enhanced to include a new auction feature, Directed AIM. The Directed AIM auction will be supported in the

Hybrid (W_MAIN) trading session via the CMi V6 interfaces and will be available in the July 2009 software release of CBOE*direct*.

Directed AIM gives order providers the ability to direct their orders to look for price improvement. The orders can be directed to either:

- a target Firm (only one per order), or
- a target DPM of a class, or
- a target PDPM

The Directed AIM request targeted at a firm will be sent to MMs that are affiliated to the target firm. The Directed AIM request targeted at DPM will be sent to the DPM of the class. The Directed AIM request targeted at PDPM will be sent to the PDPM based on the PDPM assignment made in the Firm/Class routing property. If MM's should choose to price improve, they will send a matching order to start the AIM auction. The MMs also have the ability to allow the Directed AIM request to time out, or to reject the Directed AIM request.

Register for Directed AIM Auction

CMi users will have the option to choose to participate in the Directed AIM auction process via the cmiV6 OrderQuery interface. An example of registering to participate using the OrderQuery interface would be
`registerForDirectedAIM("W_MAIN",69206067).`

module cmiV6

{

interface OrderQuery : cmiV3::OrderQuery

{

void registerForDirectedAIM(in string sessionName, in

cmiProduct::ClassKey classKey)

raises(exceptions::SystemException,

exceptions::CommunicationException,

exceptions::DataValidationException,

exceptions::TransactionFailedException,

exceptions::NotAcceptedException,

exceptions::AuthorizationException);

};

}

The DirectedAIM indicator will be set at a Firm/Class level. The Firm should be able to configure this on a daily basis. The available values are True/False. The default value for this will be False.

The Firms will set this value only once in a day

Helpdesk user will have the ability to override the value if required

The selection will be removed for each firm as part of the end of day process, and reset to its default value of false. Below are examples of registration scenarios.

Registration Table

User Action	Current Registration Status	Updated Registration For the Day	Result	Register for Direct AIM Status
Register for Directed AIM	No Firm Affiliation	N/A	Data validation code: USER_NOT_AFFILIATED_TO_ANY_FIRM =7005	No.
Register for Directed AIM	Registered	No	Data validation code: ALREADY_UPDATES_AS_REGISTERED = 7007	Yes
Register for Directed AIM	Registered	Yes	Data validation code: ALREADY_UPDATES_AS_REGISTERED = 7007	Yes
Register for Directed AIM	Unregistered	No	Register	Yes

Order Submission

As with all auctions, both a primary order and a match order must be submitted for auction participation.

Primary Order

The extensions field in the orderEntryStruct will be enhanced to allow users to send a Directed AIM request to a target firm/PMM or DPM. Valid values for the extensions field from are:

dfirm = Affiliated Firm Acronym if they want to target a Firm.

dfirm = DDPM if they want to target the DPM of the class.

dfirm = DPMM if they want to target the PDPM for the order provider.

The optionalData field will contain the value, A:AIR;

Example:

dfirm=PAX -> where PAX is the Affiliated Firm Acronym

dfirm=DDPM -> where DDPM indicates DPM choice

dfirm=DPMM -> where DPMM indicates PDPM choice

Match Order

The Directed AIM Response Order from the Firm (Match Order) will include the primary order information in the optionalData field as **DAIM:highcobeid:lowcboeid**; Where

highcobeid:lowcboeid is provided in the Directed AIM Notification message published out to the users.

```
module cmiOrder
struct OrderEntryStruct
   {
      cmiUser::ExchangeFirmStruct executingOrGiveUpFirm;
      string branch;
      long branchSequenceNumber;
      string correspondentFirm;
      string orderDate; // YYYYMMDD format
      cmiUser::ExchangeAcronymStruct originator;
      long originalQuantity;
      cmiProduct::ProductKey productKey;
      cmiUtil::Side side;
      cmiUtil::PriceStruct price;
      cmiOrder::TimeInForce timeInForce;
      cmiUtil::DateTimeStruct expireTime;
      cmiOrder::OrderContingencyStruct contingency;
      cmiUser::ExchangeFirmStruct cmta;
      string extensions;
      string account;
      string subaccount;
      cmiOrder::PositionEffect positionEffect;
      cmiOrder::CrossingIndicator cross;
      cmiOrder::OriginType orderOriginType;
      cmiOrder::Coverage coverage;
      cmiOrder::NBBOProtectionType orderNBBOProtectionType;
      string optionalData;
      string userAssignedId;
      cmiSession::TradingSessionNameSequence sessionNames;
   };
   typedef sequence <OrderEntryStruct> OrderEntryStructSequence;

module cmiConstants
```

```
interface ExtensionFields{

// This extension is added for Directed AIM
        const ExtensionField DIRECTED_FIRM = "dfirm";
        const ExtensionField DPM = "DDPM";
        const ExtensionField PDPM = "DPMM";
        }
```

Error Codes and Constants

Below are the new CMi constants and error codes that correspond to Directed AIM.

```
module cmiConstants
{
interface ActivityTypes{
// DirectedAIM Notification Start and End
const cmiTraderActivity::ActivityType DIRECTED_AIM_NOTIFICATION_START =
817;
const cmiTraderActivity::ActivityType DIRECTED_AIM_NOTIFICATION_END =
818;
        }



interface AuctionTypes{
// Auction type codes
const cmiOrder::AuctionType AUCTION_DAIM = 8;
        }
}

module cmiErrorCodes
{
   interface DataValidationCodes  {

// DirectedAIM NotAccepted Codes.
const exceptions::ErrorCode DIRECTED_AIM_PRIMARY_EXPIRED = 7000;
const exceptions::ErrorCode NOT_REGISTERED_FOR_DIRECTED_AIM = 7001;
```

```
const exceptions::ErrorCode NO_PDPM_AVAILABLE_FOR_DIRECTED_AIM =
7002;

const exceptions::ErrorCode NO_DPM_AVAILABLE_FOR_DIRECTED_AIM = 7003;

const exceptions::ErrorCode
NO_AFFILIATED_MM_AVAILABLE_FOR_DIRECTED_AIM = 7004;

const exceptions::ErrorCode USER_NOT_AFFILIATED_TO_ANY_FIRM = 7005;

const exceptions::ErrorCode INVALID_AFFILIATED_FIRM = 7006;

const exceptions::ErrorCode ALREADY_UPDATED_AS_REGISTERED = 7007;

const exceptions::ErrorCode ALREADY_UPDATED_AS_UNREGISTERED = 7008;

        }

}
```

Examples of Data Validation Errors

The table below gives examples of data validation scenarios.

No.	Action	Expect Data Validation Exception Codes
1.	When Directed AIM Notification has expired or the 1st match Order is received.	DIRECTED_AIM_PRIMARY_EXPIRED
2.	When Directed AIM order is sent to the targeted firm, but the target firm is not registered for Directed AIM.	NOT_REGISTERED_FOR_DIRECTED_AIM
3.	When the target is DPMM => PDPM choice, but there are no PMMs setup in routing property.	NO_PDPM_AVAILABLE_FOR_DIRECTED_AIM
4.	When the target is DDPM =>choice, but there is no DPM setup for the class.	NO_DPM_AVAILABLE_FOR_DIRECTED_AIM
5.	When target firm exists but there are no affiliated MMs for the Firm, the Directed AIM order will get rejected.	NO_AFFILIATED_MM_AVAILABLE_FOR_DIRECTED_AIM
6.	If a DPM User is not affiliated to any firm and Directed AIM is targeted for the DPM user.	USER_NOT_AFFILIATED_TO_ANY_FIRM
7.	When user tries to register more than once. But user subscription for auction type Directed AIM does not fail.	ALREADY_UPDATED_AS_REGISTERED
8.	Whenever user tries to unregister more than once. User will not unsubscribe for auction types Directed AIM for that specific class.	ALREADY_UPDATED_AS_UNREGISTERED

Cancel/Cancel Replace Request for Directed AIM Order

Users will be allowed to cancel and cancel replace their Directed AIM request. However, if the targeted Firm had responded to the Directed AIM request and the Directed AIM auction had started, then the cancel or cancel replace request will wait until the end of the auction. For cancel replace of a Directed AIM order, the orderEntry struct will include dfirm in the extensions field and primary order information in the optional data field.

Quote Trigger

What causes a quote trigger?

Quote trigger events are initiated by a quote crossing with a resting non-market maker order. The quote trigger event will disseminate a last sale event immediately. Trigger settings are configurable and should be verified per class with the CBOEdirect Help Desk or the Trading Operations department. During the trigger, other market makers may join in on the trade by sending in quotes that cross the resting customer order. Upon completion of the quote trigger, fill reports will be disseminated to all the parties involved. During this time period, the trade is considered already done. We are simply holding onto the fill reports since the traded quantity has yet to be determined.

What does the CMi user currently see?

Currently, the CBOEdirect user does not receive any special notification that a quote trigger is happening.

What happens if the CMi user tries to cancel their quote?

If the CBOEdirect user attempts to cancel their quote during a quote trigger, the system will allow them to cancel their quote and will notify the user that their quote is now removed from the market. The quote delete report in this case will contain the cancel Reason of "USER" indicating that the quote was removed due to a user initiated action. The act of canceling a quote will not, however, end a quote trigger. Upon the completion of a quote trigger, a fill report will still be generated. So it is possible to receive notification from CBOE that the quote has been removed from the market and then n(5) seconds later receive a fill report.

What happens if the CMi user tries to update their quote?

If the CBOEdirect user attempts to update their quote during a quote trigger, the system will reject the update with a NotAcceptedException. The NotAcceptedException will contain the error code 4060, QUOTE_BEING_PROCESSED, indicating that the quote cannot be changed because it is currently involved in a quote trigger. If a single quote in a mass quote is rejected, that quote will have a 4060 value, QUOTE_BEING_PROCESSED, in the return sequence, indicating the system did not accept that quote. If all quotes in a mass quote fail due to quote triggers, then the server will throw a NotAcceptedException with the error code 4060, QUOTE_BEING_PROCESSED. To protect the CBOEdirect user, if an update is attempted during a quote trigger, the quote involved in a trigger will be marked for deletion. Upon the completion of the quote trigger, a fill report will be disseminated, any remaining quantity in the quote will be immediately canceled and a quote delete report will be disseminated. The quote delete report in this case will contain a cancel Reason of "SYSTEM" to indicate the quote was removed by a system-initiated action.

What Ends a Quote Trigger Early?

There are two events that will cause an early termination of a quote trigger. If there is an incoming order or quote on the opposite side of a quote trigger, which is at or crosses the quote trigger price. The second reason is if the product is closed or halted.

Can a participant in a Quote Trigger not receive a fill?

Yes. If the available customer volume is less than the number of quotes, then one or more quotes may not be filled. For example, a Sell Order for 5 @ 1.00 hits 8 resting Quotes, each 10 @ 1.00 x 10 @ 1.20. This will result in one or more quotes not being filled.

Quote Locked Notification

What causes a quote locked scenario?

Quote locks are caused by the combination of two market maker quotes or orders crossing in the book. When this happens, a current market event showing a crossed book will be disseminated and then immediately quote locked notification messages will be sent to all the market makers involved in the quote lock. This provides the market maker with an opportunity to cancel or change their quotes in order to avoid trading with another market maker accidentally. The quote lock will last for n(10) seconds before trading out the crossed book.

There are a few important things to note about quote locks.

- If a non-market maker order comes in and crosses the book, it will be traded immediately against the quotes.
- If a market maker updates their quote and the updated quote results in a locked market again, a new quote lock will start.
- The quote-locked message will contain the acronyms of the market makers involved in the quote lock. It is possible to appear in the list or on both sides if the market maker has both I-orders and quotes involved in the lock.

Quote and Order Limits

CBOE regulates the number of quotes that can be generated at one time. When a quote is not accepted due to the quote limit being exceeded, you will receive a Quote Acknowledgement response that indicates that quote has not been accepted. The most common recovery action is to resubmit the quote. CBOEdirect will reject the entire block of quotes that puts the user over the threshold. Quote rate limits are configurable and subject to change.

The limitations for quoting are:

CBOE does not consider quote or mass quote cancels when calculating quote thresholds.

Hybrid (in session W_MAIN)

- 400 Quotes (products) per Mass Quote message
  (SEQUENCE_SIZE_EXCEEDED)
- 133 Mass Quote or Quote message calls per user per 1 second period
  (RATE_EXCEEDED)

- 4000 total quotes (products) per user per 3 second period **(QUOTE_RATE_EXCEEDED)**

CFE_MAIN (CBOE Futures Exchange) and **ONE_MAIN** (OneChicago)

- 12 Quotes (products) per Mass Quote message
- 250 Mass Quote or Quote message calls per user per five (5) second period
- 1000 total quotes (products) per user per five (5) second period

W_STOCK (CBSX-CBOE Stock Exchange)

- 100 quotes per one (1) second period

The limitations for orders are:

Hybrid (in session W_MAIN)
30 orders per one (1) second period

CFE_MAIN (CBOE Futures Exchange) and **ONE_MAIN** (OneChicago)

30 orders per one (1) second period

W_STOCK (CBSX-CBOE Stock Exchange)

200 orders per one (1) second period

Retrieving the Current Rate Limits

The *UserTradingParameters* interface was extended to include a new method, *getUserRateSettings* in the CMi V5 module. This method takes a session name and returns the current values of various rate limits. Each rate limit is returned as a generic struct containing a key and a value, both of which are string types. Currently, this method returns the quote rate limit, quote call limit, order rate limit, order call limit and book depth call limit. This feature will be available in April 2008.

```
interface UserTradingParameters : cmi::UserTradingParameters
{
cmiUtil::KeyValueStructSequence
getUserRateSettings(in string sessionName)
raises(
exceptions::SystemException,
exceptions::CommunicationException,
exceptions::AuthorizationException,
exceptions::DataValidationException
);
};
```

And example of the return data:

User settings for W_MAIN

quoteConstraints.rateMonitorWindow=133	Calls per second.
quoteConstraints.rateMonitorInterval=1000	Monitor window size in milliseconds for call rate.
quoteConstraints.quoteRateMonitorInterval=3000	Monitor window size in milliseconds for quote rate.
quoteConstraints.quoteRateMonitorWindow=4000	Quotes per monitor interval

quoteConstraints.rateMonitorQuoteSequenceSize=400 Quote block size
orderConstraints.rateMonitorWindow=100 Orders per monitor interval
orderConstraints.rateMonitorInterval=1000 Monitor window size in milliseconds for Order rate.
marketDataConstraints.rateMonitorWindow=1
marketDataConstraints.rateMonitorInterval=1000

Linkage Order Functionality

Firms can transmit new linkage orders via CBOE to the OLA hub and NBBO markets. The normal message for Linkage is the Principal (P) order. The path the order will take is from the market maker (terminal) to the market maker firm, through the firm's CMi interface with CBOE. CBOE will validate and pass the linkage order to the central OLA hub. The OLA hub will read the destination on the order, and OLA will transmit the linkage order to the NBBO exchange for Linkage order handling and execution. CBOE will allow inbound P orders for CBSX in the W_STOCK session.

The CMi user would use one of the Order Entry methods currently available in the Order Entry interface (in the cmi.idl). The differences between a P order and a non P order is the originator and the restrictions on the contingency type. Using either the acceptOrder() or acceptOrderByProductName() method the order struct is filled in the following manner:

Creating and submitting a Linkage Principal (P) Order using Existing OrderEntry Interface

P Order specific fields are in *cmiOrder::OrderEntryStruct anOrder:*

anOrder.orderOriginType = OrderOrigins.PRINCIPAL // cmiConstants.idl

anOrder.contingency.type= ContingencyTypes.IOC // cmiConstants.idl

anOrder.sessionNames[0]=W_MAIN

anOrder.extensions fields: // ExtensionsField in cmiConstants.idl
EXCHANGE_DESTINATION

Creating and submitting a Linkage Satisfaction (S) Order using Existing OrderEntry Interface

S Order specific fields are in *cmiOrder::OrderEntryStruct anOrder:*

anOrder.orderOriginType = OrderOrigins.Satisfaction

anOrder congingency = type has to be IOC

anOrder.extension = in this field, the NBBO Agent needs to specify:

SATISFACTION_ALERT_ID // high:low format

This field will be used to associate an outbound S order with the satisfaction alert that the S order is based on.

EXCHANGE_DESTINATION

Note: S orders to not apply to stock.

Creating and submitting a Linkage Principal/Agency (P/A) Order using Existing OrderEntry Interface

S Order specific fields are in *cmiOrder::OrderEntryStruct anOrder:*

anOrder.orderOriginType = OrderOrigins.Satisfaction

anOrder congingency = type has to be IOC

anOrder.extension = in this field, the NBBO Agent needs to specify:

ASSOCIATED_ORDER_ID // high:low format

This field will be used to associate an outbound P/A order with a customer order on behalf of the P/A order.

EXCHANGE_DESTINATION

NBBO Agent Receiving Order Status

For P/A or S orders entered by the NBBO agent, the agent will get order status from the existing CMi callback CMIOrderStatusConsumer. S orders do not apply to stock.

For an order that is sent to the NBBO agent for manual handling, the NBBO agent will receive order status from the new CMi callback CMIIntermarketOrderStatusConsumer.

Order Status for Linkage P Orders

The CMi user continues to use the existing methods in *CMIOrderStatusConsumer* to receive order status message. The Cancel and Filled reports have new report type values and the extension field can contain additional information.

Cancel Reports for P Orders

New acceptCancelReport report types (cmiUtil::ReportType)

- Away exchange rejects a P order entered by a CMi user

  canceldReport. cancelReportType = ReportTypes. NEW_ORDER_REJECT

- Away exchange cancels a P order

  canceldReport. cancelReportType = ReportTypes. REGULAR_REPORT

- TPF can’t process the input P order, or FIX can’t deliver the P order or OLA HUB can’t deliver the P order

  canceldReport. cancelReportType = ReportTypes. NEW_ORDER_REJECT

- In the case of a partial fill, the away exchange may or may not send a CancelReport for remaining quantity. If the away exchange sends a CancelReport for remaining quantity, the report type is

  canceldReport. cancelReportType = ReportTypes. REGULAR_REPORT

The new valid linkage activity reasons for cancelReport.cancelReason(cmiUtil::ActivityReason) are listed in cmiConstants::ActivityReasons:

```
// The following are used for Linkage
    const cmiUtil::ActivityReason BROKER_OPTION = 100;
    const cmiUtil::ActivityReason DUPLICATE_ORDER = 103;
    const cmiUtil::ActivityReason EXCHANGE_CLOSED = 104;
    const cmiUtil::ActivityReason GATE_VIOLATION = 105;
    const cmiUtil::ActivityReason INVALID_ACCOUNT = 106;
    const cmiUtil::ActivityReason INVALID_AUTOEX_VALUE = 107;
    const cmiUtil::ActivityReason INVALID_CMTA = 108;
    const cmiUtil::ActivityReason INVALID_FIRM = 109;
    const cmiUtil::ActivityReason INVALID_ORIGIN_TYPE = 110;
    const cmiUtil::ActivityReason INVALID_POSITION_EFFECT = 111;
    const cmiUtil::ActivityReason INVALID_PRICE = 112;
    const cmiUtil::ActivityReason INVALID_PRODUCT = 113;
    const cmiUtil::ActivityReason INVALID_PRODUCT_TYPE = 114;
    const cmiUtil::ActivityReason INVALID_QUANTITY = 115;
    const cmiUtil::ActivityReason INVALID_SIDE = 116;
    const cmiUtil::ActivityReason INVALID_SUBACCOUNT = 117;
    const cmiUtil::ActivityReason INVALID_TIME_IN_FORCE = 118;
    const cmiUtil::ActivityReason INVALID_USER = 119;
    const cmiUtil::ActivityReason NOT_FIRM = 121;
    const cmiUtil::ActivityReason MISSING_EXEC_INFO = 122;
    const cmiUtil::ActivityReason NO_MATCHING_ORDER = 123;
    const cmiUtil::ActivityReason NOT_NBBO = 125;
    const cmiUtil::ActivityReason OTHER = 128;
    const cmiUtil::ActivityReason PRODUCT_HALTED = 130;
    const cmiUtil::ActivityReason PRODUCT_IN_ROTATION = 131;
    const cmiUtil::ActivityReason STALE_EXECUTION = 132;
    const cmiUtil::ActivityReason STALE_ORDER = 133;
    const cmiUtil::ActivityReason ORDER_TOO_LATE = 134;

// Currently used for TPF linkage; in future may be used for CBOEdirect
    const cmiUtil::ActivityReason UNKNOWN_ORDER = 137;
```

```
const cmiUtil::ActivityReason INVALD_EXCHANGE = 138;
const cmiUtil::ActivityReason TRANSACTION_FAILED = 139;
const cmiUtil::ActivityReason NOT_ACCEPTED = 140;

// Used for linkage when cancel reason is not provided (could be user cancel or cancel remaining)
const cmiUtil::ActivityReason AWAY_EXCHANGE_CANCEL = 199;

// Linkage Business Message Reject codes
const cmiUtil::ActivityReason LINKAGE_CONDITIONAL_FIELD_MISSING = 900;
const cmiUtil::ActivityReason LINKAGE_EXCHANGE_UNAVAILABLE = 901;
const cmiUtil::ActivityReason LINKAGE_INVALID_MESSAGE = 902;
const cmiUtil::ActivityReason LINKAGE_INVALID_DESTINATION = 903;
const cmiUtil::ActivityReason LINKAGE_INVALID_PRODUCT = 904;
const cmiUtil::ActivityReason LINKAGE_SESSION_REJECT = 905;
```

The new data that can be found in the Extensions Field (canceldReport.order.extensions) are:

AUTO_EXEC_SIZE

EXCHANGE_DESTINATION

Fill Reports

The CMi user will receive a fill report when the NBBO exchange fills the order.

New acceptFilledReport status report types(*cmiutil::ReportType*)

filledReport.filledReport[0].FillReportType = ReportTypes.REGULAR_REPORT;

- FilledReport.filledReport[0].extensions
  - TEXT
  - AWAY_EXCHANGE_EXEC_ID
- Field in filledReport.order.extensions
  - AUTO_EXEC_SIZE
  - EXCHANGE_DESTINATION

Listed below are detailed description for each activity reason code:

CMI value	Meaning

905	BusinessRejectReason - Other (Used if a message relayed by the Hub received a session-level Reject from the destination participant.)
903	BusinessRejectReason - Unknown ID (Invalid DeliverToCompID)
904	BusinessRejectReason - Unknown Security
902	BusinessRejectReason – Bad message.
901	BusinessRejectReason - Application not available (The session between the Hub and the destination participant is unavailable or the destination exchange's order handling application is down.)
900	BusinessRejectReason - Conditionally required field missing
100	Broker option (admission of non-compliance)
113	Unknown, invalid, or ineligible (for linkage) symbol (series, expiration, strike)
104	Exchange closed (An executing participant's order handling system receives an order when the market is closed)
115	Order Exceeds Limit
134	Too Late to enter
137	Unknown order
103	Duplicate Order (e.g. dupe of ClOrdID)
133	Stale Order (Time-to-live of received order has expired)
131	Instrument state is invalid for Linkage (the receiver is in Rotation)
121	Instrument state is invalid for Linkage (the receiver is in Non-Firm mode)
130	Instrument state is invalid for Linkage (the receiver has the instrument halted)
125	Not at NBBO
112	Reference Price is out of bound
108	Unknown clearing firm
117	Sub-Account ID missing
107	Invalid Auto-Ex value
119	Account missing
118	TimeInForce missing/invalid
111	OpenClose missing/invalid
109	Exec Broker missing
106	Clearing account missing
122	Execution information missing
105	Order received too soon (does not meet gate requirement)
110	OrderCapacity missing/invalid
120	Late print to OPRA Tape
126	Communications delays to OPRA
102	Manual (Crowd) Trade
129	Processing problems at market center
114	Complex order
136	Trade Rejected
135	Trade Busted/Corrected
127	Original order rejected
124	Cancel due to non-block trade

Stop and Stop Limit Order Functionality

Stop and Stop Limit contingency orders are initiated when certain market conditions occur. This section describes session availability and triggers for these types of orders.

Session Availability

Contingency	CBOE Open Outcry (W_MAIN)	OneChicago Securities Futures (ONE_MAIN)	CBOE Futures Exchange (CFE) (CFE_MAIN)	CFE Options on Futures (COF_MAIN)	CBOE Stock (W_Stock)
Stop (Stop Loss)	Yes	Yes	Yes	Yes	No
Stop Limit	Yes	Yes	Yes	Yes	No

Triggering of STOP and STOP LIMIT Orders

Both STOP and STOP LIMIT orders are triggered in the same manner. The difference between the two is that when triggered, a STOP order turns into a market order and a STOP LIMIT order turns into a limit order.

Buy STOP orders will be triggered when one of two market conditions occurs.

- A trade occurs in the product that is equal to or greater than the trigger price (contingency price) of the STOP order.
- The best bid price in the market (top of book) is equal to or greater than the trigger price (contingency price) of the STOP order.

Sell STOP orders will be triggered when one of two conditions occurs.

