Development Guide

myOp€nTr@der

Lets get started with some development

It is assumed that you have successfully downloaded, installed, configured and started MyOpenTrader on your host.

Things to consider before getting started:

- Make sure that your initial development happens in standalone mode (not connected to any broker/feeder/exchange).
- Make sure that you always set the simulation flag. This flag will ensure that each trade/order is only used internally, but not sent to the exchanges/brokers.
- Test with small amounts of data and grow from there. Starting too complicated may not result in what you were looking for!
- Make sure that you have Maven & ANT installed on your development machine.

Create a new Strategy Client Project:

All of your development will be packaged into its own jar container. You can then later deploy the jar into your MyOpenTraderBin/libs folder and run the strategy in live mode. I am using maven for this - makes life much easier!

```
# Create a new project directory - in this case mot-testclient. Run this in your
command line:
mvn -B archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
-DgroupId=org.sg.test -DartifactId=mot-testclient
```

This should create a new directory called mot-testclient. In this directory, you will find the pom.xml file - go ahead and add the following dependency:

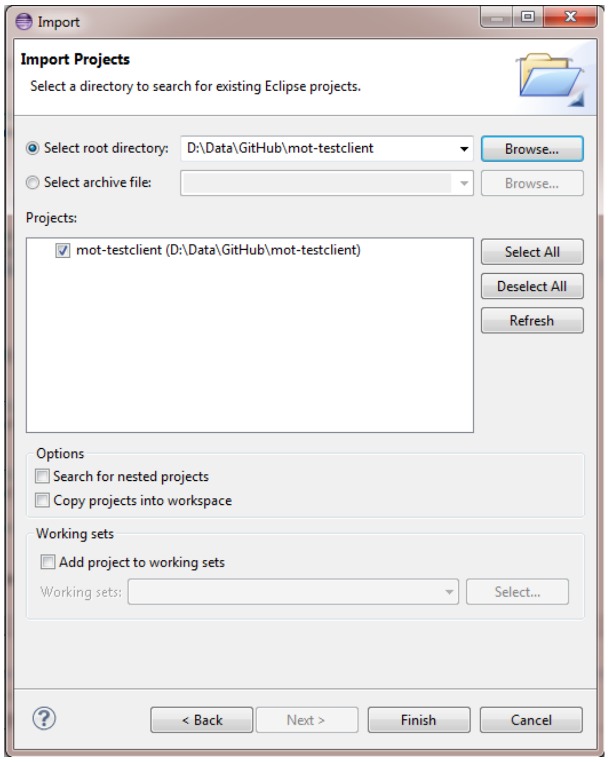
```
# Add into the dependencies block
...
 <dependency>
  <groupId>org.mot.common</groupId>
  <artifactId>MyOpenTraderCommon</artifactId>
  <version>LATEST</version>
 </dependency>
...
```

(I tend to just use the version=LATEST - you may want to limit your project to a particular version, to ensure that updates to the common package dont destroy your strategies)

Ready to start - first, lets create our project information for Eclipse or Netbeans etc:

```
# Run in your command line
mvn eclipse:eclipse
```

Open Eclipse (or whatever tool you prefer) and import the new project into your workspace:


Import

Import Projects

Select a directory to search for existing Eclipse projects.

(●) Select root directory: D:\Data\GitHub\mot-testclient

( ) Select archive file:

Projects:

[✓] mot-testclient (D:\Data\GitHub\mot-testclient)