- A trade occurs in the product that is equal to or less than the trigger price (contingency price) of the STOP order.
- The best ask (offer) price in the market (top of book) is equal to or less than the trigger price (contingency price) of the STOP order.

STOP Orders

OrderEntryStruct {

cmiUtil::PriceStruct price; ---whole (*ignored*), fraction (*ignored*), priceType (***market***)

(above signifies that a market order will be entered after the STP is triggered)

OrderContingencyStruct contingency;

{

ContingencyType type; (***STP)***

cmiUtil::PriceStruct price; ---whole (*xxx*), fraction (*yyyyyyyy)* (*this is the trigger price*)

priceType (***limit***)

long volume (*ignored*);

}

}

STOP LIMIT Orders

OrderEntryStruct {

cmiUtil::PriceStruct price; --- whole (*xxx*), fraction (*yyyyyyyy)*, priceType (***limit***)

(above is the limit price of the order that will be entered after the STP LIM is triggered)

OrderContingencyStruct contingency;

{

ContingencyType type; (***STP_LIMIT***)

cmiUtil::PriceStruct price; ---whole (*xxx*), fraction (*yyyyyyyy)* (*this is the trigger price*)

priceType (***limit***)

long volume; (*ignored*)

}

}

Spread Order Functionality

A “Spread Order”, “Strategy Order”, or “Complex Order” is an Order to trade a multi-legged (strategy) product. CMi users would use one of the Order Entry methods currently available in the OrderEntry interface (in the cmi.idl) to enter spread orders.

Creating and submitting a Cross-Product Spread Order using Existing OrderEntryInterface

A Cross-Product Spread (CPS) order is an order to Buy or Sell a stated number of Option contracts, and also Buy or Sell the same underlying Stock or Exchange-Traded Fund (ETF) share (loosely, the Equity contracts), generally, in an amount that would offset (on a one-for-one basis) the option position.

A cross-product spread order needs leg-specific fields set, and so would need to use one of the OrderEntry methods that has cmiOrder::LegOrderEntryStructSequence parameters.

Leg specific fields are in cmiOrder::LegOrderEntryStructSequence legEntryDetails

(For documentation clarity, the Option Leg will be referred to as legEntryDetails[0], and the Equity Leg will be referred to as legEntryDetails[1]).

LegEntryDetails[0].clearingFirm = this must be the clearing firm used for the option product.

LegEntryDetails[1].clearingFirm = this must be the clearing firm used for the equity product.

Creating and submitting a "Delta Neutral" Spread Order using Existing OrderEntryInterface

A "Delta Neutral" Spread order needs leg-specific fields set, and so would need to use one of the OrderEntry methods that has cmiOrder::LegOrderEntryStructSequence parameters.

Order specific fields are in cmiOrder::OrderEntryStruct anOrder:

anOrder.price.whole = 0

anOrder.price.fracion = 0

anOrder.price.type = PriceTypes.NO_PRICE

LegEntryDetails[0].mustUsePrice.type = must be of PriceTypes.VALUED

LegEntryDetails[1].mustUsePrice.type = must be of PriceTypes.VALUED

Strategy Fill/Cancel Reports

Transaction Sequence Number Change for Strategy Fill/Cancel Reports

A recent change was made to the API that impacts the Transaction Sequence Number field, transSequenceNumber, in strategy fill reports and strategy order cancel reports. With this change, the transSequenceNumber field is duplicated in fill/cancel reports of the strategy itself and fill/cancel reports of each leg. Previously, the transSequenceNumber was different in the strategy fill/cancel report and each leg fill/cancel report.

If the API user is using the transSequenceNumber only to filter out duplicate fill reports, each leg fill report could be duplicated. CBOE suggests the API user utilize a combination of orderID, tradeID and transSequenceNumber to uniquely identify each fill report.

CBOE 2 (C2)

The CBOE exchange, CBOE 2 (C2) is a fully electronic options exchange supporting a maker-taker pricing model. C2 functionality is similar to the existing CBOE (W_MAIN) functionality with some key differences. The table below details the differences between CBOE and CBOE 2 with respect to the CMi API.

Interface	CBOE	CBOE 2	Comments
const cmiOrder::ContingencyType	Contingencies that are not accepted: -Not Held (NH)	Contingencies that are not accepted: -Minimum volume (MIN) -Market-if-Touched (MIT)	In C2 'Not Held' orders must be ACCEPTED where in the contingency is ignored and the orders are handled like regular

Interface	CBOE	CBOE 2	Comments
		-With Discretion (WD) -Midpoint Cross -Cross -Tied Cross -Autolink Cross -Autolink Cross Match -Cross_Within -Tied_Cross_Within	orders without any contingency.
cmiSession::TradingSessionName sessionName	W_MAIN	C2_MAIN	
const cmiUser::Exchange	CBOE	CBOE2	
const BillingTypeIndicator	Billing type indicators: Maker=A Taker=R Flash_Response=E Flash=F Cross=C Linked_AWAY=X Linked_AWAY_Response=L Opening=O ODD_LOT_FLASH=N ODD_LOT_RESPONSE=B	Billing type indicators: Maker=A Taker=R Flash_Response=E Flash=F Cross=C Linked_AWAY=X Linked_AWAY_Response=L Opening=O ODD_LOT_FLASH=N ODD_LOT_RESPONSE=B RESTING=Q CROSS_PRICE_IMP=S FLASH_PRICE_IMP=T FLASH_RESPONSE_PRICE_IMP=U MAKER_TURNER=V RESTING_TURNER=W	CBOE will be enhanced to include all the values in CBOE 2.

CBOE Stock Exchange (CBSX)

The CBOE Stock Exchange (CBSX) has its own trading session, W_STOCK, on CBOE*direct* for equity trading. CBOE*direct* supports stock order contingency types (cmiConstants): Intermarket Sweep orders (ISO), Reserve orders (also available for options in the W_MAIN session)and Cross orders. Firms use the OrderEntry.acceptOrder interface to transmit ISO and Reserve orders.

const cmiOrder::ContingencyType INTERMARKET_SWEEP = 15; // Intermarket sweep order (ISO)

Intermarket Sweep Orders (ISO) are treated as IOC orders but will trade against the book without regard to the NBBO. Trades executed as a result of these orders are exempted from RegNMS trade-through rules. ISO orders may only be sent if the sender has simultaneously sent orders to any other markets with protected quotes priced better than the limit price on the ISO.

const cmiOrder::ContingencyType RESERVE = 16; // Reserve order

A reserve order has two quantities associated with it, order quantity and display quantity. Only the display quantity is visible as CBOE's book. The remaining quantity (order quantity - display quantity) is available to trade but not visible.

Reserve orders will only display a portion of the total quantity available. The maximum volume to display is specified in OrderEntryStruct.contingency.volume. Any quantity specified in OrderEntryStruct.originalQuantity greater than the contingency volume is considered reserved and will not be show to the marketplace. Reserved quantity is available to trade but is at a lower priority than displayed quantity. Reserved quantity will become displayed quantity only after an amount at least equal to the contingency volume minus one lot is traded. In other words, for a contingency volume of five lots, only after at least four lots have traded will the display quantity be refreshed from the reserved quantity. Anytime reserved quantity becomes displayed the order will lose its priority at that price. Only round lot quantities can be entered for both contingency volume and originalQuantity.

Cross Orders

Firms can send Cross orders as a pair using the OrderEntry.acceptCrossingOrder() interface or as two single orders via the OrderEntry.acceptOrder interface. Using OrderEntry.acceptCrossingOrder enables the user to send two orders (BUY and SELL) in one message.

When two single orders are submitted using the OrderEntry.acceptOrder interface, the second crossing order must be received within a certain time interval of receiving the first crossing order. There can only be one resting Cross order for a particular product. Further, a cross order can only trade with another cross order of the same contingency type (i.e. a 'MIDPOINT_CROSS' will only trade with another 'MIDPOINT_CROSS' that has the same price, but opposite side). In all cases, the matching orders (if not arriving as a pair) have to arrive within a specified time otherwise the original cross order will be cancelled.

const cmiOrder::ContingencyType MIDPOINT_CROSS = 17; // Mid Point Cross

Mid Point cross: These two orders can arrive as a pair (together) or one after the other. In either case they will need the same contingency. A mid-point-cross trades at the middle of current NBBO and will trade at ½ cent increments. Both orders must be for the same size.

const cmiOrder::ContingencyType CROSS = 18; // Cross

These two orders can arrive as a pair (together) or one after the other. In either case they will need the same contingency. If they are for the same price and size they will trade against each other immediately as long as the price is at or within CBOE's quote and the NBBO. If the trade price is equal to the CBOE's best market and the market includes public customer orders, the cross order must be: X shares or more; be for a dollar amount greater than or equal to $Y; and larger than any public customer interest at that price. Both orders must be for the same price and size and neither order will execute unless both orders are received.

const cmiOrder::ContingencyType TIED_CROSS = 19; // Tied cross (CURRENTLY UNSUPPORTED)

Tied Cross: Similar to Cross orders except that they can trade at or better than CBOE's current market and NBBO trade through is allowed.

const cmiOrder::ContingencyType AUTOLINK_CROSS = 20; // Auto link cross

Auto Link Cross is an order that will be Autolinked if CBOE is not the NBBO and the order is tradable at other markets. If there is remaining quantity, then this order will trade against an AutoLink_Cross_Match order (see below). Autolink_cross orders and autolink_cross_match orders can route in a single paired message or as two separate

orders. Both orders must be for the same price but do not necessarily need to be for the same size. The autolink_cross order will not execute unless an autolink_cross_match order is received.

const cmiOrder::ContingencyType AUTOLINK_CROSS_MATCH = 21; // Auto link cross match
Users have to submit two orders for AUTOLINK crosses. One is AUTOLINK_CROSS and the second one is AUTOLINK_CROSS_MATCH. If away markets are better, then AUTOLINK_CROSS will first sweep away markets and the CBOE book. If there is still quantity remaining, then it will trade against the AUTOLINK_CROSS_MATCH.

const cmiOrder::ContingencyType CROSS_WITHIN = 22; // (CURRENTLY UNSUPPORTED)
Similar to Cross orders except that the CROSS_WITHIN order shall trade at or better than the NBBO and within CBOE's market.

const cmiOrder::ContingencyType TIED_CROSS_WITHIN = 23; // (CURRENTLY UNSUPPORTED)
Similar to TIED_CROSS orders except that the TIED_CROSS_WITHIN order shall trade at or better than the NBBO and within CBOE's market.

Drop Copies

CMi users that login as a Firm Display user (const cmiUser::UserRole FIRM_DISPLAY = 'R') in the CBSX session will receive drop copies (order/quote fills, busted order/quote fills) for their clearing firm and exchange. This functionality is available on exchange basis. Clearing firms that login for CBSX drop copies will not get drop copies from other exchanges.

Upon logging into a single CAS, the Firm Display user will receive all unacknowledged and real-time drop copies for his/her clearing firm and exchange. If the Firm Display user is not logged in during an occurrence of fill/bust or bust re-instate events, the missing events will be delivered after the users next login. Those reports will be delivered as Poss Resends. Missed events up to one day old will be delivered to the user.

CBOE Trade Type Indicators

Trade type billing indicators are defined in the interface, BillingTypeIndicator, in the cmiConstants.idl. Currently, BillingTypeIndicators are only available for CBSX. The trade types included: maker, taker, cross, flash, flash response, cross, link away, linked away response and opening. Below are the CMi constant values for each BillingTypeIndicator.

```
module cmiConstants


interface BillingTypeIndicators
  {
    const BillingTypeIndicator MAKER              = 'A';
    const BillingTypeIndicator TAKER              = 'R';
    const BillingTypeIndicator FLASH_RESPONSE           = 'E';
    const BillingTypeIndicator FLASH                    = 'F';
```

```
    const BillingTypeIndicator CROSS                = 'C';
    const BillingTypeIndicator LINKED_AWAY          = 'X';
    const BillingTypeIndicator LINKED_AWAY_RESPONSE = 'L';
    const BillingTypeIndicator OPENING              = 'O';
    const BillingTypeIndicator ODD_LOT_FLASH        = 'N';
    const BillingTypeIndicator ODD_LOT_RESPONSE     = 'B';
    const BillingTypeIndicator RESTING              = 'Q';
    const BillingTypeIndicator CROSS_PRICE_IMP      = 'S';
    const BillingTypeIndicator FLASH_PRICE_IMP      = 'T';
    const BillingTypeIndicator FLASH_RESPONSE_PRICE_IMP
                = 'U';
    const BillingTypeIndicator MAKER_TURNER         = 'V';
    const BillingTypeIndicator RESTING_TURNER       = 'W';
};
```

Below are descriptions of the trade type indicators.

Maker: refers to the person adding liquidity to the market, by basically having an order or quote resting in the book to be traded against. (i.e. they are establishing the price)

Taker: refers to the person taking liquidity from the market, by basically sending an order to trade against the book. (i.e. they are coming in and taking out the best price)

Flash: refers to an order that is being presented to the dealers for a short-term auction for step-up, before the order is routed to an away exchange for a fill.

Flash Response: refers to the dealer responding to a flash and effectively stepping up to improve the CBSX market to the prevailing price and fulfilling the customer here.

Linked Away: refers to an order that was sent to another market for execution.

Linked Away Response: refers to the response from the other exchange filling the CBSX order sent to them.

Opening Trade: refers to all executions that take place as part of the opening rotation process itself.

Cross: refers to a trade whereby both buyer and seller are represented on a single transaction. Thus, neither is really a maker or taker per se, but rather virtually meet one another.

ODD_LOT_FLASH: used for CBSX odd lot orders or the odd lot portion of a mixed lot order that is being flashed.

ODD_LOT_RESPONSE: is used for all responses to odd lot orders that are being flashed.

Fill Reports

Fill reports will include the billing type indicators in const ExtensionField BILLING_TYPE = "billingType" where “billingType” equals A, R, E, etc. This information will be reported in the FilledReportStruct.extensions.

Names Later

Traders can indicate a non real-time clearing (“Names Later”) order using const OptionalDataPhrase NAMES_LATER = “NLTR”. If the order gets executed, the trade will not automatically clear. The trader will have to provide the information about the contra party at the end of the day before the trade clears.

```
module cmiConstants


interface OptionalDataFields
  {
    const OptionalDataField NAMES_LATER    = "NLTR";
  };
```

CBOE Futures Exchange (CFE)

The CBOE Futures Exchange (CFE) has its own trading session, CFE_MAIN, on CBOE*direct* to support the listing of futures.

Special Considerations for CFE Orders

- The only CFE user role that can enter orders of origin "M" is the market-maker role. The broker-dealer role is not allowed to enter "M" orders into the CFE_MAIN session.
- Market-maker users may not clear their trades in the market-maker origin.
  - The market-maker’s firm must notify the OCC whether the market-maker's trades will clear in the Market-Maker, Firm, or Customer origin at OCC.
  - The CBOEdirect Administrator (CBOE Help Desk) will set up the market-maker's profile to define the default origin of either M (market-maker), V (CFE customer) or E (CFE firm) to be populated for market-maker trades.
  - These users must also identify their bookkeeping account number at their clearing firm, which will be entered into the default Subaccount field in the OrderEntryStruct on market-maker orders.
  - For users whose default origin is M, the Subaccount field in the OrderEntryStruct must = the market-maker acronym.
- OCC does not facilitate large trader reporting for customer positions. Market-maker user types who clear in the customer origin will report large trader positions to the exchange and CFTC via SIAC.
- Trades may now be flagged as give up trades at the time of order entry. Users wishing to do this, without specifying the firm that the trade will be given up to, must enter a CFE:G

in the CMTA and CMTA Exchange fields in the OrderEntryStruct. Users who know the give up firm may enter that firm number into the CMTA field (e.g. CFE:690).

- Post trade processing will be conducted at The Clearing Corporation. These matched trades will be transferred to the OCC at the end of the day. Firms will not be able to use their ITP on line screens to view or edit CFE trades. Firms will be able to use existing connectivity to The Clearing Corporation or The Clearing Corporation's web-based on-line screens to view and edit trades. Each day's Final Trade Register will only come directly from OCC.
- Since The Clearing Corporation does not support an Optional Data field, the OptionalData field in the OrderEntryStruct will not flow from CBOE to OCC. If users enter OptionalData on a CFE order, they will receive that data back in the fill report from CBOEdirect, but they will not see it on any data that they receive from The Clearing Corporation or OCC.

CFE Options on Futures (COF) Trading

CBOE API Volume 2 CMi Programmer's Guide to Interfaces and Operations (COF_MAIN) is a trading session that allows users to trade options on futures on the CBOE Futures Exchange (CFE). Options on futures is Screen Based Trading only and does not include Hybrid functions. The session is currently set to pre-open at 7:58a.m., open at 8:30a.m. and close at 3:02p.m.

COF Details

CFE users who have access to CBOE*direct* will continue to use the same login and password. The CBOE help desk will set user enablements to allow trading in the COF_MAIN session.

Market makers entering quotes for options on futures will use the existing cmiQuote::QuoteEntryStruct. Users will need to specific the options fields. Required fields include: Underlying Symbol (Futures contract), Series (Month, Strike, Year, Call/Put Code), Price, Side (Bid, Ask), Quantity, Executing Firm and Broker Acronym.

Market makers entering orders for options on futures will use the existing cmiOrder::OrderEntryStruct. Users will need to specific the options fields. Required fields include: Order Type, Underlying Symbol (Futures contract), Series (Month, Strike, Year, Call/Put Code), Side (Buy, Sell), Price, Quantity, Account Designation, Executing Firm, Broker Acronym, Time In Force, Account Type (Origin), and Optional Data Field (if applicable).

CFE eligible Trading Privilege Holders and Authorized Traders that are not market makers may enter only orders, not quotes, into CBOE*direc*t. Users will need to specific the options fields. Required fields include: Order Type, Underlying Symbol (Futures contract), Series (Month, Strike, Year, Call/Put), Side (Buy, Sell), Price, Quantity, Time in Force (Day or Good-‘til-Canceled), Session, Account Type (Origin), Executing Firm, Broker Acronym, and Customer Account Number (Sub-account ID).

Options and Futures Clearing Information

This section provides clearing information for both Options and Futures.

CMi Options Order Clearing Information in cmiOrder::OrderEntryStruct

Field Name	Sample	Description
account	ABC or QAB	For market-makers, this typically would be either the joint account (often called q-account) or the market-maker three-letter badge acronym. Passed through to OCC. Required for Market-Maker and DPM roles in all sessions. For Market-Maker and DPM roles, CBOE validates the value of this field on inbound orders against the CBOE Membership system. For Market-Maker and DPMs, user cannot use more than one account per class. Optional for Broker-Dealer and Firm roles. CBOE performs no validation checks on the value of this field for Broker-Dealer and Firm roles. Exact size is 3 and data type is alpha only.
cmta	*Exchange*: CBOE *Firm#*: 123	The CMTA (Clearing Member Trade Agreement) field is used to designate an OCC clearing firm if it is different from the executingOrGiveUpFirm. CBOE performs no validation checks on the CMTA field against the CBOE Membership system. This field is optional for all roles in all sessions. CMTA is comprised of two components: an e*xchangeFirmStruct* which contains the exchange code and CMTA *firmNumber*. If you use CMTA, then you must use submit both of these two components. The e*xchange string* is the exchange on which your order will trade. The exchange portion of the CMTA field is alpha only. The *firmNumber* is the OCC clearing firm where the order will clear. The firmNumber portion of the CMTA field is numeric only. Even though the maximum size for the firmNumber component is 5, CBOE will read the first three numbers of this field to use as the OCC clearing firm. In other words, if the desired CMTA firm at the OCC is “123”, do not send “00123”, send “123”.
correspondentFirm	ABC or ABCD	The correspondent firm field is used by the executing give up firm to differentiate the firm or system sending the order. The 1st three characters of this field are mapped to the optional data field on the CBOE Trade Match (CTM) record. This field has no impact on the clearing of the trade. This field is optional. CBOE performs no validation checks on the correspondentFirm field against the CBOE Membership system. Maximum size is 4 characters and data type is uppercase alpha only.

Field Name	Sample	Description
executingOrGiveUpFirm	CBOE:123 or 123	This is the CBOE clearing firm that is representing the order in live trading (post trade processing firm). If no CMTA firm is present in the order, then the executingOrGiveUpFirm represents the OCC clearing firm where the order will clear. This field is required for all orders sent to CBOE for all roles in all sessions regardless of whether a CMTA firm is given or not. CBOE performs validation checks of executingOrGiveUpFirm against the CBOE Membership system on options orders routed to the W_MAIN session. Broker-Dealer and Firm roles must choose from a list of pre-approved and pre-configured executingOrGiveUpFirms and the Market-Maker and DPM roles must use default executingOrGiveUpFirm only. executingOrGiveUpFirm is comprised of two components: an *ExchangeFirmStruct* which contains the Exchange code and *firmNumber*. The *Exchange string* is the exchange on which your order will trade. The Exchange portion of the executingOrGiveUpFirm field is alpha only. If there is no CMTA given in the order, then the executingOrGiveUpFirm *firmNumber* will be the OCC clearing firm where the order will clear. If there is a CMTA given in the order, then the firmNumber is the CBOE clearing firm that is representing the order in live trading (post trade processing firm). The firmNumber portion of the CMTA field is numeric only. Even though the maximum size for the firmNumber component is 5, CBOE will only read the first three numbers of this field to use as the executingOrGiveUpFirm. In other words, if the desired executingOrGiveUpFirm firm is "123", do not send "00123", send "123".
optionalData	M:ABC ABC123ABC	**Orders of origin Customer ("C"):** The first four characters are reported to the last four characters of CBOE Trade Match Optional Data field. These four characters are reported to OCC. Do not put "C:" in this field. **Orders of origin In-Crowd Market-maker ("I")** This field is not required. Do not put "I:" in this field. In-Crowd Market-Maker (ICM) options orders of origin 'I', similar to two-sided quotes, take their clearing information from the market-maker profile in the CBOE System Administrator GUI (SAGUI). **Orders of origin Market-maker ("M") and Away Market-maker ("N")** This field is ***required*** for all options orders of origin "M" or "N" that are sent to the CBOE. This contains data that will be passed on to the CBOE Trade Match system (CTM) and will be part of clearing information sent to the OCC. The data is specific to each member firm. For "M" and "N" orders routed to the CBOE Trading Floor (W_MAIN session), this field should contain the Market Maker Account (Q Account, joint account, or market maker acronym). If a subaccount is also used, then it must be supplied as well. If a firm sends an origin of "N", then this field must begin with the characters "M:", not "N:". **Preferred DPM** Firms that give one DPM priority in participating in a trade use this field. Firm is specified as **P:*firm*;** and can coexist with other data that may be present in this field. *"Firm"* is the CBOE firm acronym as listed in the Order Test Plan. Please note that the colon : and semi-colon ; are both mandatory. **Linkage**: This field is *not* used for Linkage.

Field Name	Sample	Description
		Strategy: For clearing purposes, this field must be populated for strategy orders not the leg struct. **Futures**: This field stays with the order for the life of the order. The first 16 bytes go to the OCC. This field is optional for all roles for futures orders. Maximum size is 128 characters and data type is alphanumeric. Do not send "M:" account information like is used for options clearing.

Field Name	Sample	Description
originator	ABC	This field would only be used for orders of origin “M”, “I”, and “N”. This field is comprised of two components: an *Exchange* string which contains the Exchange code and *acronym*. The *Exchange* string is the exchange on which the market-maker will clear the trade. The Exchange portion of the originator field is alpha only. The *acronym* is the three-letter acronym (“badge”) of the market-maker who originates the order. This field will typically be three characters (occasionally two). **Orders of origin “M”** Orders of origin “M” entered by a broker-dealer role must supply the three-letter acronym (“badge”) of the market-maker. If a broker-dealer role submits an options order of origin “M” on behalf of a CBOE market-maker, then the three-letter (all alpha, all caps) MM acronym must go into either the originator field (tag 9465 in FIX) or the originator portion (positions 13-15) of the optional data field (tag 9324 in FIX). The market-maker role does not have to enter the originator field when entering orders of origin "M". However, if a market-maker role wishes to enter the originator field when entering orders of origin "M", then it may enter the originator acronym into either the originator field (tag 9465 in FIX) or the originator portion (positions 13-15) of the optional data field (tag 9324 in FIX). **Orders of origin “I”** Orders of origin “I” entered by a broker-dealer role must supply the three-letter acronym (“badge”) of the market-maker. If a broker-dealer role submits an options order of origin “I” on behalf of a CBOE market-maker, then the three-letter (all alpha, all caps) MM acronym must go into the originator field (tag 9465 in FIX). If a broker-dealer role submits an options order of origin “I” on behalf of a CBOE market-maker, then it may also if it wishes put the originator acronym in the originator portion (positions 13-15) of the optional data field (tag 9324 in FIX). The market-maker role does not have to enter the originator field when entering orders of origin "I". However, if a market-maker role wishes to enter the originator field when entering orders of origin "I", then it may enter the originator acronym into either the originator field (tag 9465 in FIX) or the originator portion (positions 13-15) of the optional data field (tag 9324 in FIX). **Orders of origin “N”** Orders of origin “N” entered by a broker-dealer role must supply the three-letter acronym (“badge”) of the market-maker. If a broker-dealer role submits an options order of origin “N” on behalf of a non-CBOE market-maker (e.g. a CBOE BD role enters an order on behalf of an AMEX market-maker), then the MM acronym must go into the originator portion (positions 13-15) of the optional data field (tag 9324 in FIX) and not the originator field (tag 9465 in FIX). If a broker-dealer role submits an options order of origin “N” on behalf of a non-CBOE market-maker at another exchange, and enters any value into the originator field (tag 9465 in FIX), then CBOE will reject the order. CBOE does not allow the market-maker role to enter orders of origin “N”.
subaccount	AB2, ABC, ABC123, QA12, QAB123	CBOE performs no validation checks on subaccount against the CBOE Membership system. It is optional for all roles. Maximum size is 6 and data type is alphanumeric. For Broker and Firm roles, if subaccount is used then the account field is not required.

CMi Futures Order Clearing Information in cmiOrder::OrderEntryStruct

Field Name	Sample	Description
account	ABC or QAB	For market-makers, this typically would be either the joint account (often called q-account) or the market-maker three-letter badge acronym. Passed through to OCC. Required for Market-Maker and DPM roles in all sessions. For Market-Maker and DPM roles, CBOE validates the value of this field on inbound orders against the CBOE Membership system. For Market-Maker and DPMs, user cannot use more than one account per class. If desired, this field can be the same value as subaccount. Optional for Broker-Dealer and Firm roles. CBOE performs no validation checks on the value of this field for Broker-Dealer and Firm roles. Exact size is 3 and data type is alpha only.
cmta	*Exchange*: CBOE *Firm#*: 123	The CMTA (Clearing Member Trade Agreement) field is used to designate an OCC clearing firm if it is different from the executingOrGiveUpFirm. CBOE performs no validation checks on the CMTA field against the CBOE Membership system. This field is optional for all roles in all sessions. CMTA is comprised of two components: an e*xchangeFirmStruct* which contains the exchange code and CMTA *firmNumber*. If you use CMTA, then you must use submit both of these two components. The e*xchange string* is the exchange on which your order will trade. The exchange portion of the CMTA field is alpha only. The *firmNumber* is the OCC clearing firm where the order will clear. The firmNumber portion of the CMTA field is numeric only. Even though the maximum size for the firmNumber component is 5, CBOE will read the first three numbers of this field to use as the OCC clearing firm. In other words, if the desired CMTA firm at the OCC is “123”, do not send “00123”, send “123”.
correspondentFirm	ABC or ABCD	The correspondent firm field is used by the executing give up firm to differentiate the firm or system sending the order. The 1st three characters of this field are mapped to the optional data field on the CBOE Trade Match (CTM) record. This field has no impact on the clearing of the trade. This field is optional. CBOE performs no validation checks on the correspondentFirm field against the CBOE Membership system. Maximum size is 4 characters and data type is uppercase alpha only.

Field Name	Sample	Description
executingOrGiveUpFirm	*Exchange:* CBOE, CFE, or ONE *Firm:*123	This is the CBOE clearing firm that is representing the order in live trading (post trade processing firm). If no CMTA firm is present in the order, then the executingOrGiveUpFirm represents the OCC clearing firm where the order will clear. This field is required for all orders sent to CBOE for all roles in all sessions regardless of whether a CMTA firm is given or not. CBOE performs validation checks of executingOrGiveUpFirm against the CBOE Membership system on options orders routed to all sessions. Broker-Dealer and Firm roles must choose from a list of pre-approved and pre-configured executingOrGiveUpFirms and the Market-Maker and DPM roles must use default executingOrGiveUpFirm only. This field is comprised of two components: an *ExchangeFirmStruct* which contains the Exchange code and *firmNumber*. The *Exchange string* is the exchange on which your order will trade. The Exchange portion of the executingOrGiveUpFirm field is alpha only. If there is no CMTA given in the order, then the executingOrGiveUpFirm *firmNumber* will be the OCC clearing firm where the order will clear. If there is a CMTA given in the order, then the firmNumber is the CBOE clearing firm that is representing the order in live trading (post trade processing firm). The firmNumber portion of the CMTA field is numeric only. Even though the maximum size for the firmNumber component is 5, CBOE will only read the first three numbers of this field to use as the executingOrGiveUpFirm. In other words, if the desired executingOrGiveUpFirm firm is “123”, do not send “00123”, send “123”.
optionalData	ThisIsBob sOrder12	This field stays with the order for the life of the order. The first 16 bytes go to the OCC. This field is optional for all roles for futures orders. Maximum size is 128 characters and data type is alphanumeric. Do not send “M:” account information like is used for options clearing.
Originator	*Exchange:* CBOE, CFE, or CME *Acronym:* ABC	This field is used for market-maker orders only. It is optional for OneChicago futures but not used for CFE futures. It would contain the three letter market-maker acronym. This field is comprised of two components: an *Exchange* string which contains the Exchange code and *acronym*. The *Exchange* string is the exchange on which the market-maker will clear the trade. The Exchange portion of the originator field is alpha only. The *acronym* is the three-letter acronym (“badge”) of the market-maker who originates the order. This field will typically be three characters (occasionally two).
Subaccount	AB2, ABC, ABC123, QA12, QAB123	Subaccount is required for CFE and OneChicago futures. It specifies the account into which the trade will clear. CBOE performs no validation checks on subaccount against the CBOE Membership system. Maximum size is 6 and data type is alphanumeric.

Expected Opening Price

The CMi expected opening price (EOP) constants have been enhanced to provide descriptive opening messages to be used by Market Makers to send in valid/tradable

quotes at the open. In addition to the current opening process, these messages will be sent prior to the opening. The messages will be sent at pre-determined intervals, currently planned for every 30 seconds. Three new EOP type messages have been defined.

(1) const cmiMarketData::ExpectedOpeningPriceType **PRICE_NOT_IN_BOTR_RANGE = 10**;

This EOP message type indicates that the potential opening price is not in the valid Best of the Rest (BOTR) range based on the current BOTR price, current potential opening quantity and the system configuration. The EOP message will provide users with the potential opening price and opening quantity.

(2) const cmiMarketData::ExpectedOpeningPriceType **NEED_MORE_BUYERS = 3**;

This EOP message type indicates that the current market is imbalanced and will need more buyers' contracts (n) at the opening price. The EOP message will provide users with the potential opening price (bid price) and the imbalanced quantity (n).

(3) const cmiMarketData::ExpectedOpeningPriceType **NEED_MORE_SELLERS = 2**;

This EOP message type indicates that the current market is imbalanced and will need more sellers' contracts (m) at the opening price. The EOP message will provide users with the potential opening price (ask price) and the imbalanced quantity (m).

The Hybrid openings have changed to calculate the opening range based on the midpoint of the Best quote (or I order) bid and the best quote (or I order) ask, minus/plus half the spread width for the price range. (Ex. Best bid 1.00, best ask 1.10 Opening Spread width .25 for non-LEAP option will calculate an opening range of .925 - 1.175 which will round to .90 - 1.20)

Example Price Not in BOTR range:

Opening quote range: 1.00 - 1.10 20 x 20

Order to sell 10 at the market; BOTR Range = 1.05 - 1.10

Message #10 sent to users will include the CBOE calculated opening price of 1.00 and the size of 10.

Example Need more buyers:

Opening quote range: 1.00 - 1.10 20 x 20

Order to sell 50 at the market

Message #3 sent to users will include the Bid price of 1.00 and the size of 30.

Example Need more sellers:

Opening quote range: 1.00 - 1.10 20 x 20

Order to buy 50 at the market

Message #2 sent to users will include the Ask price of 1.10 and the size of 30.

Client Application Access to the CAS

Logging onto the CAS

You must first logon to the CAS in order to obtain access services. The UserAccess service provides a logon() operation. There are several steps that must be completed prior to invoking the logon() operation.

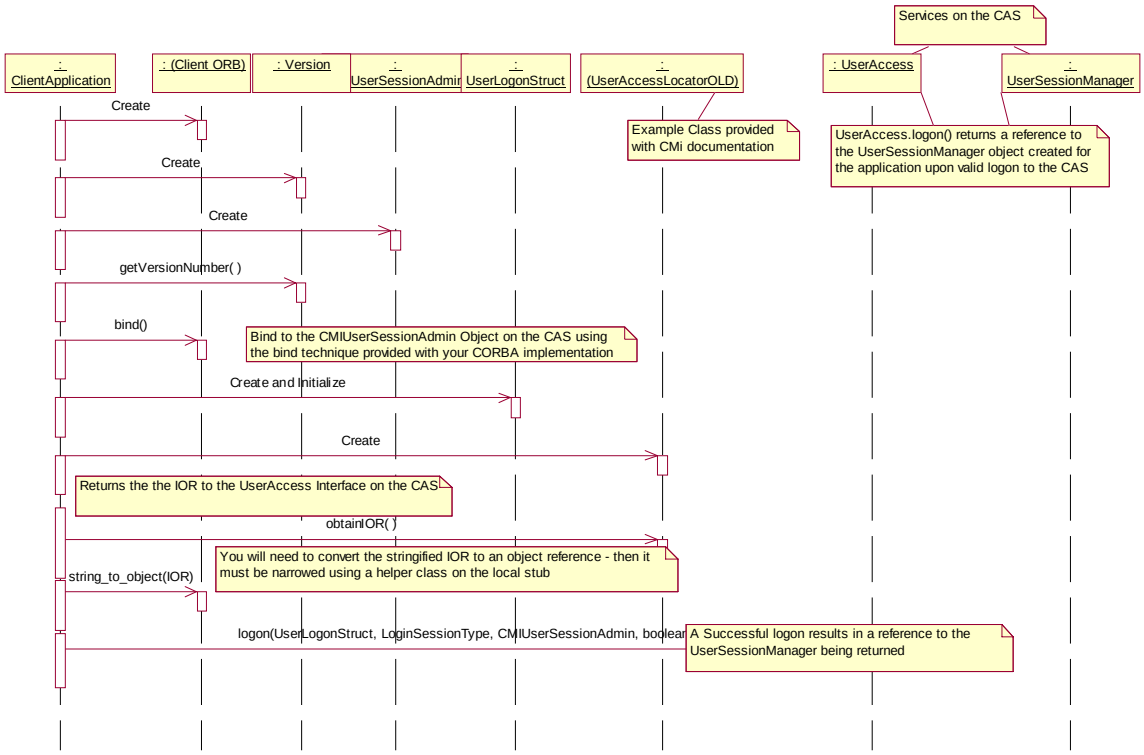
Step 1. You must obtain the object reference to the UserAccess object located on the CAS. To do this CBOE has provided source code for a UserAccessLocator class that contacts the CAS using the http protocol.

Step 2. You must create and populate a UserLogonStruct, which includes the userid, password, a CMI version, and session mode. In order to obtain the CMI version you must implement the CMI Version object in your application.

Step 3. You must implement a CMIUserSessionAdmin call back object in your CAS - including registering this object with your ORB - so that it is available for use by the CAS. The CMIUserSessionAdmin object has an operation for accepting a heartbeat from the CAS.

Step 4. Invoke the UserAccess.logon() operation, passing as arguments the UserLogonStruct and the CMIUserSessionAdmin object.

The example programs from CBOE have two example classes that encapsulate the aforementioned behavior. The com.cboe.examples.common.CASAccessManager class - manages all access to the CAS. Through the CASAccessManager you can logon to the CAS and access the CAS Services on an as needed basis. The com.cboe.examples.common.UserSessionAdminCallback class is an implementation of the com.cboe.idl.cmiCallback.CMIUserSessionAdmin interface.



Obtaining the User Access Initial Object Reference

UserAccessLocator is used to obtain the Interoperable Object Reference (IOR) for the UserAccessLocator.

CORBA objects are referenced by their Interoperable Object Reference. The CAS provides an IOR to access the UserAccess object via an http connection. This is commonly referred to as the bootstrap process - whereby the client application obtains an initial IOR via some out of band protocol, in this case HTTP. The TCP port number for the http server in the CAS is set to a default value of 8001. The TCP port number can be changed via a CAS configuration parameter - if need be to avoid potential conflicts with other software that maybe running with the CAS.

Each client application program that wants to access the CAS must first obtain the IOR for the UserAccess service. Once this is obtained the Client Application Program must narrow the reference and then bind to the UserAccess object in the CAS.

The CBOE has provided example source code in both Java and C++ to accomplish this task. The name of this example class is called UserAccessLocator.

The class is provided by CBOE in source code format.

Source Code for UserAccessLocator

```
package com.cboe.examples.common;

import java.net.*;
import java.io.*;
import java.util.*;

/**
 *
 * Access to the CAS is accomplished by acquiring an interoperable
 * object reference to the UserAccess object on the CAS using the
 * http protocol. This class communicates to a CAS - specified by
 * its IP address and and TCP Port number.
 * <br><br>
 * Copyright © 1999-2001 by the Chicago Board Options Exchange ("CBOE"), as an unpublished
work.
 * The information contained in this software program constitutes confidential and/or trade
 * secret information belonging to CBOE. This software program is made available to
 * CBOE members and member firms to enable them to develop software applications using
 * the CBOE Market Interface (CMi), and its use is subject to the terms and conditions
 * of a Software License Agreement that governs its use. This document is provided "AS IS"
 * with all faults and without warranty of any kind, either express or implied.
 *
 * @version 1.0
 *
 * @author Jim Northey
 *
 * @since Version 0.5
 */

public class UserAccessLocator {

  String ipAddress;
  int tcpPortNumber;

  final String IOR_REFERENCENAME = "/UserAccess.ior";
  final String DEFAULT_CAS_IP = "localhost";
  final int DEFAULT_CAS_TCP_PORT = 8003;

  String httpResponse; // Contains the HTTP Header information returned
                       // by the Server

  /**
   *  Construct a UserAccessLocator Object
   *
   * @param String ipAddress The IP Address for the CAS
   * @param String tcpPortNumber The Port Number of the HTTP Server on the CAS
   *
   */
  public UserAccessLocator(String ipAddress, int tcpPortNumber) {
    this.ipAddress = ipAddress;
    this.tcpPortNumber = tcpPortNumber;

  }
  /**
   *  Construct a UserAccessLocator Object using a default
   *  IP Address of DEFAULT_CASIP and TCP Port number of DEFAULT_CASTPCPORT.
   *
   */
  public UserAccessLocator() {
         this.ipAddress = DEFAULT_CAS_IP;
         this.tcpPortNumber = DEFAULT_CAS_TCP_PORT;
  }
  /**
   * Obtain the IOR for the UserAccess object of the CAS using the HTTP protocol
   *
   */
  public String obtainIOR() throws com.cboe.exceptions.CommunicationException {

      String ior;

      int numFields = 0;
```

```
      URLConnection conn;

      StringBuffer httpRespBfr = new StringBuffer();

      try {
        URL url = new URL("http",ipAddress,tcpPortNumber,IOR_REFERENCENAME);

        conn = url.openConnection();

        //
        // Skip over the headers in the return message
        // Save the headers in the event the user wants
        // to access them
        //

        String s = null;
        for(numFields=0; ; numFields++) {
            s = conn.getHeaderField(numFields);
            if (s == null) break;
            httpRespBfr.append(s);
            httpRespBfr.append("\n");
        }

        httpResponse = httpRespBfr.toString();

        if (numFields == 0) {
           com.cboe.exceptions.CommunicationException e = new
com.cboe.exceptions.CommunicationException();
           e.details = new com.cboe.exceptions.ExceptionDetails();
           e.details.message = "No CAS Found at IP Address: "+ipAddress+ " Port:
"+tcpPortNumber;
           e.details.dateTime =
DateTimeHelper.dateTimeStructToString(DateTimeHelper.makeDateTimeStruct(new Date()));
           e.details.error = 9000;
           e.details.severity = 1;
           throw e;
        }

        BufferedReader in  = new BufferedReader(
              new InputStreamReader((InputStream)conn.getContent()));

        ior = in.readLine();

      }
      catch(java.io.IOException e) {
           com.cboe.exceptions.CommunicationException cmiexception = new
com.cboe.exceptions.CommunicationException();
           cmiexception.details = new com.cboe.exceptions.ExceptionDetails();
           cmiexception.details.message = e.getMessage();
           cmiexception.details.dateTime =
DateTimeHelper.dateTimeStructToString(DateTimeHelper.makeDateTimeStruct(new Date()));
           cmiexception.details.error = 9000;
           cmiexception.details.severity = 1;
           throw cmiexception;
      }


      return ior;
  }
  /**
   * Return the HTTP Response that was obtained from the CAS Server
   *
   */
  public String getHttpResponse() {
        return httpResponse;
  }

  /**
   * Provide a way for the user to query what IP address was used to access the CAS
   */
  public String getCASIPAddress() {
        return ipAddress;
  }
```

```
   /**
    * Provie a way for the user to query what TCP Port number was used to access the CAS
    */
   public int getTCPPortNumber() {
         return tcpPortNumber;
   }
}
```

Accessing CAS Services

Once the object reference to the UserSessionManager has been returned from the UserAccess::logon() operation, the client application can obtain access to CAS services by invoking operations provided by the UserSessionManager to each CAS service.

NOTE: Not every service is available to all users. For instance the Quote Service is only available to market makers.

NOTE: Since each login is given its own UserSessionManager, the application will receive unique references to each service, as well.

Once you have used the UserSessionManager to obtain references to the services required by your application, the next step is to obtain initial start up information.


:
ClientApplication
:
UserSessionManager
getMarketQuery( )
getOrderEntry( )
getOrderQuery( )
getProductDefinition( )
getProductQuery( )
getQuote( )
getTradingSession( )
getUserPreferenceQuery( )
getTradingSession( )
getUserTradingParameters( )

Summary of UserSessionManager Operations

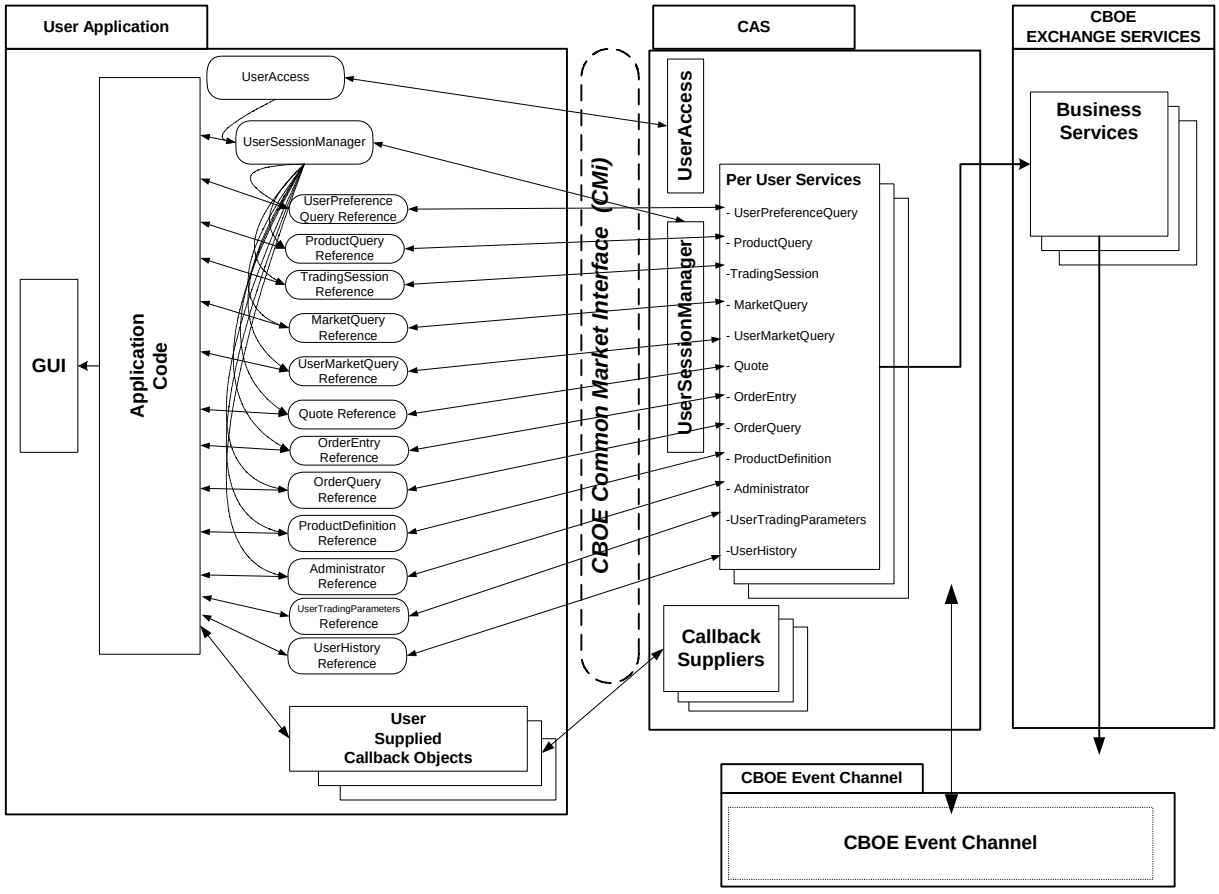
UserSessionManager Operation	Description
authenticate	Used to re-authenticate the client to the CAS after a period of inactivity is detected by the CAS. This method is invoked as a response to an invocation of the UserSessionAdminCallback.acceptAuthenticationNotice() operation. This method is required for all roles (Market Maker, DPM, Broker, Class Display, Firm Display and Firm).
changePassword	This operation is provided to change the password for the user currently logged into the CAS. You must specify the old password and the new password. Developers of interactive applications are strongly encouraged to prompt the user to confirm the new password beforehand and then validate the new password by comparing the new password with the confirmed version prior to invoking the changePassword() operation. This method is required in case the CBOE requires users to change their password at some point in time for security reasons. This method is required for all roles (Market Maker, DPM, Broker, Class Display, Firm Display and Firm).
getAdministrator	Obtain a reference to the CAS administrator object. This method is required for the Market Maker, DPM, Broker and Firm roles. Optional for the Firm Display role. Not allowed for the Class Display role.
getMarketQuery	Obtain the reference to the Market Query Service. This method is optional for Market Maker, DPM, Broker, Firm Display and Firm roles but required for the Class Display role.
getOrderEntry	Obtain the reference to the Order Entry service. This method is required for Broker and Firm roles. Required for Market-Makers and DPMs who intend to submit orders. Not allowed for the Class Display and Firm Display roles.
getOrderQuery	Obtain the reference to the Order Query Service. This method is required for Broker and Firm roles. Required for Market-Makers and DPMs who intend to submit orders. Not allowed for the Class Display and Firm Display roles.
getProductDefinition	Obtain the reference to the Product Definition

UserSessionManager Operation	Description
	Service. This method is implemented but not used in Version 1.0 of CBOEdirect.
getProductQuery	Obtain the reference to the Product Query Service. This method is required for all roles (Market Maker, DPM, Broker, Class Display, Firm Display and Firm).
getQuote	Obtain the reference to the Quote Service. This method is required for Market Makers and DPMs, optional for the Broker Dealer role. Not allowed for Class Display role. Optional for the Firm and Firm Display roles.
getSystemDateTime	Returns the System Date and Time from the CAS. This method is optional for all roles (Market Maker, DPM, Broker, Class Display, Firm Display and Firm).
getTradingSession	Obtain a reference to the Trading Session Service. This method is required for all roles (Market Maker, DPM, Broker, Class Display, Firm Display and Firm).
getUserHistory	Get the reference to the user history object that is used to retrieve trader history. This method is not allowed for the Class Display or Firm Display roles. It is optional for the Market Maker, DPM, Firm and Broker roles.
getUserPreferenceQuery	Obtains the reference to the User Preference Service. This method is not allowed for all roles. For CBOE internal use only.
getUserTradingParameters	Returns the UserTradingParameters interface on the CAS. This method is not allowed for the Class Display or Firm Display roles. It is optional for Market Maker, DPM, Broker and Firm roles.
getValidSessionProfileUser	Gets the valid session profile for the user from the session profile user struct. A user can have different profiles for different sessions.
getValidUser	The CAS returns detailed information about the user (such as clearing firm, Q-account, etc.) currently logged into the CAS. This method is required for Broker and Firm roles. Required for Market-Makers and DPMs who intend to submit orders. Optional for Class Display and Firm Display roles.
getVersion	Obtain the reference to the Version Service. A VersionLable is returned from the CAS. This is crucial to match the version of CMi that the client is using with the version that the CBOE is using. This

UserSessionManager Operation	Description
	method is required for all roles (Market Maker, DPM, Broker, Class Display, Firm Display and Firm).
logout	Logs the current user off of the CAS. This method is required for all roles (Market Maker, DPM, Broker, Class Display, Firm Display and Firm).

The following diagram shows the architecture of a typical application and the sequence of steps in obtaining references to CAS provided services.