Options

[ ] Search for nested projects

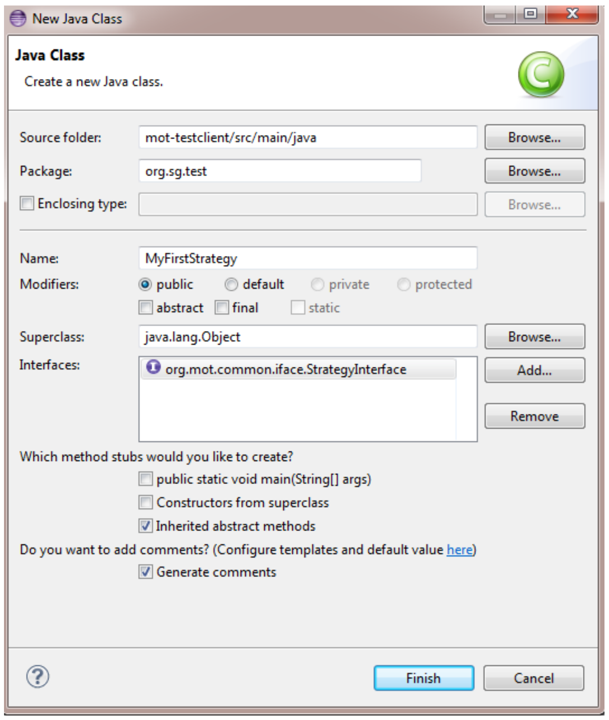
[ ] Copy projects into workspace

Working sets

[ ] Add project to working sets

Working sets:

You should see your project details and classes in the package explorer. Now simply add a new class to your project and make sure you add the StrategyInterface class to inherit all methods.


New Java Class

Java Class

Create a new Java class.

Source folder: mot-testclient/src/main/java

Package: org.sg.test

[ ] Enclosing type:

Name: MyFirstStrategy

Modifiers: (●) public ( ) default ( ) private ( ) protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

Interfaces: org.mot.common.iface.StrategyInterface

Which method stubs would you like to create?

[ ] public static void main(String[] args)

[ ] Constructors from superclass

[✓] Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

[✓] Generate comments

Your new class should look similar to this:

```
package org.sg.test;

import java.util.ArrayList;
import org.mot.common.iface.StrategyInterface;
import org.mot.common.objects.Expression;
import org.mot.common.objects.LoadValue;
import org.mot.common.objects.Order;
import org.mot.common.objects.SimulationRequest;
import org.mot.common.objects.SimulationResponse;
import com.espertech.esper.client.EventBean;

public class MyFirstStrategy implements StrategyInterface {
 /* (non-Javadoc)
  * @see
org.mot.common.iface.StrategyInterface#closeOrder(org.mot.common.objects.Order,
java.lang.Double, java.lang.Double, java.lang.Long)
  */
 @Override
 public Boolean closeOrder(Order arg0, Double arg1, Double arg2, Long arg3) {
  // TODO Auto-generated method stub
  return null;
 }

 /* (non-Javadoc)
  * @see org.mot.common.iface.StrategyInterface#getEsperExpression()
  */
 @Override
 public ArrayList<Expression> getEsperExpression() {
  // TODO Auto-generated method stub
  return null;
 }
...
<truncated>
```

Ready to go! You have now successfully created your first strategy wrapper... wohoo!

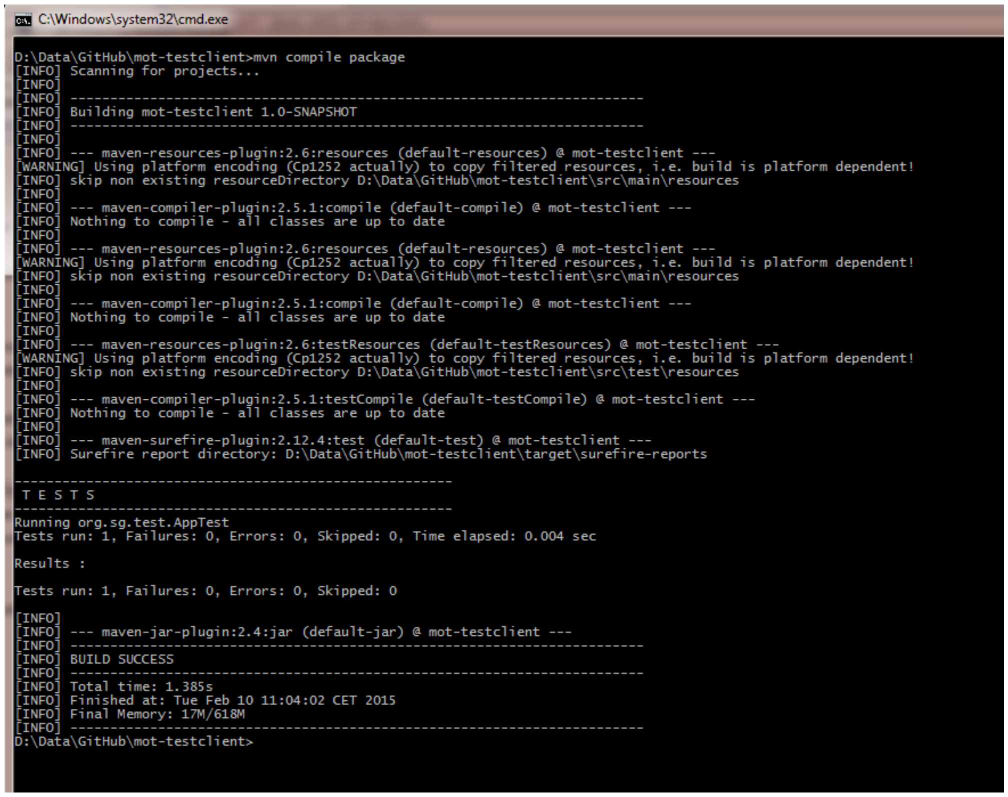
Start with your development - this is the tricky bit! Take a look at the StrategyInterface and its explanations.

Packaging & Deploying your strategy

Once you have created your own strategy and are ready to test, you first have to package your strategy, so that it can be deployed.

```
# We use maven again - simply run
mvn compile package
```

You should see:



```
D:\Data\GitHub\mot-testclient>mvn compile package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building mot-testclient 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ mot-testclient ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory D:\Data\GitHub\mot-testclient\src\main\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ mot-testclient ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ mot-testclient ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory D:\Data\GitHub\mot-testclient\src\main\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ mot-testclient ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ mot-testclient ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory D:\Data\GitHub\mot-testclient\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ mot-testclient ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ mot-testclient ---
[INFO] Surefire report directory: D:\Data\GitHub\mot-testclient\target\surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.sg.test.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ mot-testclient ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.385s
[INFO] Finished at: Tue Feb 10 11:04:02 CET 2015
[INFO] Final Memory: 17M/618M
[INFO] ------------------------------------------------------------------------
D:\Data\GitHub\mot-testclient>
```

If everything is correct (check the build status), Your new strategy is now compiled into the target directory ie. target/mot-testclient-1.0-SNAPSHOT.jar

Make sure to copy this jar file into your MyOpenTraderBin/libs directory. Now restart your MyOpenTrader Core Engine - this will read in your new classes.

Activating your new strategy:

Now that your new strategy is deployed into the MyOpenTrader libs folder, you are ready to start testing and activate your strategy.

As of 0.1-alpha, the web-interface to perform those actions is not ready. Therefore those steps have to be done manually. It requires you to add/modify your data base tables. Simply use Toad/PhpMyAdmin etc for this.

Connect to your database and add a new strategy into your strategy table:

```
# SQL command to add new strategy:
# The format is: (`ID`, `NAME`, `TYPE`, `SYMBOL`, `LOADVALUES`, `ENABLED`, `AMOUNT`,
`SIMULATED`, `TIMESTAMP`)
INSERT INTO `mot`.`strategies` VALUES ('999', 'myFirstStrategy',
'org.sg.test.MyFirstStrategy', 'STOCK#1', 'value1=abc;value2=def', '1', '1000', '1',
CURRENT_TIMESTAMP);
```

Here is a little bit to explain:

- The Type field is the path/namespace directive to your new strategy. In our sample test case: org.sg.test.MyFirstStrategy
- The symbol is a single value. Which symbol do you want this strategy to apply to. Stock#1, Stock#2, Stock#3 can be used for testing, as the tick simulator creates them. Later on modify this to reflect a real symbol.
- The amount field is indicating how much money you would like to invest into this strategy. For testing purposes lets set this to 1000 EUR
- The simulation flag is very important. **It is good practice to always set this field to TRUE!** This will make sure that the order engine will

not place any orders on the exchange. This could be a very costly mistake!!
- The loadvalues are dynamic properties. Use these settings for everything you dont want to hardcode into your strategy. These properties can modify the behavior of your strategy as they are passed in, at startup. Take a look at the loadValue Converter.