Architecture of a Typical CAS Client Application



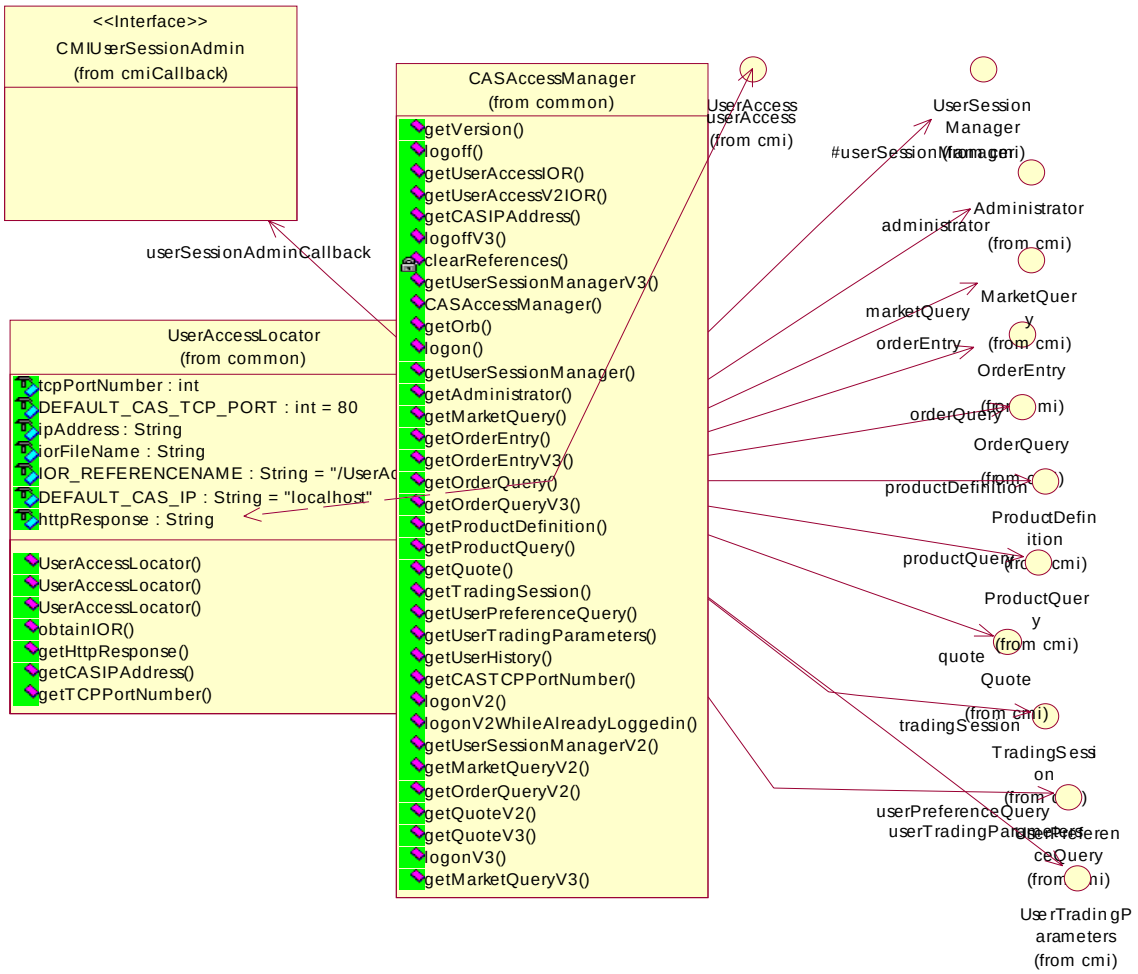
A CAS Access Manager

CBOE has created an example class that manages access to the CAS. It is provided as one possible approach for access to the CAS.

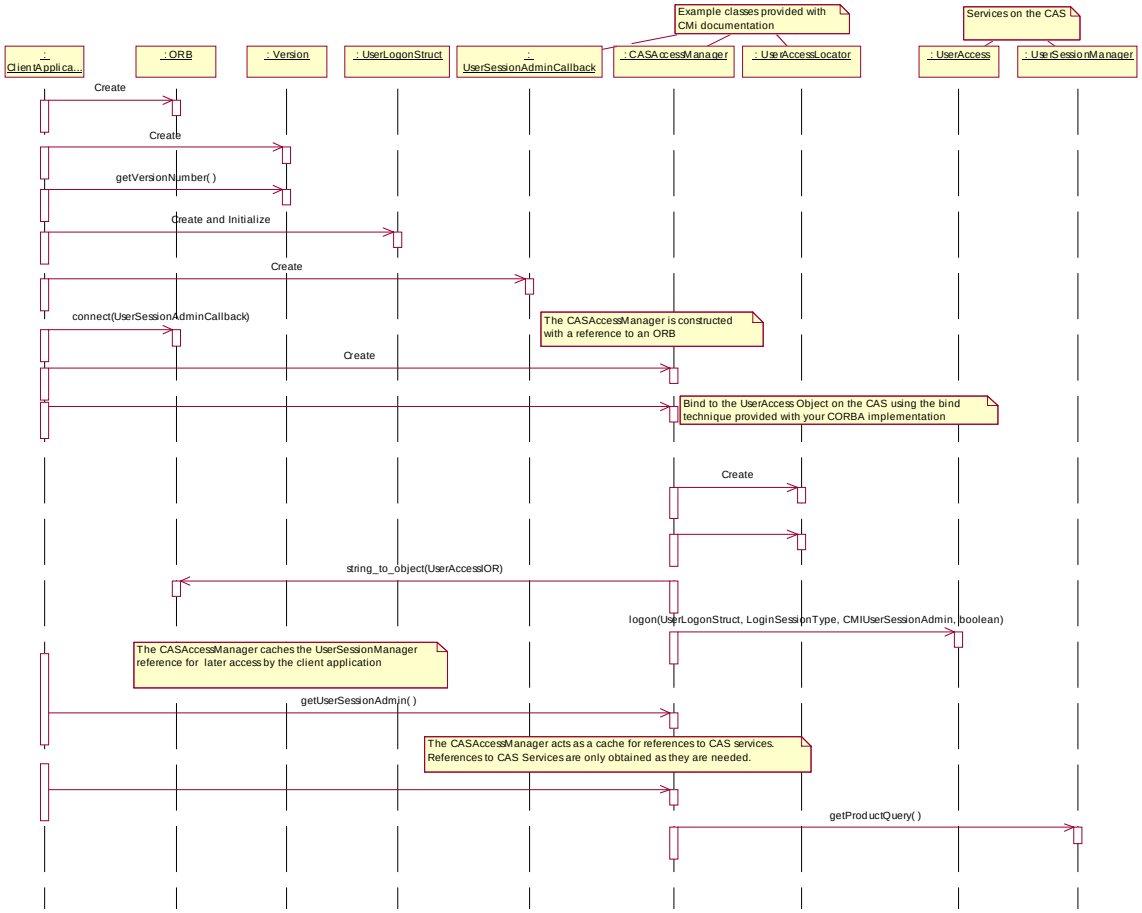
The com.cboe.examples.common.CASAccessManager encapsulates the behavior of obtaining the Interoperable Object Reference (IOR) from the CAS HTTP server, creating an object reference to a UserAccess object, logging on to the UserAccess object. The CASAccessManager then caches the object reference to the UserSessionManager service on the CAS. Methods to access other CAS services are provided. The CASAccessManager obtains the CAS object references on an as needed basis and caches them for subsequent usage by applications.

The Java example programs provided with the CAS all use this CASAccessManager class. An similar implementation is provided in C++ and is used in selected C++ examples.

CASAccessManager Class Diagram



CASAccess Manager Sequence Diagram



Source Code for CASAccessManager

```
package com.cboe.examples.common;


import java.util.Date;

import com.cboe.exceptions.AuthorizationException;
import com.cboe.exceptions.CommunicationException;
import com.cboe.exceptions.ExceptionDetails;
import com.cboe.exceptions.SystemException;
import com.cboe.idl.cmiConstants.LoginSessionTypes;
import com.cboe.idl.cmiV3.UserSessionManagerV3;
import com.cboe.idl.cmiV4.UserSessionManagerV4;

/**
 * Manages access to a CAS, including logging on to the CAS, returning object
 * references to CAS Interfaces, creation and registration of the
 * CMIUserSessionAdmin callback object. <br>
 * <br>
 * Copyright © 1999-2006 by the Chicago Board Options Exchange ("CBOE"), as an
 * unpublished work. The information contained in this software program
 * constitutes confidential and/or trade secret information belonging to CBOE.
 * This software program is made available to CBOE members and member firms to
 * enable them to develop software applications using the CBOE Market Interface
 * (CMi), and its use is subject to the terms and conditions of a Software
 * License Agreement that governs its use. This document is provided "AS IS"
 * with all faults and without warranty of any kind, either express or implied.
 *
 * @version 4.0
 *
 * @author JN
 * @author JZW
 *
 */

public class CASAccessManager {

        protected org.omg.CORBA.ORB orb; // Orb provided to CASAccessManager when
                                                                                // it was
constructed

        String userAccessIOR; // Stringified IOR returned by CAS

        String casVersion; // Version returned by CAS as a string

        static final boolean gmdTextMessaging = true;

        public static String USER_ACCESS_V2_REFERENCENAME = "/UserAccessV2.ior";

        public static String USER_ACCESS_V3_REFERENCENAME = "/UserAccessV3.ior";

        public static String USER_ACCESS_V4_REFERENCENAME = "/UserAccessV4.ior";

        String userAccessV2IOR;

        String userAccessV3IOR;

        String userAccessV4IOR;

        String casIPAddress; // Keep the CAS IP Address and TCP Port Number
                                                        // around

        int casTCPPortNumber; // in case the user wants to access it

        //
        // References to the CAS Interfaces
```

```
        //
        // The CASAccessManager is responsible for obtaining the references
        // to the CAS interfaces on an as needed basis.
        //
        com.cboe.idl.cmi.Administrator administrator;

        com.cboe.idl.cmi.UserAccess userAccess;

        protected com.cboe.idl.cmi.UserSessionManager userSessionManager;

        com.cboe.idl.cmi.OrderEntry orderEntry;

        com.cboe.idl.cmi.OrderQuery orderQuery;

        com.cboe.idl.cmi.TradingSession tradingSession;

        com.cboe.idl.cmi.ProductDefinition productDefinition;

        com.cboe.idl.cmi.MarketQuery marketQuery;

        com.cboe.idl.cmi.Quote quote;

        com.cboe.idl.cmi.ProductQuery productQuery;

        com.cboe.idl.cmi.UserPreferenceQuery userPreferenceQuery;

        com.cboe.idl.cmi.UserTradingParameters userTradingParameters;

        com.cboe.idl.cmi.UserHistory userHistory;

        // services defined in cmiV2
        com.cboe.idl.cmiV2.Quote quoteV2;

        com.cboe.idl.cmiV2.MarketQuery marketQueryV2;

        com.cboe.idl.cmiV2.OrderQuery orderQueryV2;

        com.cboe.idl.cmiV2.UserAccessV2 userAccessV2;

        com.cboe.idl.cmiV2.SessionManagerStructV2 sessionManagerStructV2;

        com.cboe.idl.cmiV2.UserSessionManagerV2 userSessionManagerV2;

        // service defined in cmiV3
        com.cboe.idl.cmiV3.OrderEntry orderEntryV3;

        com.cboe.idl.cmiV3.OrderQuery orderQueryV3;

        com.cboe.idl.cmiV3.Quote quoteV3;

        com.cboe.idl.cmiV3.MarketQuery marketQueryV3;

        com.cboe.idl.cmiV3.UserAccessV3 userAccessV3;

        com.cboe.idl.cmiV3.UserSessionManagerV3 userSessionManagerV3;

        // service defined in cmiV4
        com.cboe.idl.cmiV4.UserAccessV4 userAccessV4;

        com.cboe.idl.cmiV4.UserSessionManagerV4 userSessionManagerV4;

        com.cboe.idl.cmiV4.MarketQuery marketQueryV4;

        /**
         * The CASAccessManager is responsible for providing a CMIUserSessionAdmin
         * interface to the CAS
         */
        com.cboe.idl.cmiCallback.CMIUserSessionAdmin userSessionAdminCallback;

        /**
         * Constructs a CAS Access Manager
         *
         * @param orb
         *            ORB provided by client
```

```
         */
        public CASAccessManager(org.omg.CORBA.ORB orb) {
                this.orb = orb;
        }

        public org.omg.CORBA.ORB getOrb() {
                return this.orb;
        }

        /**
         * Logon to the CAS
         *
         * @param userLogonStruct
         * @param userSessionAdminCallback
         *             Provide a constructed userSession admin callback object. The
         *             client is responsible
         * @param casIPAddress
         *             The IP Address or domain name of the CAS
         * @param casTCPPortNumber
         *             The TCP Port Number of the CAS HTTP Server
         */
        public void logon(
                        com.cboe.idl.cmiUser.UserLogonStruct userLogonStruct,
                        com.cboe.idl.cmiCallback.CMIUserSessionAdmin
userSessionAdminCallback,
                        String casIPAddress, int casTCPPortNumber)
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException,
                        com.cboe.exceptions.AuthenticationException,
                        com.cboe.exceptions.DataValidationException {

                this.userSessionAdminCallback = userSessionAdminCallback;

                // orb.connect(userSessionAdminCallback); // We will connect the object
                // for the user

                this.casIPAddress = casIPAddress;
                this.casTCPPortNumber = casTCPPortNumber;

                UserAccessLocator locator = new UserAccessLocator(casIPAddress,
                                casTCPPortNumber);

                //
                // UserLocator.obtainIOR() will throw a
                // com.cboe.exceptions.CommunicationException
                // if it cannot talk to the CAS. This will be handled by the
                // CASAccessManager client.
                //
                this.userAccessIOR = locator.obtainIOR();

                org.omg.CORBA.Object objRef;

                objRef = orb.string_to_object(userAccessIOR);

                //
                // The UserAccessHelper Class is generated by the CORBA IDL
                // compiler for Java.
                //

                userAccess = com.cboe.idl.cmi.UserAccessHelper.narrow(objRef);

                short sessionType = LoginSessionTypes.PRIMARY;
                //
                // Exceptions thrown by the UserAccess.logon() operation will be handled
                // by the CASAccessManager client.
                //
                userSessionManager = userAccess.logon(userLogonStruct, sessionType,
                                userSessionAdminCallback, gmdTextMessaging);

                return;
        }

        /**
```

```
         * Return the object reference for the UserSessionManager interface on the
         * CAS
         *
         */
        public com.cboe.idl.cmi.UserSessionManager getUserSessionManager()
                        throws com.cboe.exceptions.AuthorizationException {

                if (userSessionManager == null) {
                        com.cboe.exceptions.AuthorizationException authorizationException =
new com.cboe.exceptions.AuthorizationException();
                        short severity = 1;
                        authorizationException.details = new
com.cboe.exceptions.ExceptionDetails(
                                        "CASAccessManager Error: Trying to access
UserSessionManager before being logged on or after being logged off",
                                       new Date().toString(), severity, 9999);
                        throw authorizationException;
                }

                return userSessionManager;
        }

        /**
         * Return the object reference for the Administrator interface on the CAS
         *
         */
        public com.cboe.idl.cmi.Administrator getAdministrator()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (administrator == null) {
                        administrator = getUserSessionManager().getAdministrator();
                }
                return administrator;
        }

        /**
         * Return the object reference for the MarketQuery interface on the CAS
         *
         */
        public com.cboe.idl.cmi.MarketQuery getMarketQuery()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (marketQuery == null) {
                        marketQuery = getUserSessionManager().getMarketQuery();
                }
                return marketQuery;
        }

        /**
         * Return the object reference for the OrderEntry interface on the CAS
         *
         */
        public com.cboe.idl.cmi.OrderEntry getOrderEntry()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (orderEntry == null) {
                        orderEntry = getUserSessionManager().getOrderEntry();
                }
                return orderEntry;
        }

        public com.cboe.idl.cmiV3.OrderEntry getOrderEntryV3()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (orderEntryV3 == null) {
                        orderEntryV3 = getUserSessionManagerV3().getOrderEntryV3();
                }
                return orderEntryV3;
        }
```

```
        /**
         * Return the object reference for the OrderQuery interface on the CAS
         *
         */
        public com.cboe.idl.cmi.OrderQuery getOrderQuery()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (orderQuery == null) {
                        orderQuery = getUserSessionManager().getOrderQuery();
                }
                return orderQuery;
        }

        public com.cboe.idl.cmiV3.OrderQuery getOrderQueryV3()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (orderQueryV3 == null) {
                        orderQueryV3 = getUserSessionManagerV3().getOrderQueryV3();
                }
                return orderQueryV3;
        }

        /**
         * Return the object reference for the ProductDefinition interface on the
         * CAS
         *
         */
        public com.cboe.idl.cmi.ProductDefinition getProductDefinition()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (productDefinition == null) {
                        productDefinition = getUserSessionManager().getProductDefinition();
                }
                return productDefinition;
        }

        /**
         * Return the object reference for the ProductQuery interface on the CAS
         *
         */
        public com.cboe.idl.cmi.ProductQuery getProductQuery()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (productQuery == null) {
                        productQuery = getUserSessionManager().getProductQuery();
                }
                return productQuery;
        }

        /**
         * Return the object reference for the Quote interface on the CAS
         *
         */
        public com.cboe.idl.cmi.Quote getQuote()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (quote == null) {
                        quote = getUserSessionManager().getQuote();
                }
                return quote;
        }

        /**
         * Return the object reference for the TradingSession interface on the CAS
         *
         */
        public com.cboe.idl.cmi.TradingSession getTradingSession()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
```

```
                com.cboe.exceptions.AuthorizationException {
        if (tradingSession == null) {
                tradingSession = getUserSessionManager().getTradingSession();
        }
        return tradingSession;
}

/**
 * Return the object reference for the UserPreferenceQuery interface on the
 * CAS
 *
 */
public com.cboe.idl.cmi.UserPreferenceQuery getUserPreferenceQuery()
                throws com.cboe.exceptions.SystemException,
                com.cboe.exceptions.CommunicationException,
                com.cboe.exceptions.AuthorizationException {
        if (userPreferenceQuery == null) {
                userPreferenceQuery = getUserSessionManager()
                                .getUserPreferenceQuery();
        }
        return userPreferenceQuery;
}

/**
 * Return object reference to UserTradingParameters service on the CAS.
 */
public com.cboe.idl.cmi.UserTradingParameters getUserTradingParameters()
                throws com.cboe.exceptions.SystemException,
                com.cboe.exceptions.CommunicationException,
                com.cboe.exceptions.AuthorizationException {
        if (userTradingParameters == null) {
                userTradingParameters = getUserSessionManager()
                                .getUserTradingParameters();
        }
        return userTradingParameters;
}

/**
 * Return object reference to UserHistory service on the CAS.
 */
public com.cboe.idl.cmi.UserHistory getUserHistory()
                throws com.cboe.exceptions.SystemException,
                com.cboe.exceptions.CommunicationException,
                com.cboe.exceptions.AuthorizationException {
        if (userHistory == null) {
                userHistory = getUserSessionManager().getUserHistory();
        }
        return userHistory;
}

/**
 * Return the object reference for the Version interface on the CAS.
 *
 */
public String getVersion() throws com.cboe.exceptions.SystemException,
                com.cboe.exceptions.CommunicationException,
                com.cboe.exceptions.AuthorizationException {
        if (casVersion == null) {
                casVersion = getUserSessionManager().getVersion();
        }
        return casVersion;
}

/**
 * Logoff the CAS - reinitialize this instance of the CASAccessManager
 *
 */

public void logoff() throws com.cboe.exceptions.SystemException,
                com.cboe.exceptions.CommunicationException,
                com.cboe.exceptions.AuthorizationException {

        getUserSessionManager().logout();
        clearReferences();
```

```
	}

	public void logoffV3() throws com.cboe.exceptions.SystemException,
			com.cboe.exceptions.CommunicationException,
			com.cboe.exceptions.AuthorizationException {

		getUserSessionManagerV3().logout();
		clearReferences();

	}

	public void logoffV4() throws com.cboe.exceptions.SystemException,
			com.cboe.exceptions.CommunicationException,
			com.cboe.exceptions.AuthorizationException {

		getUserSessionManagerV4().logout();
		clearReferences();

	}

	private void clearReferences() {
		//
		// Clear out the CAS interface references
		//

		administrator = null;
		userSessionManager = null;
		userAccess = null;
		orderQuery = null;
		marketQuery = null;
		orderEntry = null;
		tradingSession = null;
		productQuery = null;
		productDefinition = null;
		userPreferenceQuery = null;
		quote = null;
		casVersion = null;
		userAccessIOR = null;
		userTradingParameters = null;
		userHistory = null;
		userSessionAdminCallback = null;
		casIPAddress = null;
		casTCPPortNumber = 0;

		userAccessV2IOR = null;
		quoteV2 = null;
		marketQueryV2 = null;
		orderQueryV2 = null;
		userAccessV2 = null;
		sessionManagerStructV2 = null;
		userSessionManagerV2 = null;

		userAccessV3IOR = null;
		orderEntryV3 = null;
		orderQueryV3 = null;
		quoteV3 = null;
		marketQueryV3 = null;
		userAccessV3 = null;
		userSessionManagerV3 = null;

		userAccessV4 = null;
		userSessionManagerV4 = null;
	}

	/**
	 * Provides the user with the Stringified IOR returned by the
	 * UserAccessLocator object
	 */
	public String getUserAccessIOR() {
		return userAccessIOR;
	}

	/**
```

```
         * Provides the user with the Stringified IOR returned by the
         * UserAccessLocator object
         */
        public String getUserAccessV2IOR() {
                return userAccessV2IOR;
        }

        /**
         * Provides the user with access to the CAS IP Address that was used to
         * access the CAS
         */
        public String getCASIPAddress() {
                return casIPAddress;
        }

        /**
         * Provides the user with access to the CAS TCP Port Number that was used to
         * access the CAS
         */
        public int getCASTCPPortNumber() {
                return casTCPPortNumber;
        }

        /**
         * Logon to the CAS
         *
         * @param userLogonStruct
         * @param userSessionAdminCallback
         *             Provide a constructed userSession admin callback object. The
         *             client is responsible
         * @param casIPAddress
         *             The IP Address or domain name of the CAS
         * @param casTCPPortNumber
         *             The TCP Port Number of the CAS HTTP Server
         */
        public void logonV2(
                        com.cboe.idl.cmiUser.UserLogonStruct userLogonStruct,
                        com.cboe.idl.cmiCallback.CMIUserSessionAdmin
userSessionAdminCallback,
                        String casIPAddress, int casTCPPortNumber)
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException,
                        com.cboe.exceptions.AuthenticationException,
                        com.cboe.exceptions.DataValidationException,
                        com.cboe.exceptions.NotFoundException {

                this.userSessionAdminCallback = userSessionAdminCallback;

                this.casIPAddress = casIPAddress;
                this.casTCPPortNumber = casTCPPortNumber;

                UserAccessLocator locator = new UserAccessLocator(casIPAddress,
                                casTCPPortNumber, USER_ACCESS_V2_REFERENCENAME);

                //
                // UserLocator.obtainIOR() will throw a
                // com.cboe.exceptions.CommunicationException
                // if it cannot talk to the CAS. This will be handled by the
                // CASAccessManager client.
                //
                this.userAccessV2IOR = locator.obtainIOR();

                org.omg.CORBA.Object objRef;

                objRef = orb.string_to_object(userAccessV2IOR);

                //
                // The UserAccessHelper Class is generated by the CORBA IDL
                // compiler for Java.
                //

                userAccessV2 = com.cboe.idl.cmiV2.UserAccessV2Helper.narrow(objRef);
```

```
                short sessionType = LoginSessionTypes.PRIMARY;
                //
                // Exceptions thrown by the UserAccess.logon() operation will be handled
                // by the CASAccessManager client.
                //
                sessionManagerStructV2 = userAccessV2.logon(userLogonStruct,
                                sessionType, userSessionAdminCallback, gmdTextMessaging);
                userSessionManagerV2 = sessionManagerStructV2.sessionManagerV2;
                userSessionManager = sessionManagerStructV2.sessionManager;
                return;
        }

        public void logonV2WhileAlreadyLoggedin(
                        com.cboe.idl.cmi.UserSessionManager userSessionManager)
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException,
                        com.cboe.exceptions.AuthenticationException,
                        com.cboe.exceptions.DataValidationException,
                        com.cboe.exceptions.NotFoundException {

                this.userSessionManager = userSessionManager;

                UserAccessLocator locator = new UserAccessLocator(casIPAddress,
                                casTCPPortNumber, USER_ACCESS_V2_REFERENCENAME);

                //
                // UserLocator.obtainIOR() will throw a
                // com.cboe.exceptions.CommunicationException
                // if it cannot talk to the CAS. This will be handled by the
                // CASAccessManager client.
                //
                this.userAccessV2IOR = locator.obtainIOR();

                org.omg.CORBA.Object objRef;

                objRef = orb.string_to_object(userAccessV2IOR);

                userAccessV2 = com.cboe.idl.cmiV2.UserAccessV2Helper.narrow(objRef);

                userSessionManagerV2 = userAccessV2
                                .getUserSessionManagerV2(this.userSessionManager);

                return;
        }

        /**
         * Return the object reference for the UserSessionManager interface on the
         * CAS
         *
         */
        public com.cboe.idl.cmiV2.UserSessionManagerV2 getUserSessionManagerV2()
                        throws com.cboe.exceptions.AuthorizationException {

                if (userSessionManagerV2 == null) {
                        com.cboe.exceptions.AuthorizationException authorizationException =
new com.cboe.exceptions.AuthorizationException();
                        short severity = 1;
                        authorizationException.details = new
com.cboe.exceptions.ExceptionDetails(
                                        "CASAccessManager Error: Trying to access
UserSessionManagerV2 before being logged on or after being logged off",
                                        new Date().toString(), severity, 9999);
                        throw authorizationException;
                }

                return userSessionManagerV2;
        }

        /**
         * Return the object reference for the MarketQuery interface on the CAS
         *
         */
        public com.cboe.idl.cmiV2.MarketQuery getMarketQueryV2()
```

```
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (marketQueryV2 == null) {
                        marketQueryV2 = getUserSessionManagerV2().getMarketQueryV2();
                }
                return marketQueryV2;
        }

        /**
         * Return the object reference for the OrderQuery interface on the CAS
         *
         */
        public com.cboe.idl.cmiV2.OrderQuery getOrderQueryV2()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (orderQueryV2 == null) {
                        orderQueryV2 = getUserSessionManagerV2().getOrderQueryV2();
                }
                return orderQueryV2;
        }

        /**
         * Return the object reference for the Quote interface on the CAS
         *
         */
        public com.cboe.idl.cmiV2.Quote getQuoteV2()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (quoteV2 == null) {
                        quoteV2 = getUserSessionManagerV2().getQuoteV2();
                }
                return quoteV2;
        }

        public com.cboe.idl.cmiV3.Quote getQuoteV3()
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException {
                if (quoteV3 == null) {
                        quoteV3 = getUserSessionManagerV3().getQuoteV3();
                }
                return quoteV3;
        }

        public void logonV3(
                        com.cboe.idl.cmiUser.UserLogonStruct userLogonStruct,
                        com.cboe.idl.cmiCallback.CMIUserSessionAdmin
userSessionAdminCallback,
                        String casIPAddress, int casTCPPortNumber)
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException,
                        com.cboe.exceptions.AuthenticationException,
                        com.cboe.exceptions.DataValidationException,
                        com.cboe.exceptions.NotFoundException {

                this.userSessionAdminCallback = userSessionAdminCallback;

                this.casIPAddress = casIPAddress;
                this.casTCPPortNumber = casTCPPortNumber;

                UserAccessLocator locator = new UserAccessLocator(casIPAddress,
                                casTCPPortNumber, USER_ACCESS_V3_REFERENCENAME);

                //
                // UserLocator.obtainIOR() will throw a
                // com.cboe.exceptions.CommunicationException
                // if it cannot talk to the CAS. This will be handled by the
                // CASAccessManager client.
                //
                this.userAccessV3IOR = locator.obtainIOR();
```

```
                org.omg.CORBA.Object objRef;

                objRef = orb.string_to_object(userAccessV3IOR);

                //
                // The UserAccessHelper Class is generated by the CORBA IDL
                // compiler for Java.
                //

                userAccessV3 = com.cboe.idl.cmiV3.UserAccessV3Helper.narrow(objRef);

                short sessionType = LoginSessionTypes.PRIMARY;
                //
                // Exceptions thrown by the UserAccess.logon() operation will be handled
                // by the CASAccessManager client.
                //
                userSessionManagerV3 = userAccessV3.logon(userLogonStruct, sessionType,
                                userSessionAdminCallback, gmdTextMessaging);
                userSessionManagerV2 = userSessionManagerV3;
                userSessionManager = userSessionManagerV3;
                return;
        }

        public UserSessionManagerV3 getUserSessionManagerV3()
                        throws AuthorizationException {

                if (userSessionManagerV3 == null) {
                        System.out.println("You are not logon");
                        AuthorizationException authorizationException = new
AuthorizationException();
                        short severity = 1;
                        authorizationException.details = new ExceptionDetails(
                                        "CASAccessManager Error: Trying to access
UserSessionManagerV3 before being logged on or after being logged off",
                                        new Date().toString(), severity, 9999);
                        throw authorizationException;
                }

                return userSessionManagerV3;
        }

        public com.cboe.idl.cmiV3.MarketQuery getMarketQueryV3()
                        throws SystemException, CommunicationException,
                        AuthorizationException {
                if (marketQueryV3 == null) {
                        marketQueryV3 = getUserSessionManagerV3().getMarketQueryV3();
                }
                return marketQueryV3;
        }

        public com.cboe.idl.cmiV4.MarketQuery getMarketQueryV4()
                        throws SystemException, CommunicationException,
                        AuthorizationException {
                if (marketQueryV4 == null) {
                        marketQueryV4 = getUserSessionManagerV4().getMarketQueryV4();
                }
                return marketQueryV4;
        }

        public void logonV4(
                        com.cboe.idl.cmiUser.UserLogonStruct userLogonStruct,
                        com.cboe.idl.cmiCallback.CMIUserSessionAdmin
userSessionAdminCallback,
                        String casIPAddress, int casTCPPortNumber)
                        throws com.cboe.exceptions.SystemException,
                        com.cboe.exceptions.CommunicationException,
                        com.cboe.exceptions.AuthorizationException,
                        com.cboe.exceptions.AuthenticationException,
                        com.cboe.exceptions.DataValidationException,
                        com.cboe.exceptions.NotFoundException {

                this.userSessionAdminCallback = userSessionAdminCallback;
```

```
                this.casIPAddress = casIPAddress;
                this.casTCPPortNumber = casTCPPortNumber;

                UserAccessLocator locator = new UserAccessLocator(casIPAddress,
                                casTCPPortNumber, USER_ACCESS_V4_REFERENCENAME);

                //
                // UserLocator.obtainIOR() will throw a
                // com.cboe.exceptions.CommunicationException
                // if it cannot talk to the CAS. This will be handled by the
                // CASAccessManager client.
                //
                this.userAccessV4IOR = locator.obtainIOR();

                org.omg.CORBA.Object objRef;

                objRef = orb.string_to_object(userAccessV4IOR);

                //
                // The UserAccessHelper Class is generated by the CORBA IDL
                // compiler for Java.
                //

                userAccessV4 = com.cboe.idl.cmiV4.UserAccessV4Helper.narrow(objRef);

                short sessionType = LoginSessionTypes.PRIMARY;
                //
                // Exceptions thrown by the UserAccess.logon() operation will be handled
                // by the CASAccessManager client.
                //
                userSessionManagerV4 = userAccessV4.logon(userLogonStruct, sessionType,
                                userSessionAdminCallback, gmdTextMessaging);
                userSessionManagerV3 = userSessionManagerV4;
                userSessionManagerV2 = userSessionManagerV4;
                userSessionManager = userSessionManagerV4;
                return;
        }

        public UserSessionManagerV4 getUserSessionManagerV4()
                        throws AuthorizationException {

                if (userSessionManagerV4 == null) {
                        AuthorizationException authorizationException = new
AuthorizationException();
                        short severity = 1;
                        authorizationException.details = new ExceptionDetails(
                                        "CASAccessManager Error: Trying to access
UserSessionManagerV4 before being logged on or after being logged off",
                                        new Date().toString(), severity, 9999);
                        throw authorizationException;
                }

                return userSessionManagerV4;
        }

}
```