After another restart of your MyOpenTraderCore Engine, it will read in your new strategy and start its simulation.

Development guidelines:

The strategyInterface is the most important part of your work. This is where all the magic happens.

Method overview

Lets take a closer look at all of the methods:

Method Name	What does it do:
update	This is the main method, receiving all of the objects from the CEP Engine. This is where you pack your logic when to buy/sell/hold
getEsperExpression	This method is called upon startup to ask the strategy, which esper expressions it should register internally.
processSimulationRequests	*Not in use today*
getSimulationRequests	*Not in use today*
closeOrder	This method is used by the watchdog scheduler to ask if a particular order should be closed. This is useful, if you want to add stop-loss protection.
startup	The startup method. Use this as your constructor to read in all load values and create a new strategy object.
shutdown	The shutdown method. Called when a strategy is disposed.

Steps to develop a new strategy:

First of all, we need to implement the startup method:

What does the startup do?

The Interface methods: startup() is used as a constructor for your new strategy. It will pass in important startup parameters, such as the invested amount and the loadvalues. Always make sure you have the constructor set up properly. This is key for any further development.

When is the startup called?

The method is called upon startup of the MyOpenTrader Core Engine. It is called once when the strategy is registered in the engine.

Inputs:

The method will provide the following inputs.

Field Name:	Description:
String name	Provides the name of the strategy, as defined in the MOT database.
String symbol	Provides the symbol of the strategy, as defined in the MOT database.
Array of LoadValues	Provides the loadvalues as specified in the MOT database. Use the loadValueConverter to extract the values
Boolean simulated	Provides the simulation flag - you have to pass this flag to the orderEngine to ensure no order is routed to the exchange.
Integer amount	Provides the investment amount, as defined in the MOT database. This is the total amount of your investment.

Outputs:

(void) The method does not request any output.

Sample code:

```
// These values should persist
 protected String symbol;
 protected boolean isSimulated;
 protected int amount;

 // Get a new loadValueConverter instance
 LoadValueConverter lvc = new LoadValueConverter();

 @Override
 public void startup(String name, String symbol, LoadValue[] values, Boolean
simulated, Integer amount) {

  // Use the loadValueConverter and extract variables var1 & var2 from it
  String var1 = lvc.getValue(values, "var1"));
  Integer var2 = Integer.valueOf(lvc.getValue(values, "var2"));

  // Persist these values for other methods...
  this.name = name;
  this.symbol = symbol;
  this.isSimulated = simulated;
  this.amount = amount;

 }
```

After this, we can register our esper expressions:

What does the getEsperExpressions do?

The Interface methods: getEsperExpressions() is used to ask your strategy, which expressions should be registered internally. The MOT Core engine will start up and check each registered strategy. It will then query the esper expressions and register them in the core engine. Your expressions will get activated.

When is the getEsperExpressions called?

The method is called after the startup method has been initialized. It will be executed once for startup and another time when shutting down the strategy.

Inputs:

There are no inputs.

Outputs:

The method will request the following outputs.

Field Name:	**Description:**
ArrayList of Expressions	Return an arraylist of expressions, which should be registered in the core MOT Esper engine

Sample code:

The Esper Complex event processing engine can become quite complex. Take a look at the query syntax: http://esper.codehaus.org/esper-4.2.0/doc/reference/en/html/epl_clauses.html

```
public ArrayList<Expression> getEsperExpression() {
  // Create a new empty array
  ArrayList<Expression> array = new ArrayList<Expression>();

  int win1 = 50;
  int win2 = 100;

  // Create a first expression
  Expression e1 = new Expression();

  // Set the name of the expression - you may need this name later, so make sure to
remember it!
  e1.setName(name + "-mvgbidPrice");

  // Set the expression - in this case:
  // Select the price of the last element for all BID ticks, where symbol is equal to
whatever strategy is being executed
  e1.setExpression("select price from org.mot.common.objects.Tick(priceField =
'BID').std:lastevent() where symbol = '" + symbol +"'");

  // Create a second expression
  Expression e2 = new Expression();
  e2.setName(name + "-mvgavg1");

  // Lets make it a bit more complex.
  // Select a number of fields, including average and standard deviation calculations,
from two tables (t1 and t2) - where each table has a different window size (t1 can
hold 50 ticks and t2
  // can hold 100 ticks) and join those two tables together
  e2.setExpression("select t1.symbol, t1.price, count(t1.price), avg(t2.price) as
t2price, stddev(t1.price) as stddev, avedev(t2.price) as t2avedev from
org.mot.common.objects.Tick(priceField = 'ASK').win:length(" + win1 + ") as t1,
org.mot.common.objects.Tick(priceField = 'ASK').win:length(" + win2 + ") as t2 where
t1.symbol = '" + symbol +"' and t1.symbol = t2.symbol");

  // Add the expressions to the array
  array.add(e1);
  array.add(e2);

  // Hand the array back to the calling engine...
  return array;
 }
```