Support for Multiple Trading Sessions

The CBOE, in conjunction with the rest of the industry, is evolving to support multiple trading sessions. The CMi has been built to support access to multiple trading sessions. This provides your trading applications with a high degree of flexibility in gaining access to all CBOE markets.

Instead of forcing you to login or attach to a trading session, the CMi login results in a connection to CBOE services that are independent of trading session. From your application you will specify which trading sessions you want to participate in. Orders can be submitted to any active trading session. Market data and quoting can occur on all trading sessions in which you are eligible to participate. CBOE supports the following trading sessions.

Trading Session Name	Description	Market Time
W_MAIN	The regular open outcry markets for equity, index, and interest rate options. Use W_MAIN for Linkage.	8:30AM – 3:00PM Central
ONE_MAIN	OneChicago security futures regular trading hours session	8:30AM - 3:00PM Central (3:15 PM closing for futures on Exchange Traded Funds)
CFE_MAIN	CBOE Futures Exchange (CFE) electronic trading	8:30AM – 3:15PM Central
COF_MAIN	CFE Options on Futures (COF) electronic trading	8:30AM – 3:02PM Central
W_STOCK	Stock Trading On CBOEdirect (formerly traded on SEMS)	8:30AM - 3:00PM Central (3:15 PM closing for futures on Exchange Traded Funds).
C2_MAIN	CBOE 2 (C2) a fully electronic options exchange supporting a maker-taker pricing model	8:30AM – 3:00PM Central

Example 1 shows how to access information about what products trade on each trading session.

Order Contingency Types Available in CBOE Trading Sessions

Contingency	Acronym	CBOE (W_MAIN)	OneChicago Securities Futures (ONE_MAIN)	CBOE Futures Exchange (CFE_MAIN)	CFE Options on Futures (COF_MAIN)	Stock Trading On CBOEdirect (W_STOCK)	CBOE 2 (C2_MAIN)
Market Order		•	•	•	•	• (supported only when product is in OPEN or FAST state)	•
Limit Order		•	•	•	•	•	•
Limit Or Better		•					•
All or None	AON	•	•	•	•	•	•
Fill or Kill	FOK	•	•	•	•	•	•
Immediate or Cancel	IOC	•	•	•	•	•	•
Minimum Quantity	MIN*	•					
Not held	NH						•
With Discretion	W/ or WD	•					
Opening	OPG	•					•
Stop (Stop Loss)	STP	•	•	•	•		•
Stop Limit		•	•	•	•		•
Market if Touched	MIT*	•					
Market on Close	MOC	•					•
Limit on Close	CLO*	•					•
Auction Response		•					•
Intermarket Sweep	ISO	•				•	•
Intermarket Sweep Book	ISB	•					•
Bid Peg Cross						•	
Offer Peg Cross						•	
Reserve		•				•	•
Midpoint Cross						•	
Cross						•	
Tied Cross*						•	
Autolink Cross						•	
Autolink Cross Match						•	
Cross Within						•	
Tied Cross Withing						•	
CASH_CROSS						•	
NEX_DAY_CROSS						•	
TWO_DAY_CROSS						•	
Wash Trade Prevention	WTP	•	•	•	•	•	•

*Currently not supported.

Accessing Product Information (Example 1)

It is common for client applications to retrieve and cache product information (Class and Product). The first example (com.cboe.examples.example1.ClientExample1) shows how to access product information. The CMi API and the services available through the API support trading in a multi-session environment. To support multi-session trading, CMi has the concept of a TradingSession. The majority of information available through the CMi is accessed for a particular trading session. This example demonstrates:

1) How to obtain a list of trading sessions and subscribe for subsequent updates to the state of those trading sessions. This will be of particular interest to developers of interactive applications that plan on providing their users with the ability to select participation in one or more trading sessions.

2) How to obtain the list of product types, classes within those product types, and products within those classes that are eligible for trading on a specific trading session. Included in this is the subscription for subsequent updates to classes and products.

3) How to obtain a list of products (types, classes, and products) that are traded at the exchange.

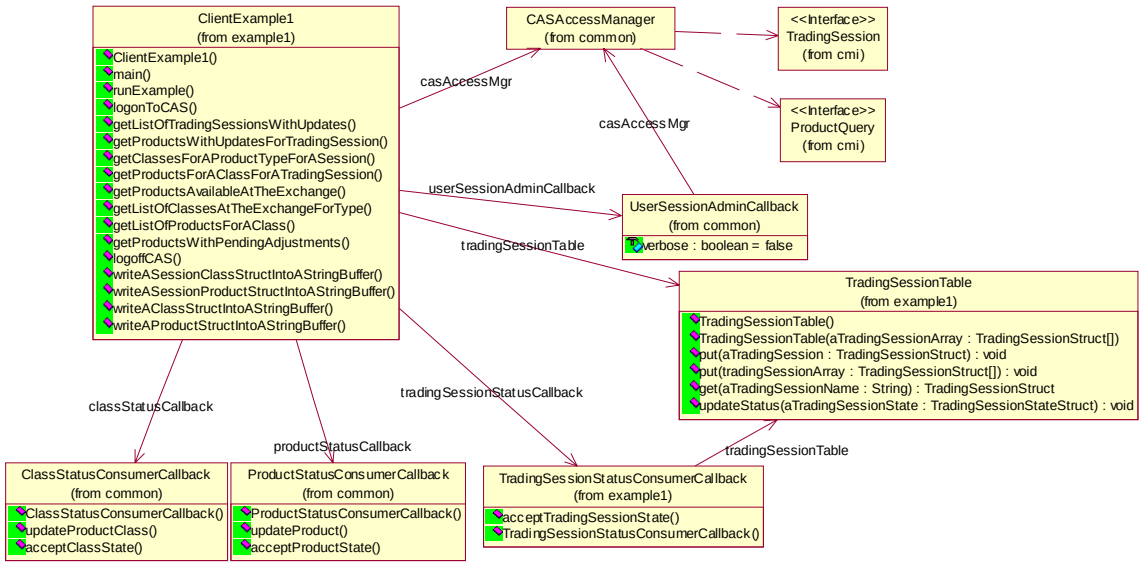
4) How to obtain a list of products for which pending adjustments (most often due to corporate actions, such as stock splits or mergers) exist.

This example is useful in showing how to use the callback objects that you must implement within your application.

The source code for this example is provided with the CMi Software Development Kit.

This example is found in the package com.cboe.examples.example1.

Accessing Product Information Class Diagram

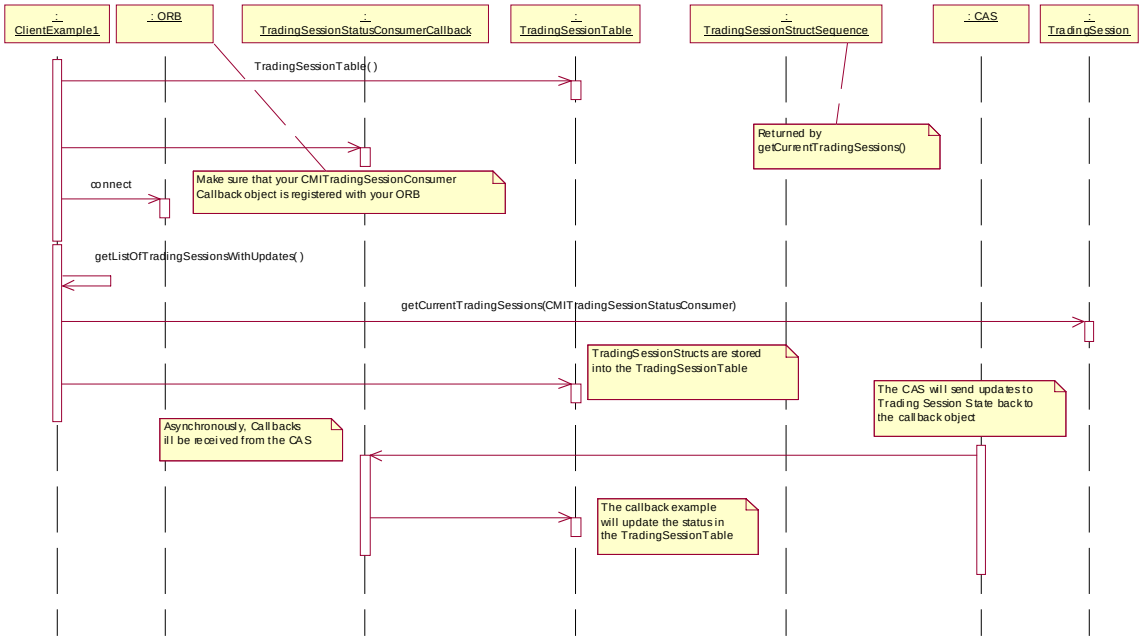


Accessing a List of Trading Sessions

The CMi provides access to information on trading sessions available at the CBOE. Trading sessions are identified by trading session name (cmiSession::TradingSessionName).

The names of the trading sessions have been standardized across the CMi API and the FIX 4.2 interface. The ClientExample1.getListOfTradingSessionWithUpdates() method shows how to obtain a list of trading sessions to be placed in a collection class (com.cboe.examples.example1.TradingSessionTable) that will be updated by a CMICallback::CMITradingSessionStatusConsumer object (com.cboe.examples.example1.TradingSessionStatusConsumerCallback). This sequence diagram shows how to create the callback and access the TradingSession service on the CAS to obtain the list of trading sessions and to subscribe for subsequent update to trading session state.

getListOfTradingSessionsWithUpdates Sequence Diagram

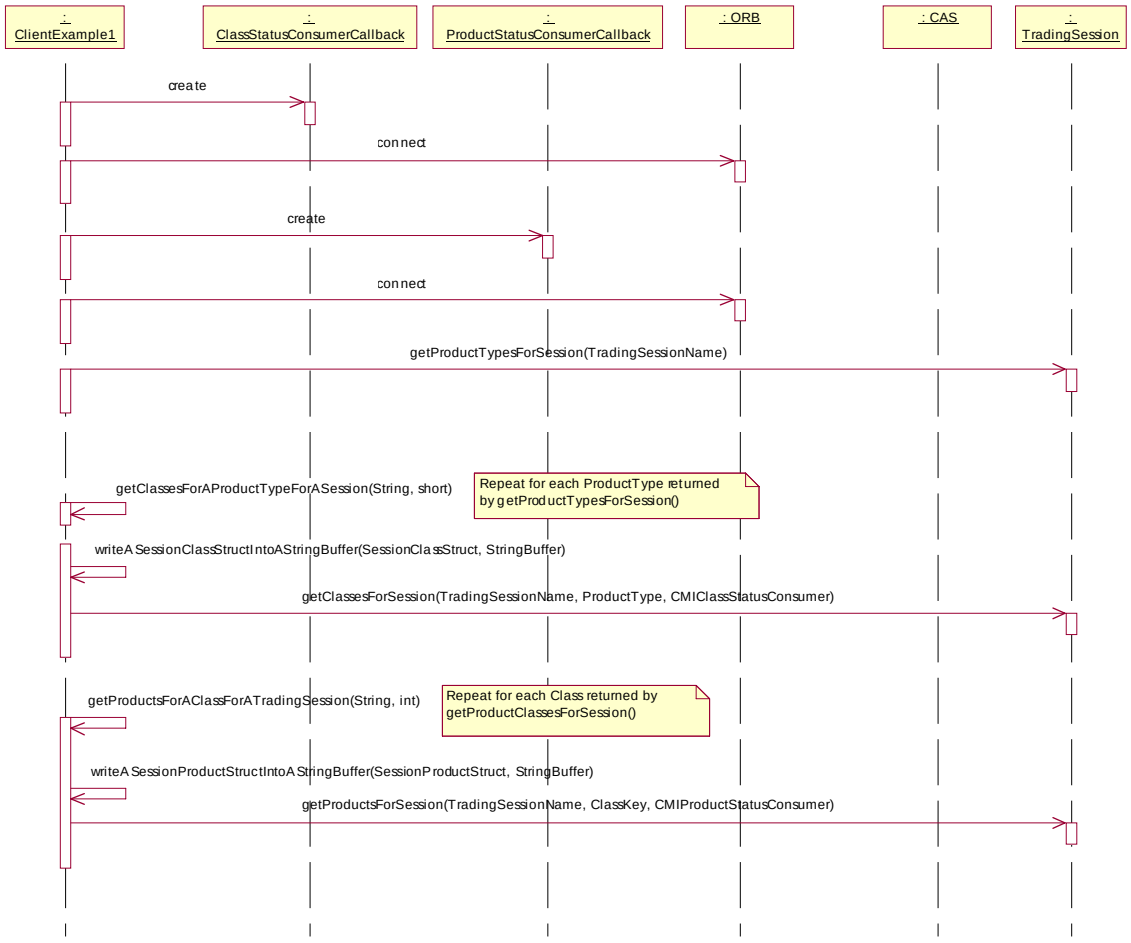


Obtaining Products for a Trading Session with Updates

The getProductsWithUpdatesForTradingSession() method will invoke operations on the cmi::TradingSession interface to first retrieve the product types eligible for trading in the trading session name that was provided as an argument. The method then iterates through the product types, retrieving all classes that are eligible for trading during that trading session for each product type. Finally, the products (series) for each class are retrieved.

Callbacks are registered for subsequent updates to the classes and products. Instead of storing the results in a collection in the example, the class and product information is written to System.out.

getProductsWithUpdatesForTradingSession Sequence Diagram



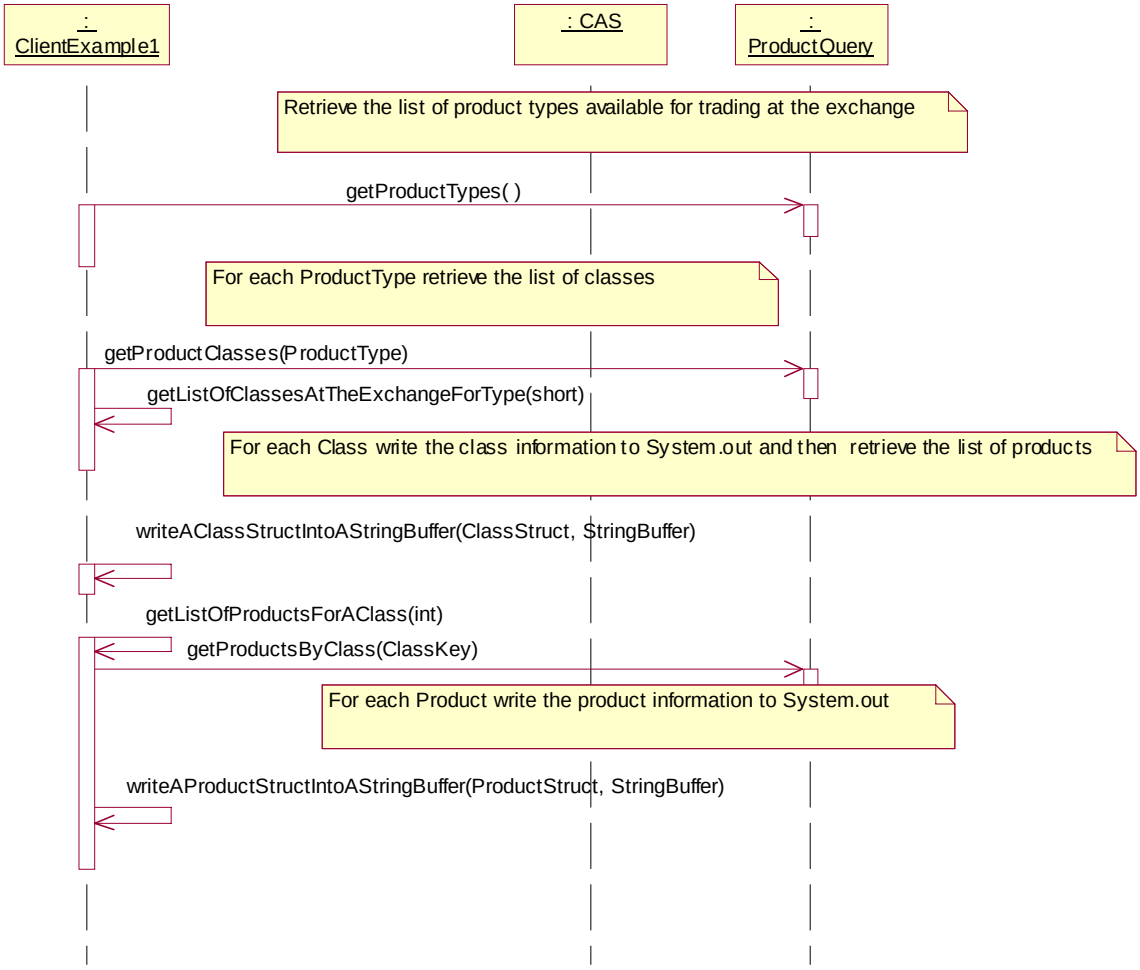
Obtaining Products Traded at the Exchange

The ProductQuery Interface is used to retrieve information on products traded at the exchange. In this example method, getProductsAvailableAtTheExchange(), the list of product types available for trading at an exchange is obtained. For each product type, all the classes of that product type are retrieved. For each of these classes, all of the products (for options these are the series) are retrieved.

As with the other examples, the data is written to System.out.

NOTE: It is important to note that the ProductQuery interface in the CAS (and the Product Service at the Exchange) has no knowledge of trading sessions. The TradingSession interface (and the Session Service of the CBOEdirect electronic trading system) maintain the association between products and trading sessions.

getProductsAvailableAtTheExchange Sequence Diagram



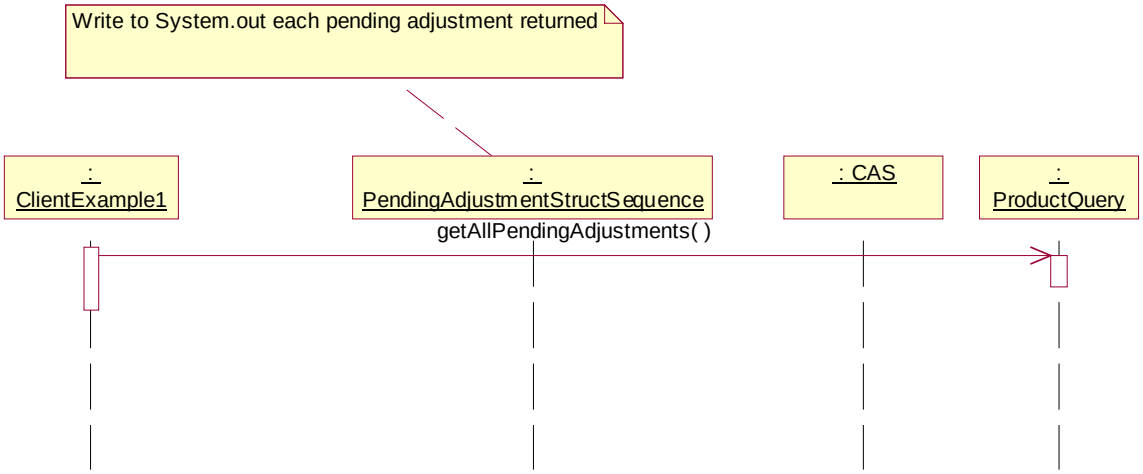
Obtaining Products With Pending Adjustments

One of the advanced features provided through the CMi is the ability to obtain a list of products for which adjustments are pending. Most often these adjustments will be the result of announced corporate actions (such as stock splits or mergers) that will occur at some time in the near future.

This example method, getProductsWithPendingAdjustments(), uses the ProductQuery::getAllPendingAdjustments() operation to retrieve the list of products for which adjustments are pending. These pending adjustments then are written to System.out.

NOTE: This interface can be used to automate maintenance of product information databases.

getProductsWithPendingAdjustments Sequence Diagram



Using CMi to Generate Quotes (Example 2)

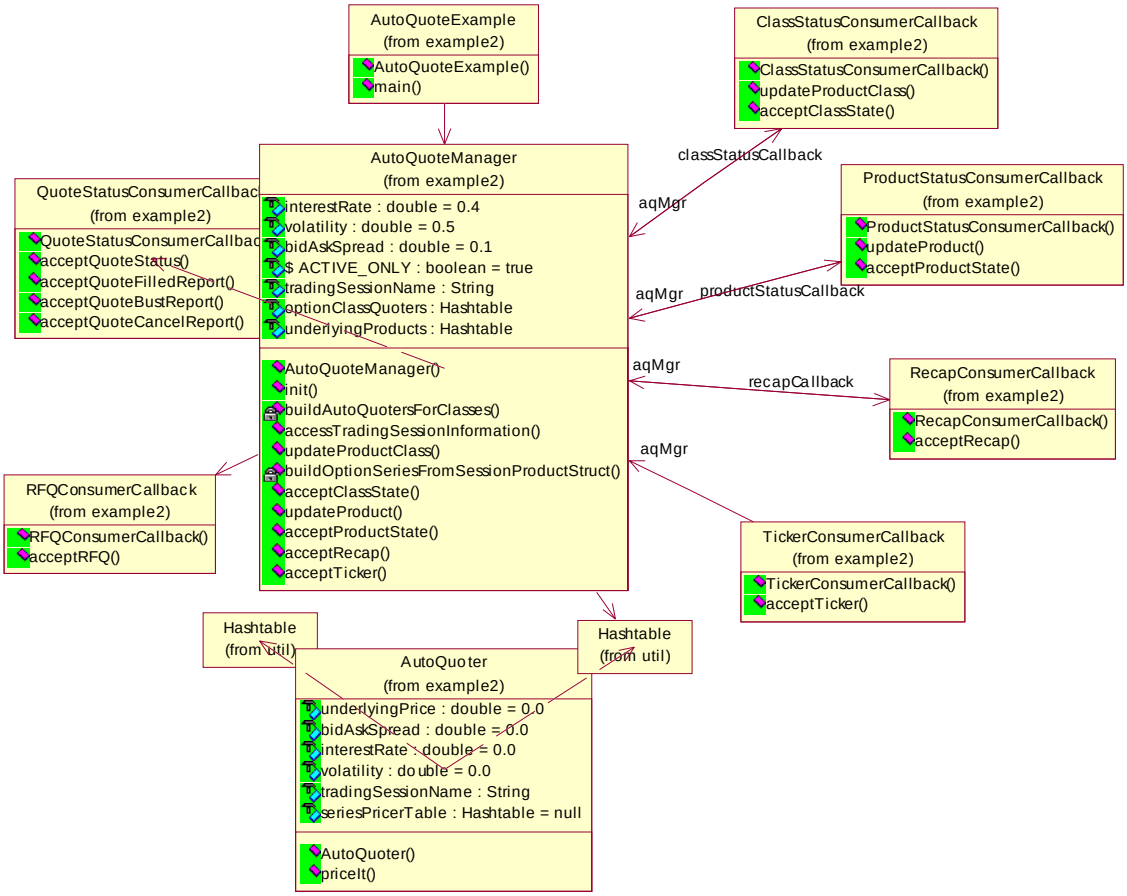
The quoting example is the most elaborate example program. It is a highly simplified "black box" quoting program. The purpose of the program is to provide a basic example that shows what interfaces and callbacks should be employed in developing an automated mass quoting application. Many simplifications are introduced including: only the Black-Scholes model is used; there is only one bid-ask spread, volatility, and interest rate used in the model; the model only takes into account changes in underlying price - not option prices.

The main class is the AutoQuoteManager. The callback objects that receive updates from the CAS have been modified to accept a reference to the AutoQuoteManager on their constructor signature. The callbacks will invoke a method on the AutoQuoteManager each time they receive a call from the CAS. This was the most simplified approach for dispatching. There are other more elaborate approaches available - such as building a separate dispatcher or event handler. The advantage of an event dispatcher approach in your application is that you may have multiple client objects that need to be alerted when an event is received from the CAS. The creation of such a facility was viewed as outside the scope of the examples.

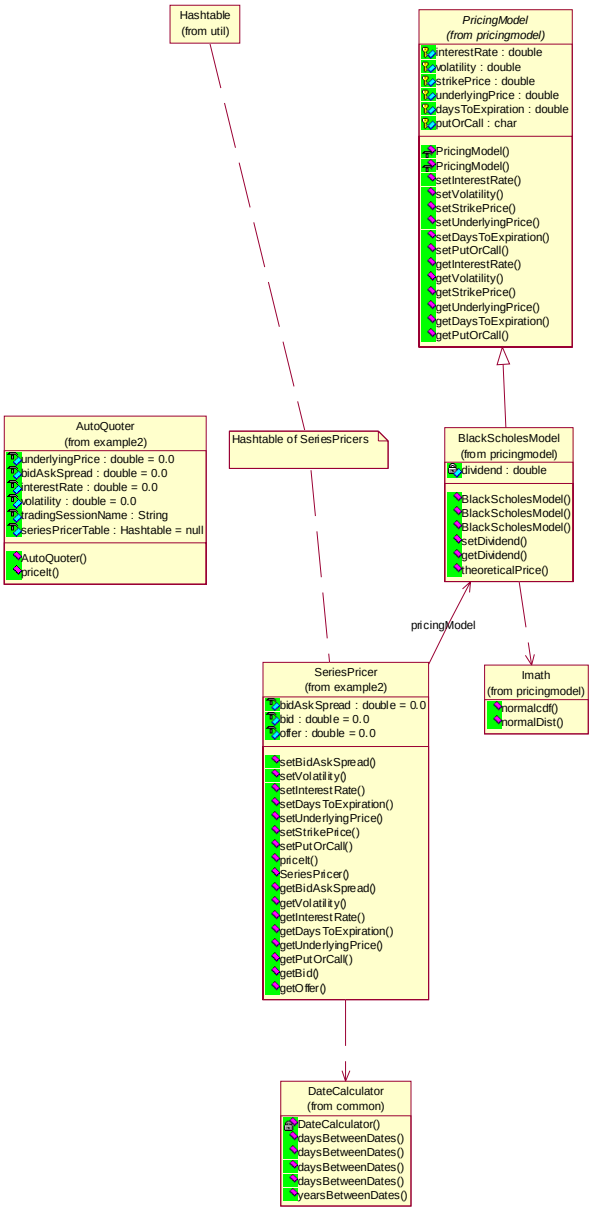
This example is found in the package com.cboe.examples.example2.

CBOE regulates the number of quotes that can be generated at one time. When a quote is not accepted due to the quote limit being exceeded, you will receive a Quote Acknowledgement response that indicates that quote has not been accepted. The most common recovery action is to resubmit the quote. CBOEdirect will reject the entire block of quotes that puts the user over the threshold. Quote rate limits are configurable and subject to change.

AutoQuoteManager Class Diagram



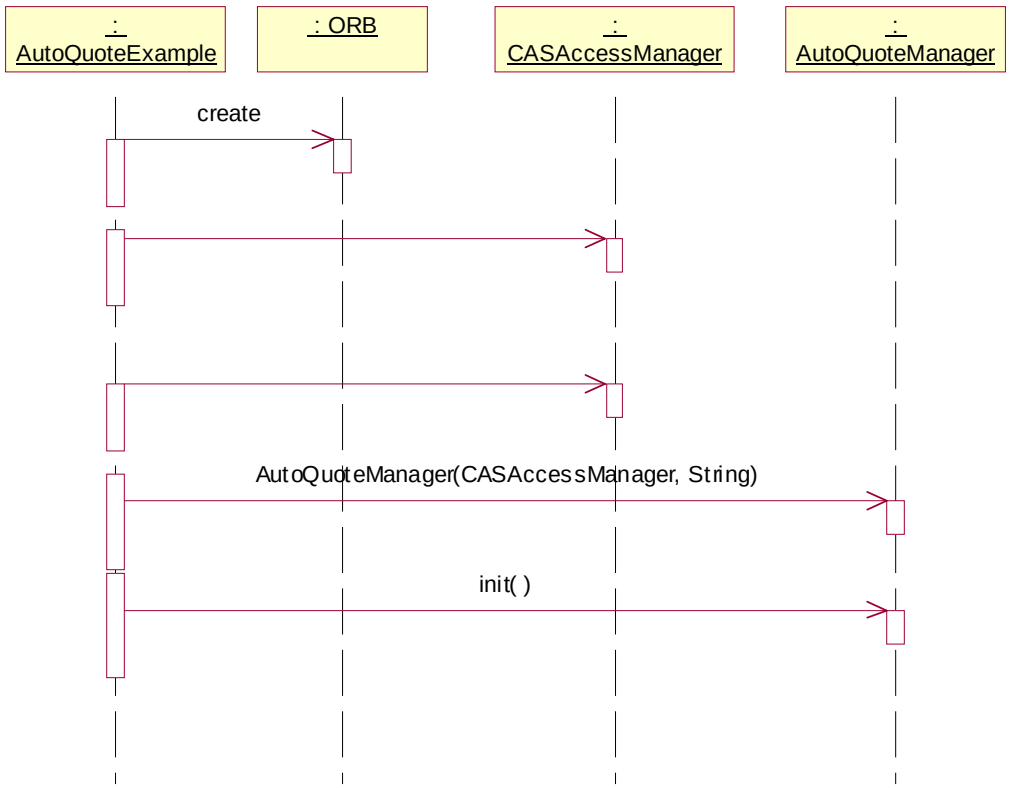
AutoQuoter and SeriesPricer Class Diagram



AutoQuote Example - Main

The AutoQuoteExample main program creates the ORB and a CASAccessManager and logs into the CAS. Once logged in, an AutoQuoteManager is built for a specific trading session and then initialized by calling the AutoQuoteManager.init() method.

main SequenceDiagram



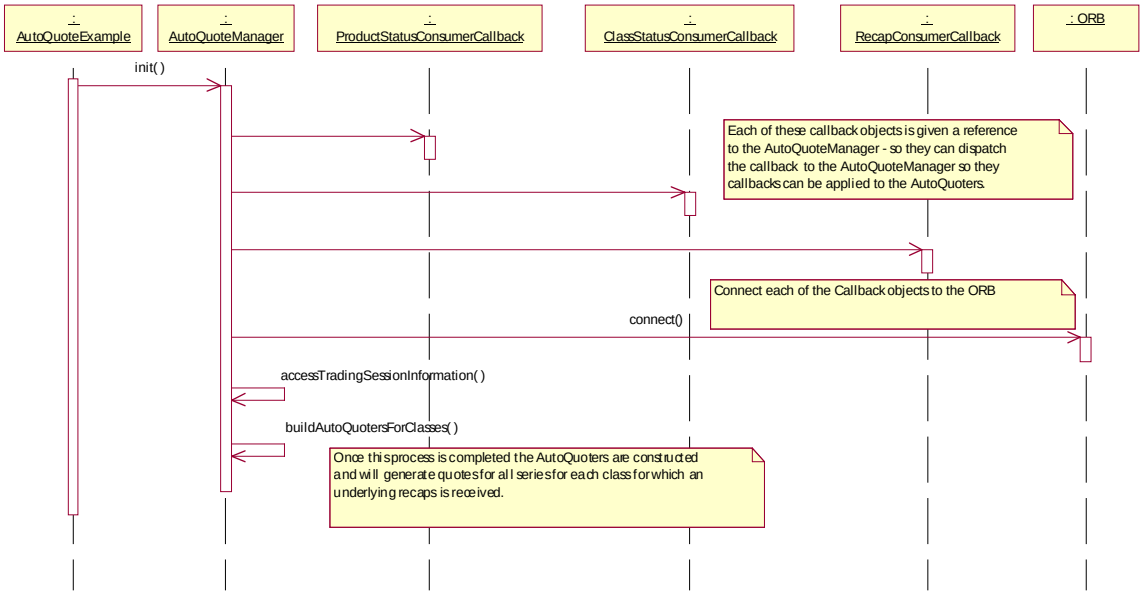
AutoQuote Manager - Initialization

The AutoQuoteManager.init() method sets up callbacks for product status, class status, and recaps. These callbacks are registered with the ORB.

Trading session information is retrieved so that updates to the trading session status can be received from the CAS (Note: this is not used in this example).

Most importantly - AutoQuoters are built for each class in the trading session by the AutoQuoteManager.buildAutoQuotersForClasses() method.

init SequenceDiagram

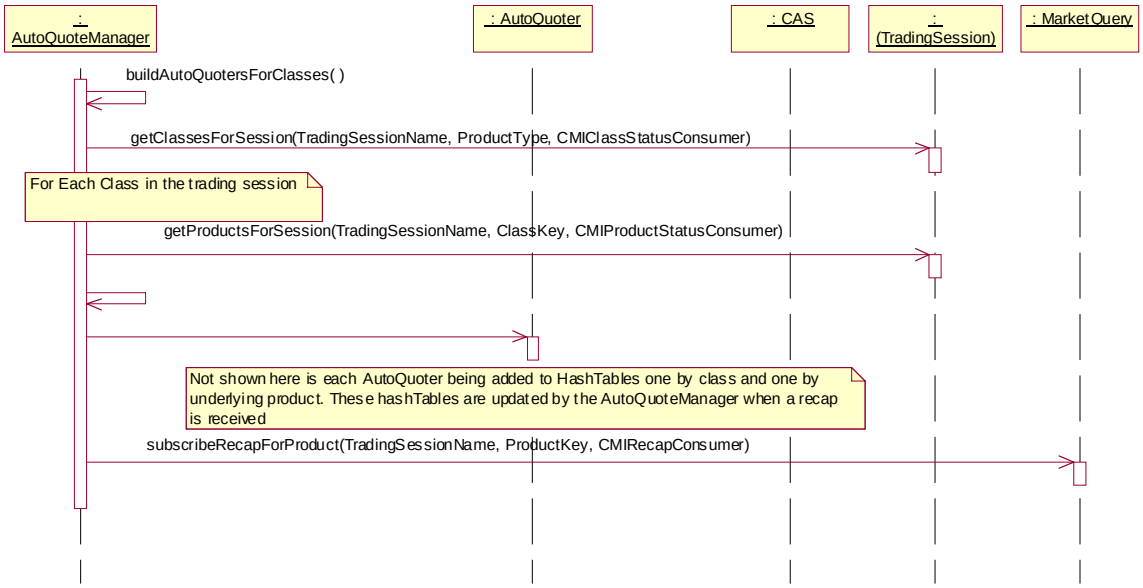


AutoQuote Manager - Build AutoQuoters

The AutoQuotManager.buildAutoQuoterForClasses() method retrieves the list of classes eligible for trading on the trading session for which the AutoQuoteManager was built.

For each class, the list of products (series) is retrieved. The series are built into a table of series pricers. An AutoQuoter is then built to contain these series pricers. Finally, a subscription is entered for the underlying recap of the class.

buildAutoQuotersForClasses SequenceDiagram



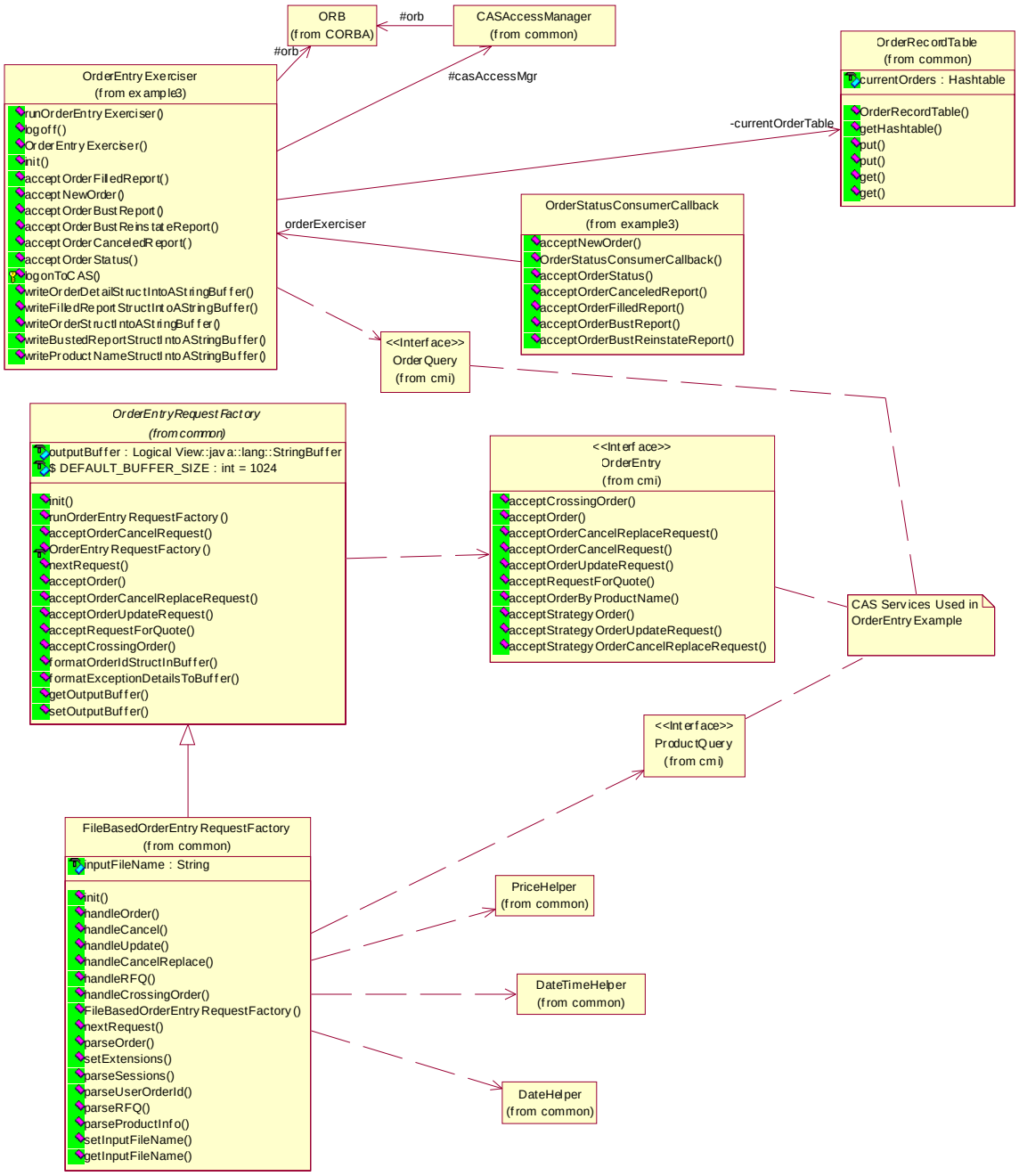
Using CMi for Order Handling (Example 3)

This example is built to provide a generic framework for order routing using the CMi. An abstract class, OrderEntryRequestFactory, encapsulates access to the the CMi OrderEntry interface. In the example a class, FileBasedOrderEntryExerciser, is provided that is derived from the OrderEntryRequestFactory that reads order routing requests from a comma delimited file and submits those requests to the OrderEntry interface,.

The main OrderEntryExerciser class sets up a CMIOrderStatusConsumer callback object to receive order status reports. The example callback has been implemented to forward all incoming events back to the OrderEntryExerciser. An order table is maintained within the OrderEntryExerciser that is updated when activity reports (order acknowledgement, fills, cancels) are issued against the requests (OrderRecordTable).

This example is found in the package com.cboe.examples.example3.

OrderEntryExerciser Class Diagram



Initializing the OrderEntryExerciser

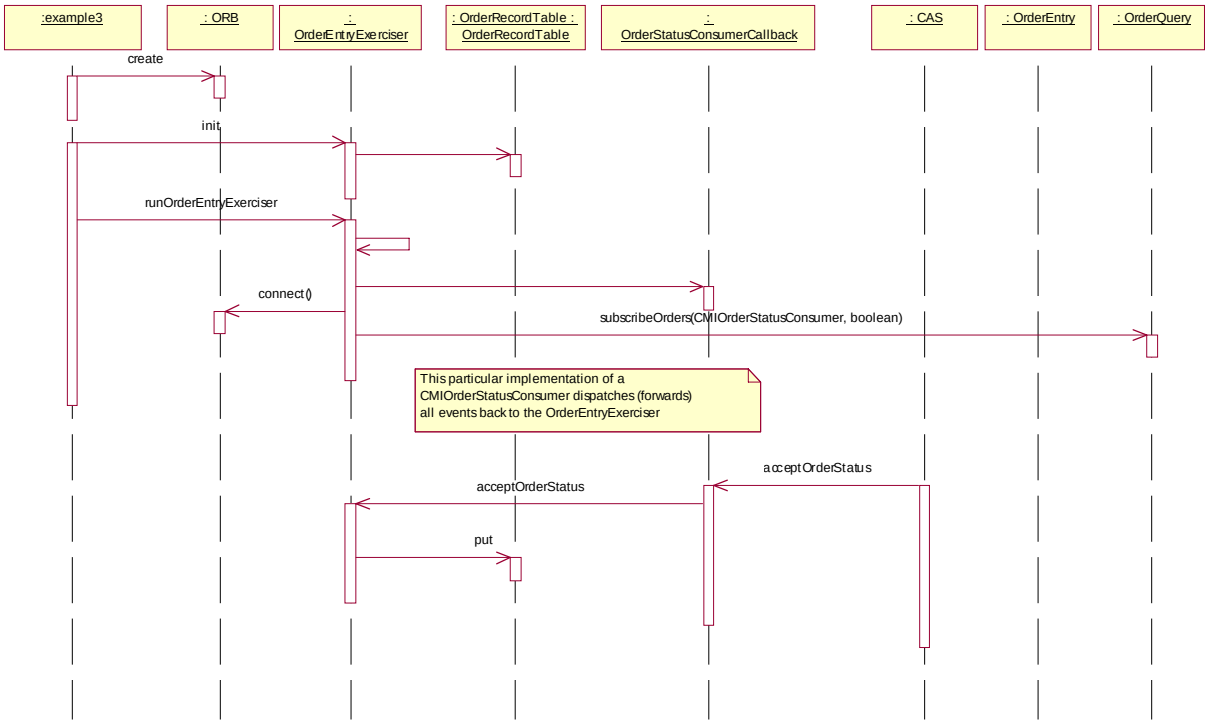
The following sequence diagram shows how the OrderEntryExample creates and invokes the OrderEntryExerciser.

The OrderEntryExerciser creates a CASAccessManager and logs on to the CAS. The OrderEntryExerciser then creates the OrderStatusConsumerCallback object and connects it to the ORB.

The OrderEntryExerciser then subscribes for order status updates by invoking the OrderQuery::subscribeOrders() operation on the CAS. This first invocation of OrderQuery.subscribeOrders() results in all orders for this user being returned by the CAS by the OrderStatusConsumerCallback.acceptOrderStatus() operation.

All order status callbacks are forwarded from the OrderStatusConsumerCallback object to the OrderEntryExerciser.

OrderEntryExerciser-AcceptOrder Sequence Diagram

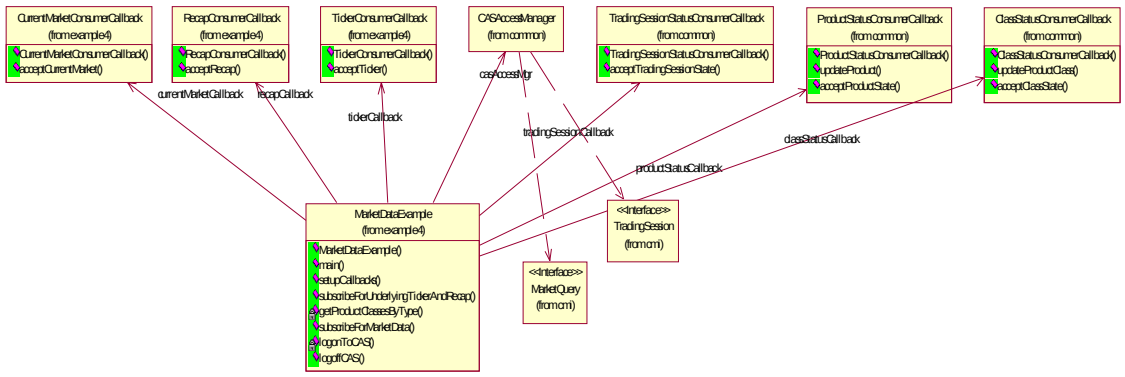


Accessing Market Data using CMi (Example 4)

This example shows basic market data available via the CMi. The first method shows how to subscribe for underlying ticker and recap information. The second method shows how to subscribe for current market and national best bid and offer for products traded during a trading session.

This example is found in the package com.cboe.examples.example4.

MarketDataExample Class Diagram



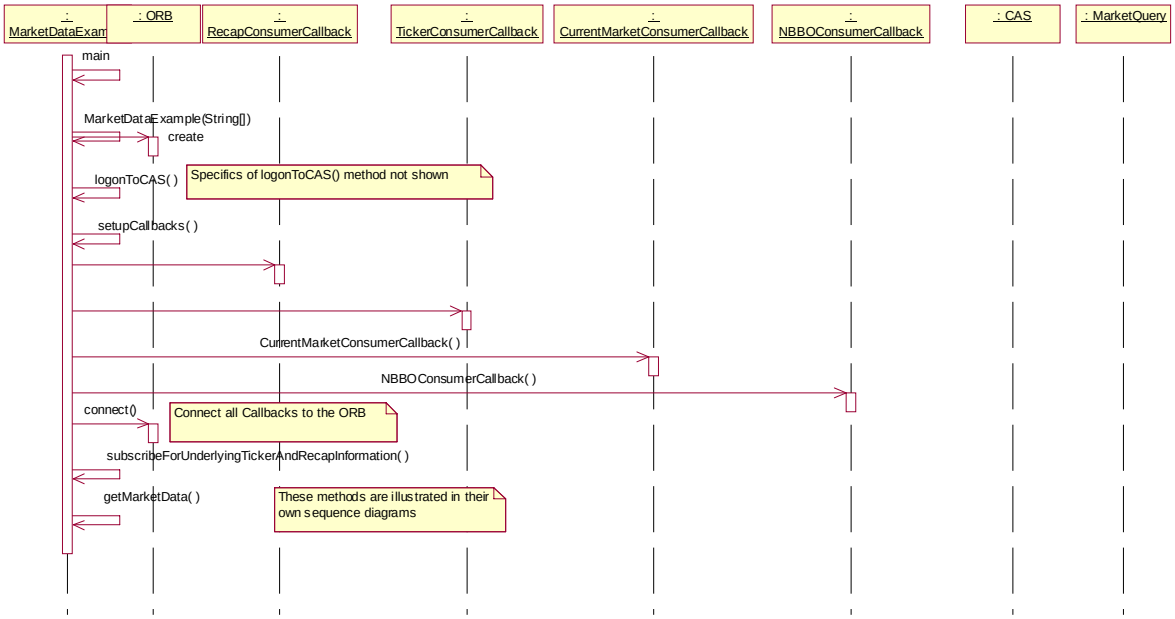
Market Data Example - Main

The MarketDataExample main constructs a MarketDataExample object. The ORB is created in the constructor in this example. The logonToCAS() method creates the CMIUserSessionAdminConsumer callback and then uses the CASAccessManager example to logon to the CAS.

The setupCallbacks() method creates the callback objects and connects them to the ORB.

WIth the callbacks registered, the subscribeForUnderlyingTickerAndRecapInformation() and the getMarketData() methods are called to subscribe for market data from the CAS (These methods are illustrated with their own sequence diagrams).

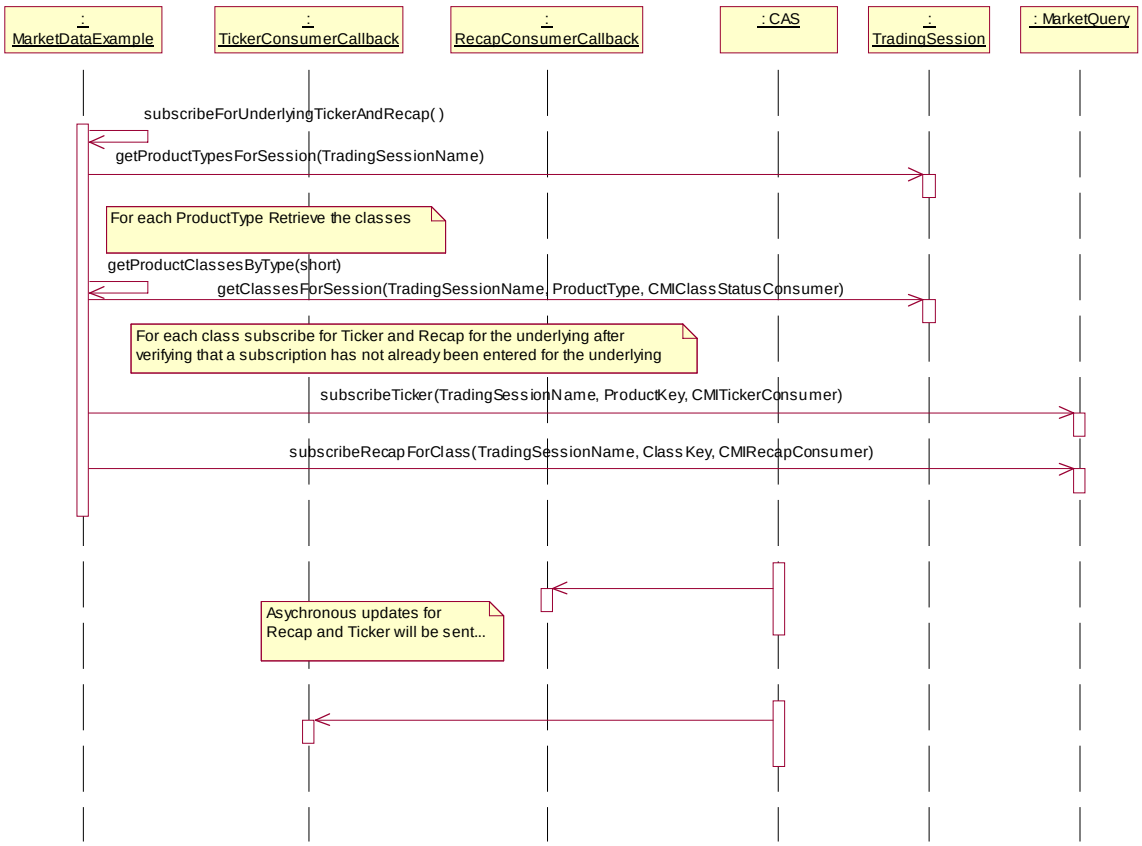
main SequenceDiagram



Market Data Example - Subscribe for Underlying

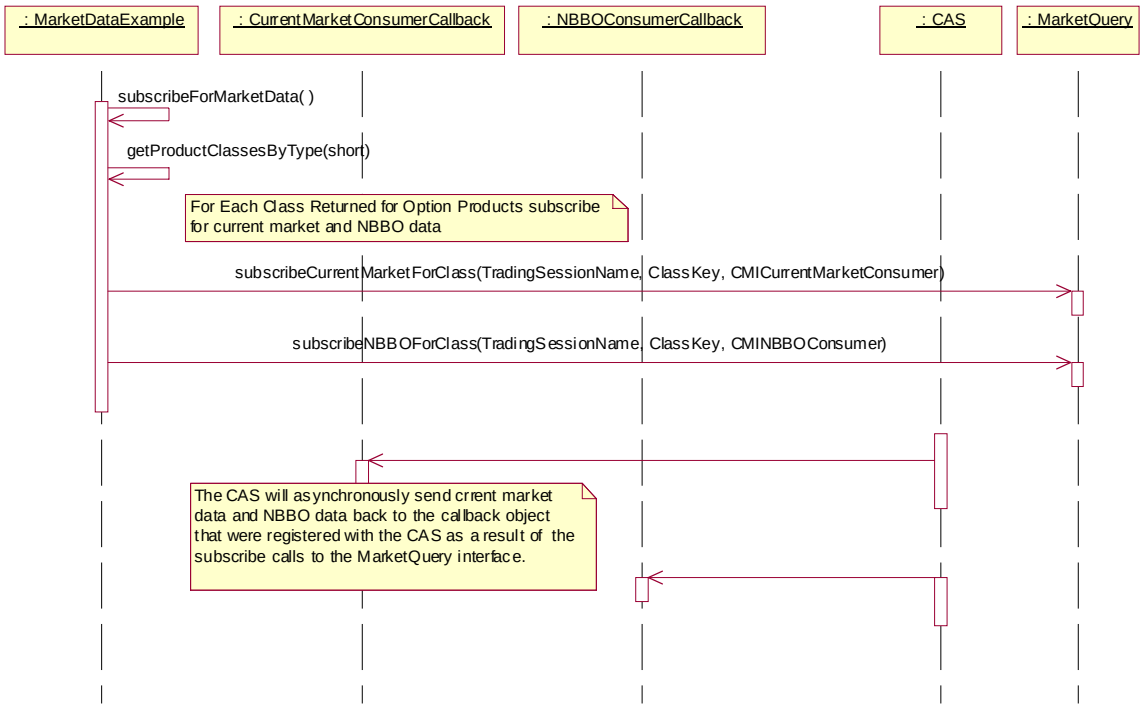
The subscribeForUnderlyingAndRecap() method retrieves the list of product types available for trading for the trading session name coded in the application. For each product type returned, the list of classes is retrieved. For each class, subscribe to the recap and ticker for the underlying product of the class, making sure that a subscription was not previously entered for the underlying product.

subscribeForUnderlying SequenceDiagram



Market Data Example - Get Market Data

subscribeForMarketData SequenceDiagram



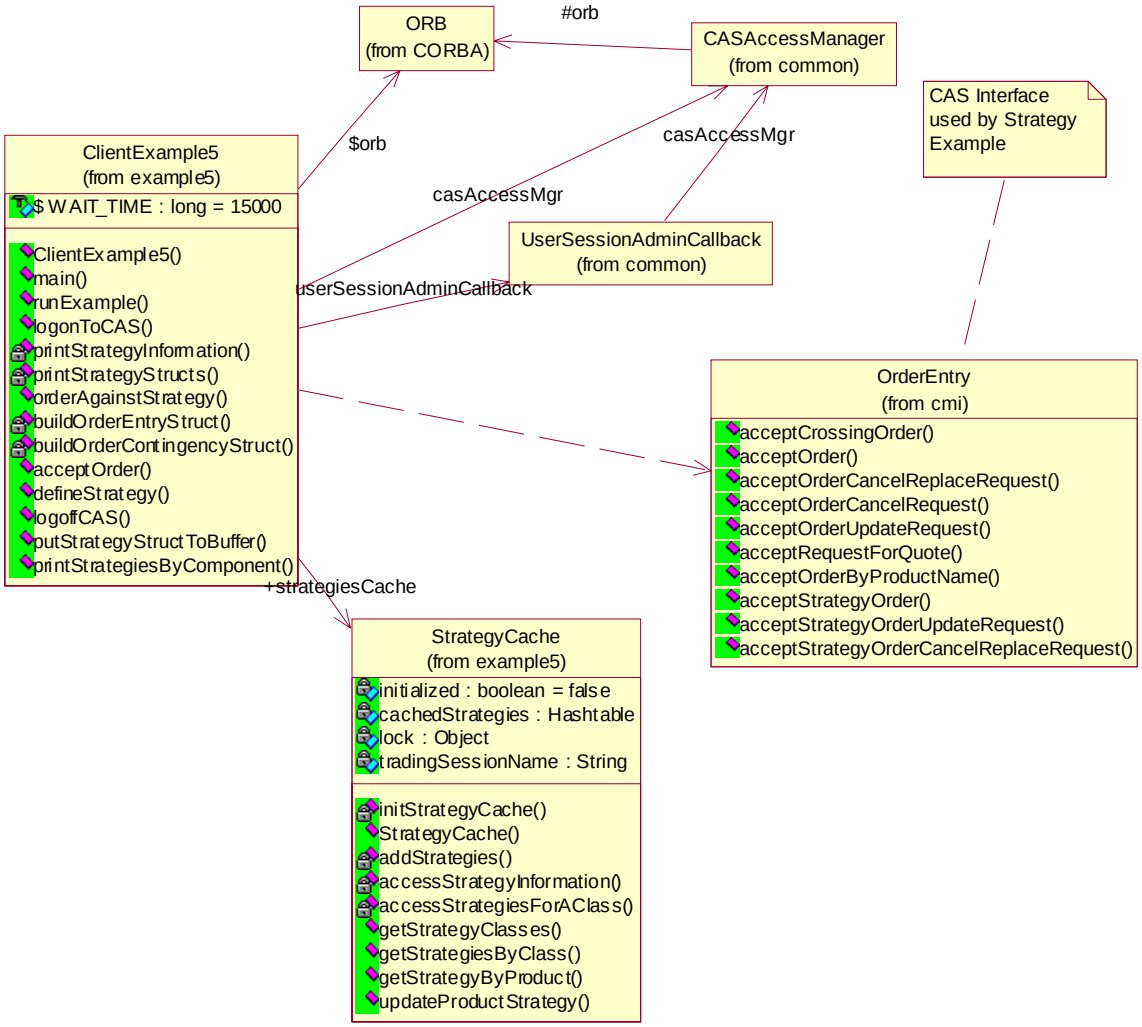
Complex Orders: Defining and Trading Strategy Products (Example 5)

The program shows how to retrieve a list of strategy classes that are available for trading in the current session. Then lists all defined trading strategies for those classes.

The program then defines a strategy using the ProductDefinition.acceptStrategy() operation. The example concludes by submitting an order for a strategy.

This example can be found in the package com.cboe.examples.example5.

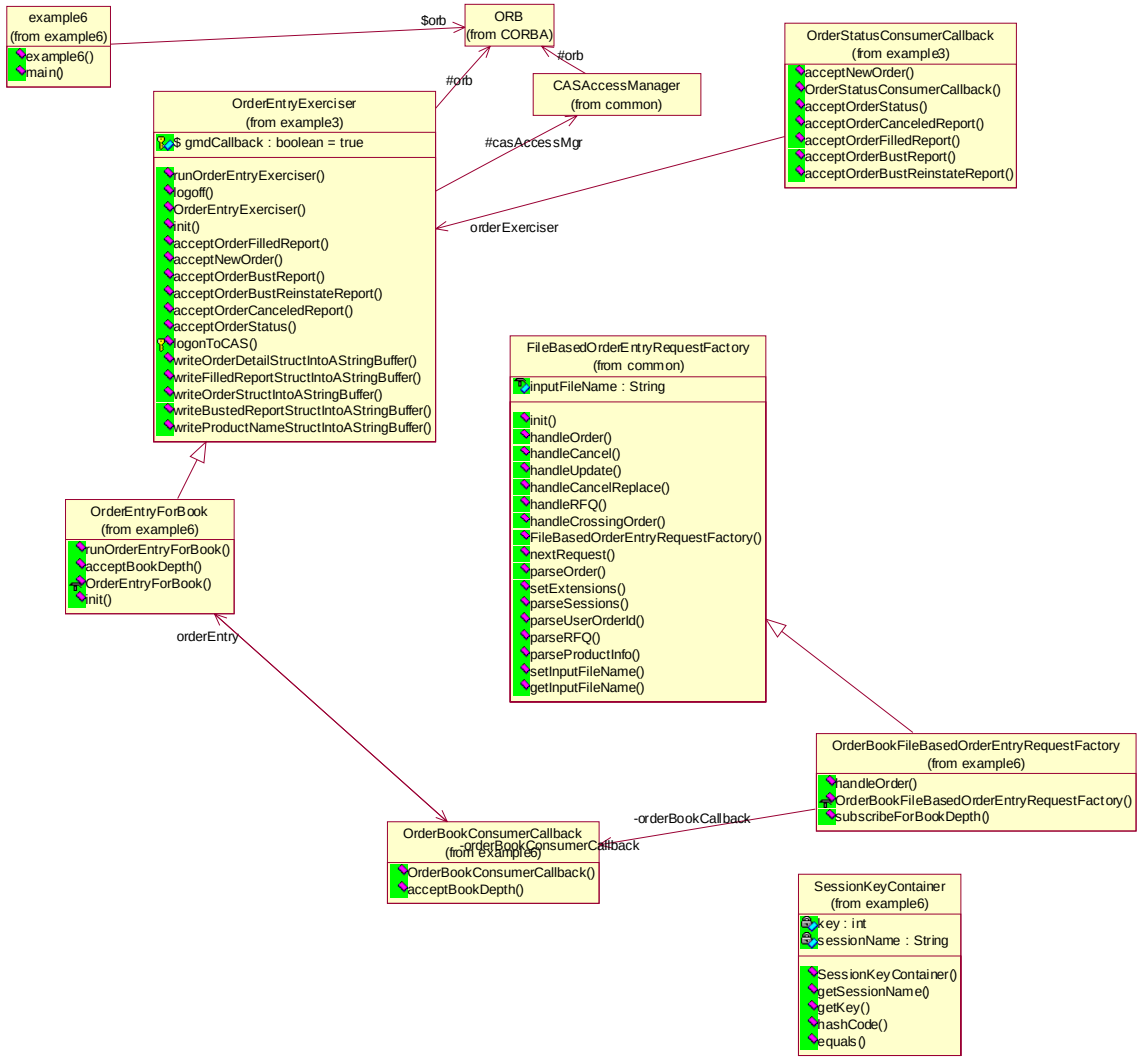
StrategyExample Class Diagram



Dynamic Book Depth Updates (Example 6)

Example 6 builds on the order entry example (Example 3). For each order read from the file of orders book depth is subscribed for the product specified on the order. After the subscription is made the order is submitted. The change to the order book that results from the order is reported back to the OrderBookDepthConsumer object.

BookDepthExample Class Diagram



IPP CAS Access (Example 7)

This program shows how to access IPP CAS services while the client is already logged into a CAS and thus has an object reference to the UserSessionManager service on the CAS. In this example:

(1) the client obtains the Interoperable Object Reference (IOR) from the CAS HTTP server,

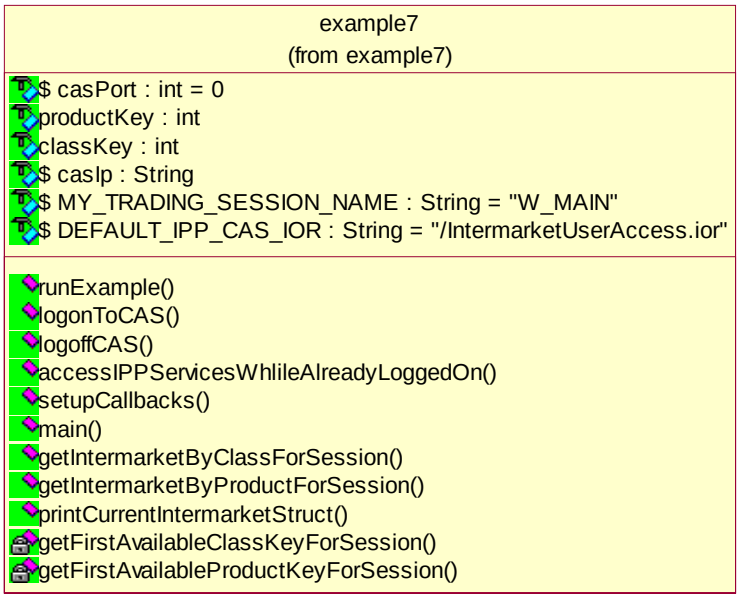
(2) creates an object reference to an IntermarketUserAccess object,

(3) provides the UserSessionManager object reference to IntermarketUserAccess to retrieve a object reference IntermarketUserSessionManager and

(4) uses the IntermarketUserSessionManager object to access IPP services on IPP CAS.

There are two IPP CAS services: IntermarketQuery service and NBBOAgent service. The example below demonstrates the use of IntermarketQuery. The use of the NBBOAgent service is shown in example 8.

IntermarketUserAccess Class Diagram



NBBO Agent Registration and Held Order Handling (Example 8)

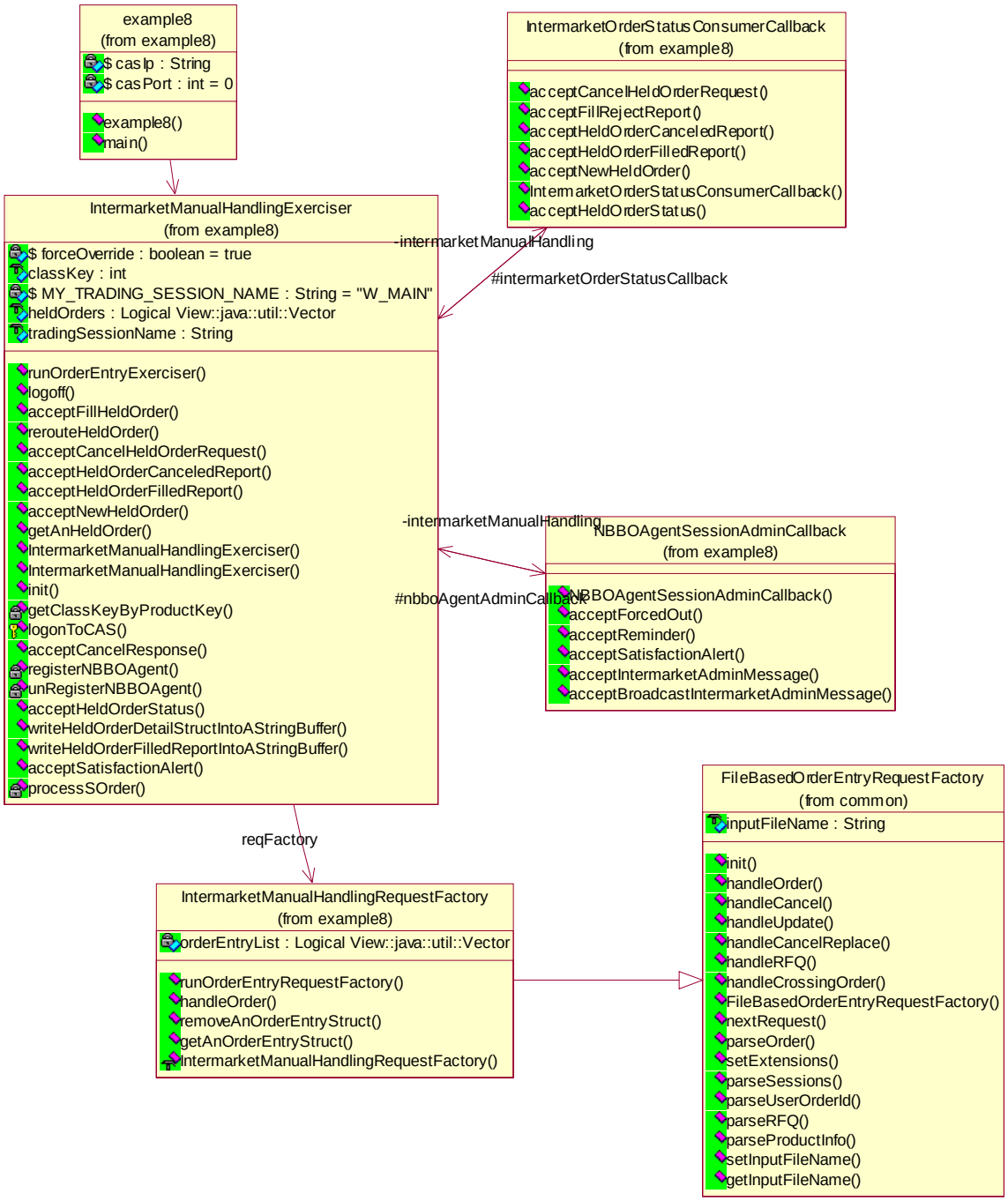
This example demonstrates the receiving of a held order status message and handling held orders using the cmiIntermarket interface. The client first must register to become a NBBOAgent in order to handle held orders and receive held order status messages. In this example the class, IntermarketManualHandlingRequestFactory, is provided to build the order entry struct collection from a text file. The order entry struct collection will then be used in the example.

The main IntermarketManualHandlingExerciser class sets up two callbacks: (1) CMIIntermarketOrderStatusConsumer for held order status reports and (2) CMINBBOAgentSessionAdmin for NBBO agent admin status.

The example first builds an order entry collection from a text file (which will also be used for filling the held order) then registers the NBBOAgent through the cmiIntermarket API. After registration, the client reroutes the first held order it received and fills the second held order it receives. When these two operations are complete, the example user can send a held order cancel request from the simulator. Upon receiving this cancel request, the example will send back a cancel response.

When the NBBO agent receives the S order, the agent will submit the acceptSatisfactionOrderInCrowdFill method. Upon receiving the Satisfaction alert, the NBBO agent will submit the S order with the OrderEntry.acceptOrder method and then call the acceptCustomerOrderSatisfy method.

HeldOrderEntryExample Class Diagram



V2 CAS User Access (Example 9)

This program shows how to get the object reference of the UserAccessV2 and logon to the V2 CAS. In this example:

(1) the client obtains the Interoperable Object Reference (UserAccessV2.IOR) from the CAS HTTP server

(2) creates an object reference to the UserAccessV2

(3) logon to the V2 CAS to retrieve the object references of the SessionManagerStructV2 , MarketQueryV2, QuoteV2 and OrderQueryV2

(4) logoff

(5) logon to V1 CAS and then retreive the object reference of the UserSessionManagerV2

(6) retreive MarketQueryV2, QuoteV2 and OrderQueryV2 references

V2CASUserAccess Class Diagram

Stock Example for the NBBO Agent (Example 10)

The examples demonstrates the Equity specific functionality of CBOEdirect. This functionality is only for NBBOAgent(DPM) and pertains to the Opening procedures and to integrates ITS functionality. The EquityMethodExerciser enumerates each of the new methods available to the NBBO Agent. Refer to the API documentation for detailed explanations of each method.

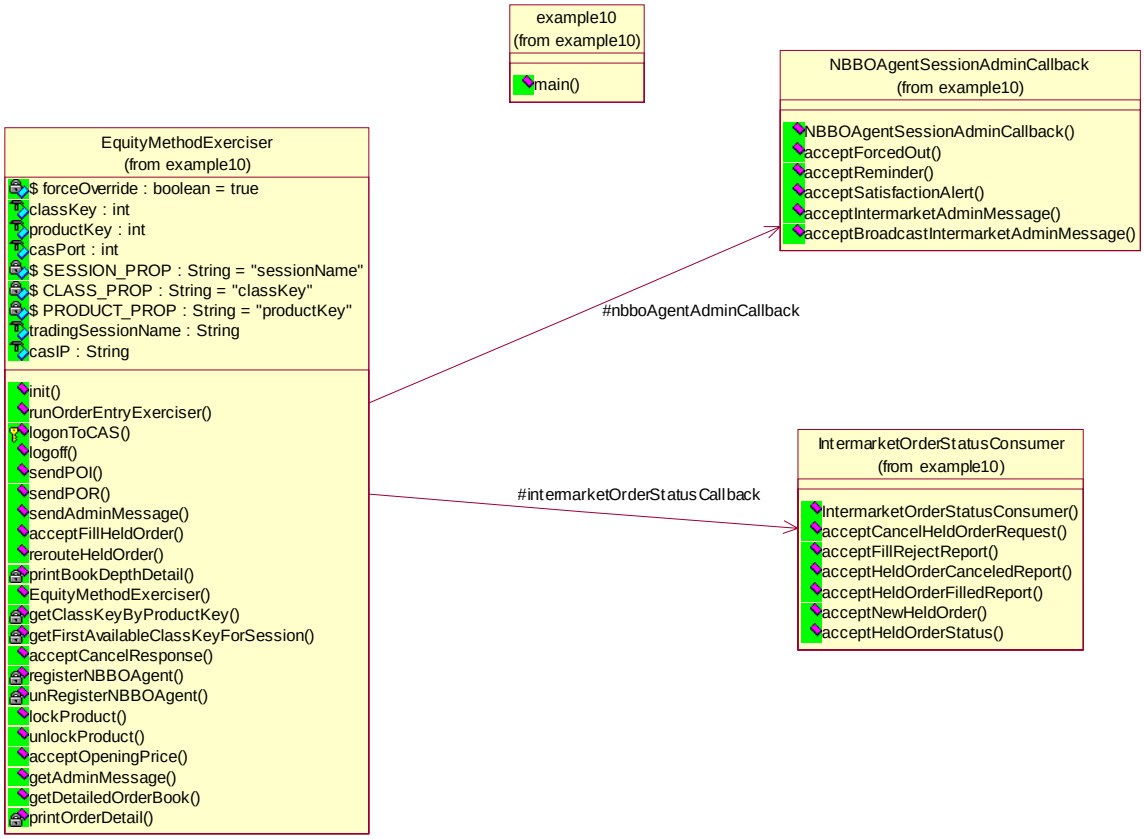
ITS Functionality

1. Send Pre-Opening Indication to other markets.
2. Send Pre-Opening Response to other markets.
3. Send ITS Administration message(complaints, etc) to other markets.
4. Receive ITS Administration message(complaints, POI, POR) from other markets.

Equity Opening Method - CBOE is not primary in issue

1. Get the detailed order book before the open.
2. Lock the order book(Opening Rotation)
3. Unlock the order book(Opening Rotation)
4. Generate the opening trade price for the issue.

StockExamples Class Diagram


example10
(from example10)
main()
NBBOAgentSessionAdminCallback
(from example10)
NBBOAgentSessionAdminCallback()
acceptForcedOut()
acceptReminder()
acceptSatisfactionAlert()
acceptIntermarketAdminMessage()
acceptBroadcastIntermarketAdminMessage()
EquityMethodExerciser
(from example10)
$ forceOverride : boolean = true
classKey : int
productKey : int
casPort : int
$ SESSION_PROP : String = "sessionName"
$ CLASS_PROP : String = "classKey"
$ PRODUCT_PROP : String = "productKey"
tradingSessionName : String
casIP : String
init()
runOrderEntryExerciser()
logonToCAS()
logoff()
sendPOI()
sendPOR()
sendAdminMessage()
acceptFillHeldOrder()
rerouteHeldOrder()
printBookDepthDetail()
EquityMethodExerciser()
getClassKeyByProductKey()
getFirstAvailableClassKeyForSession()
acceptCancelResponse()
registerNBBOAgent()
unRegisterNBBOAgent()
lockProduct()
unlockProduct()
acceptOpeningPrice()
getAdminMessage()
getDetailedOrderBook()
printOrderDetail()
#nbboAgentAdminCallback
#intermarketOrderStatusCallback
IntermarketOrderStatusConsumer
(from example10)
IntermarketOrderStatusConsumer()
acceptCancelHeldOrderRequest()
acceptFillRejectReport()
acceptHeldOrderCanceledReport()
acceptHeldOrderFilledReport()
acceptNewHeldOrder()
acceptHeldOrderStatus()

Current Market V3 (Example 11)

This example describes how to subscribe and unsubscribe to current market V3. The example demonstrates how to:

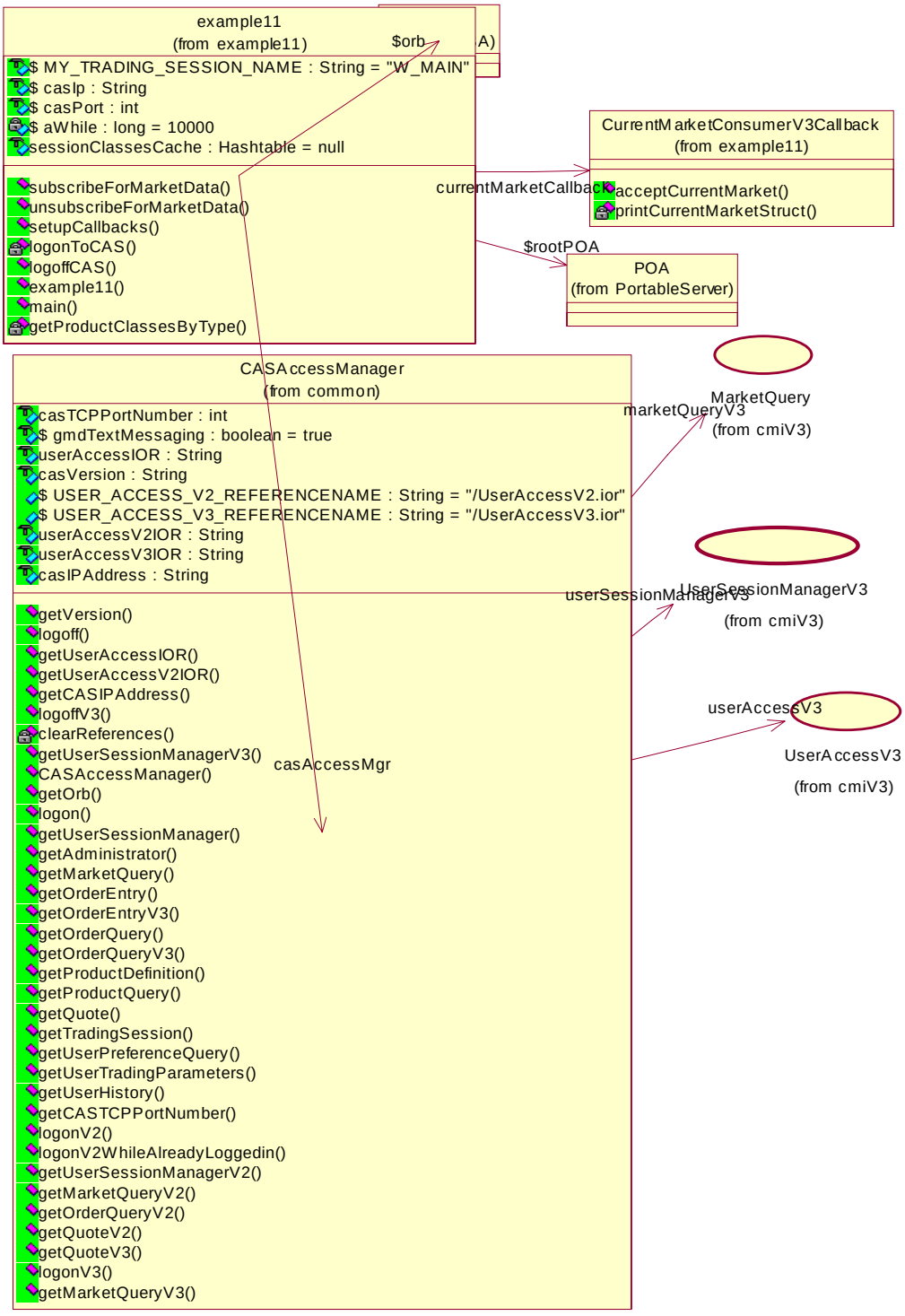
(1) Logon using V3 user session manager

(2) Subscribe to current market for classes using new market query interface

(3) Unsubscribe from current market for classes using new interface, and

(4) and Logout

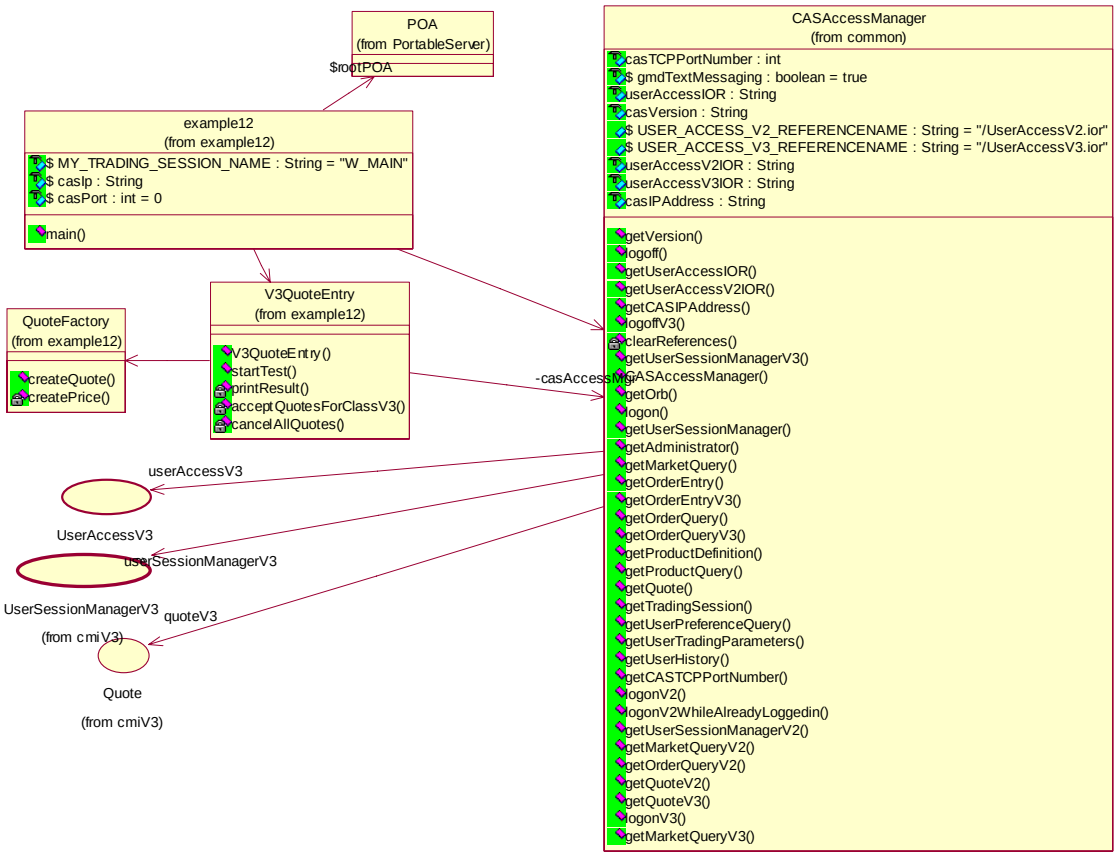
CurrentMarketV3 Class Diagram



Quote V3 (Example 12)

This example demonstrates the usage of the V3 interface to send mass quotes. The example depicts: (1) sending quotes with control id = 1, (2) cancelling all quotes and sending quotes with control id = 1 and (3) sending quotes with control id = 2. The result of each action will be printed.

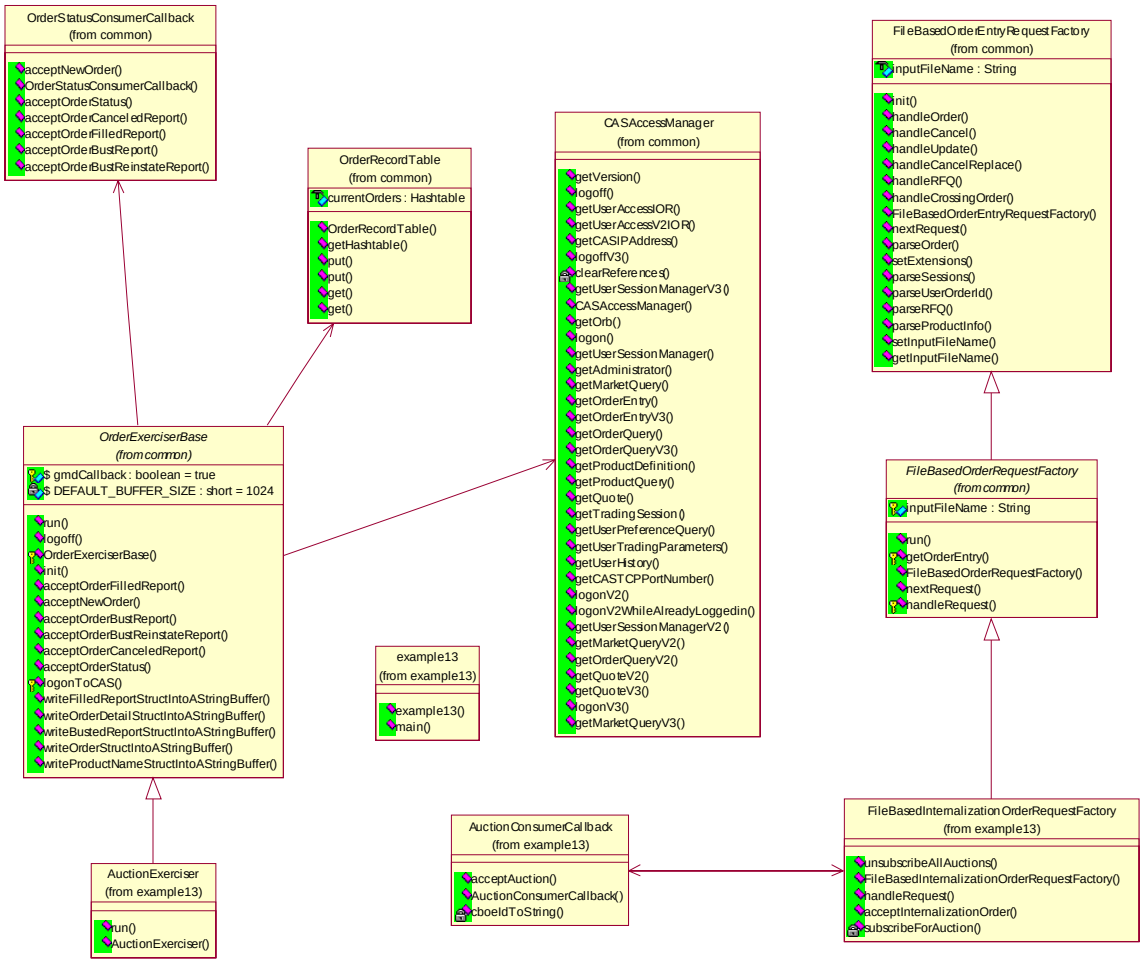
QuoteV3 Class Diagram



Auction and Internalization (Example 13)

This example reads paired orders from input and submits them to the CAS. It determines the class for the product in the order and subscribes to auctions on that class. When the program receives notice of an auction, it submits an order to participate in the auction.

Auction and Internalization Class Diagram



Multileg Strategy Order Entry (Example 14)

This code example shows how to request creation of a multi-leg strategy product via CMi. This involves the specification of individual strategy leg products and the quantity and side for each product in the strategy. After creating the strategy, the example code submits an order for the strategy using the acceptStrategyOrder method on the OrderEntry interface. Fields specific to individual strategy legs are included as part of the order entry operation. The usual order status callback method invocations by CBOEdirect for the strategy product are logged to standard output by the example code as the status messages are delivered.

CMi Interface Programmer's Guide

The CORBA IDL for the CBOE Market Interface is divided into the following modules:

Module	Description
cmi	The CMI module contains the interfaces to application server services. It contains the interfaces for requests between client and the CAS. References to the application server services objects are obtained via the SessionManager. A reference to the SessionManager is obtained via the user access service. Refer to the API documentation for a more thorough discussion.
cmiCallback	All CBOE Market Interface call back interfaces are contained within this module. The callback interfaces are used to support subscriptions to various information being published by the exchange.
cmiCallbackV2	All CBOE Market Interface call back interfaces are contained within this module. The callback interfaces are used to support subscriptions to various information being published by the exchange
cmiCallbackV3	All CBOE Market Interface call back interfaces for Version 3.0 are contained within this module. The callback interfaces are used to support subscriptions to various information being published by the exchange such as Top of the Book. In addition, the cmiCallbackV3 provides a public market parameter on the new callbacks to inform users of the Customer and Professional size if, and only if, there is Customer and/or Professional interest at the Top of the Book.
cmiCallbackV4	The CMiV4 callback interfaces are oneway callbacks. These are high performance calls that function differently than a standard CORBA call. Normally a CORBA function call will wait for the full round trip to complete before continuing. A oneway CORBA call will not wait for any acknowledgement from the server before continuing. In addition, the cmiCallbackV4 provides a sequence number to check for messags received in proper order.
cmiCallbackV5	The CMi CallbackV5 interfaces are used to notify CMi users of any outage from the subscriptions

Module	Description
	provided in the UserSessionManagerV8.
cmiErrorCodes	The exceptions thrown by the CMi interfaces contain a com.cboe.exceptions.ExceptionDetails object. Within the ExceptionDetails object is an error code that is set by the application raising the exception. Error codes from CMi are defined as constants in the cmiErrorCodes interfaces.
cmiIntermarket	All CBOE Market Interface Intermarket interfaces are contained within this module.
cmiIntermarketCallback	All CBOE Market Interface Intermarket call back interfaces are contained within this module.
cmiTradeMaintenanceService	Contains interfaces to accept block trades, EFPs and trade busts from One Chicago, LLC.
cmiV2	The CMI module contains the interfaces to application server services. It contains the interfaces for requests between client and the CAS. References to the application server services objects are obtained via the SessionManager. A reference to the SessionManager is obtained via the user access service. Refer to the API documentation for a more thorough discussion.
cmiV3	The CMI module contains the interfaces to application server services. It contains the interfaces for requests between client and the CAS. References to the application server services objects are obtained via the SessionManager. A reference to the SessionManager is obtained via the user access service. Refer to the API documentation for a more thorough discussion of V3 functionality.
cmiV4	The CMI module contains the interfaces to application server services. It contains the interfaces for requests between client and the CAS. References to the application server services objects are obtained via the SessionManager. A reference to the SessionManager is obtained via the user access service. Users connecting to the Market Data Express (MDX) data feed must reference V4 interfaces. Refer to the API-08 documentation for a more thorough discussion of V4 functionality and MDX.
cmiV5	The CMI module contains the interfaces to application server services. It contains the interfaces for requests between client and the CAS. References to the application server services objects are obtained via the SessionManager. A reference to the SessionManager is obtained via the user access service. Refer to the API documentation for

Module	Description
	a more thorough discussion of V5 functionality.
cmiV6	The CMI module contains the interfaces to application server services. It contains the interfaces for requests between client and the CAS. References to the application server services objects are obtained via the SessionManager. A reference to the SessionManager is obtained via the user access service. Users performing Market Maker Hand Held functionality must reference V6 interfaces. In addition, the V6 interfaces are used to registration for Directed AIM participation.
cmiV7	The CMI module contains the interfaces to application server services. It contains the interfaces for requests between client and the CAS. References to the application server services objects are obtained via the SessionManager. A reference to the SessionManager is obtained via the UserAccessV7 service. V7 interfaces use a synchronous call for order entry. Orders entered through the V7 interfaces will return a full OrderStruct containing all the pertinent order information, including the embedded OrderIDStruct.
cmiV8	The CMI module contains the interfaces to application server services. It contains the interfaces for requests between client and the CAS. References to the application server services objects are obtained via the SessionManager. A reference to the SessionManager is obtained via the UserAccessV8 service. The CMiV8 module provides a new interface to report outage on a trading class during trading hours. The new service is based on a subscription mechanism. Subscription could be made on a trading group (which is a list of classes) or at a class level within a trading group. Once a group/class is marked down. CBOE will not accept any new orders, cancel replace or quotes on that class. However, order/quote cancels will be processed.
cmiV9	The CMiV9 new interfaces support light orders. The CMI module contains the interfaces to application server services. It contains the interfaces for requests between client and the CAS. References to the application server services objects are obtained via the SessionManager. A reference to the SessionManager is obtained via the UserAccessV9 service.

Module	Description
cmiV10	The CMiV10 module provides new interfaces to support cancel/replace of light orders. It also provides the functionality to subscribe to receive only quote fill messages. Buy subscribing to receive only quote fill messages, the queuing of quote fill messages behind other quote status messages is avoided. It contains the interfaces for requests between client and the CAS. References to the application server services objects are obtained via the SessionManager. A reference to the SessionManager is obtained via the UserAccessV10 service.

The following modules are covered in *Volume 3 CMi Programmer's Guide to Messages and Data Types*.

Module	Description
cmiAdmin	Types and messages for use in administration of the CAS and Client interface.
cmiIntermarketMessages	Intermarket messages used in the CBOE Market Interface.
cmiMarketData	Order book depth information provides the total volume and contingency volume for the top N(5) best bid and ask prices. Order book depth information is available on a request not on a subscription basis. Access to this call will be limited on a per minute basis and by user type.
cmiOrder	The cmiOrder module contains messages for order routing. Message are available for submission of orders, cancel requests, request for quotes, and reports on order activity.
cmiProduct	Messages describing products within the system.
cmiQuote	The cmiQuote module contains structs and datatypes for Quotes and Request for Quotes. Messages are provided for quote status and quote filled reports.
cmiSession	Messages from the TradingSession service. The TradingSession service is used to determine which products trade in a given session. The service is also used to obtain session and product state information.
cmiStrategy	Contains structs and types required to support trading derivative strategies.
cmiTrade	Contains structs and types required to support

Module	Description
	external trades.
cmiTraderActivity	This is an optional interface - it is not needed for trading on CBOE markets. The cmiTraderHistory module contains messages used to report trader activity. An activity record is generated for each new quote and order and each subsequent activity on a quote or order. TraderHistory information is accessed using the cmi::UserHistory interface. Unlike the other interfaces, there are several different types of history records that are returned via the same UserHistory operation. Information is returned as a sequence of named value pairs. The UserHistory interface is provided to support CBOE applications. It is not necessary as part of a trading application. The UserHistory interface also does not replace existing CBOE interfaces for accessing matched trade information.
cmiUser	User preferences and privileges
cmiUtil	Common utility type used within the CBOE Market Interface