Lets implement the fancy logic:

What does the Update do?

The Interface methods: update() is called directly by the Esper engine as an updateListener. Your strategy was been successfully registered with the core MOT engine and your expression has been executed. This is where the main trading logic will go into.

When is the Update called?

The method is called several times. Each time your esper expression is being executed.

Inputs:

From the Esper documentation (http://esper.codehaus.org/esper-4.9.0/doc/api/com/espertech/esper/client/UpdateListener.html):

Notify that new events are available or old events are removed. If the call to update contains new (inserted) events, then the first argument will be a non-empty list and the second will be empty. Similarly, if the call is a notification of deleted events, then the first argument will be empty and the second will be non-empty. Either the newEvents or oldEvents will be non-null. This method won't be called with both arguments being null (unless using output rate limiting or force-output options), but either one could be null. The same is true for zero-length arrays. Either newEvents or oldEvents will be non-empty. If both are non-empty, then the update is a modification notification.

The method will provide the following inputs.

Field Name:	Description:
Array of EventBeans (newEvents)	Provides all new (inserted) events to your strategy
Array of EventBeans (oldEvents)	Provides all removed events to your strategy

Outputs:

(void) The method does not request any output.

Sample code:

```
public void update(EventBean[] newEvents, EventBean[] oldEvents) {

 // Remember - we registered the following statement in our engine:
 // "select t1.symbol, t1.price, count(t1.price), avg(t2.price) as t2price,
stddev(t1.price) as stddev, avedev(t2.price) as t2avedev
 // from org.mot.common.objects.Tick(priceField = 'ASK').win:length(" + win1 + ") as
t1, org.mot.common.objects.Tick(priceField = 'ASK').win:length(" + win2 + ")
 // as t2 where t1.symbol = '" + symbol +"' and t1.symbol = t2.symbol"

 // Lets get some values out of this expression
 // Get the t2price  (which is the average) and map it into var1
 Double var1 = ef.getLastDoubleValue(name + "-mvgavg1", "t2price");

 // Get the stddev value from the expression (which is the standard deviation) and map
it into var2
 Double var2 =  ef.getLastDoubleValue(name + "-mvgavg1", "stddev");



 ...
 // Do some fancy logic here
 ...
 }
}
```

and finally, lets make sure we have a stop-loss protection:

What does the closeOrder do?

The Interface method: closeOrder is executed by the MOT Core Engine internal watchdog scheduled service. The watch dog is used to periodically check all open orders. If an order is open, it will invoke the close order method of your strategy, to make sure this should still be open. You can define safe guards here to make sure your protected from massive losses. **(Be sure to check that the watchdog is enabled !!)**

When is the closeOrder called?

The method is called repeatedly, as defined in the watchdog configuration (usually every 60 seconds - *watchdog.frequency.seconds=60* - see scheduler.properties). This method will only get called, if your strategy has an open order quantity. You can also query the orderEngine to find out how many shares are still open for a particular strategy.

Inputs:

The closeOrder method has the following inputs

Field Name:	Description:
Order order	The order in question. Should this order be closed?
Double lastKnownPrice	The Double value, representing the last known price (as per our Database)
Double diffPct	The Difference in Percent (from the original order buy price)
Long openMinutes	How long has this order been open (in minutes)?

Outputs:

The following outputs are required:

Field Name:	Description:
Boolean close	If true, the watchdog will close the position. If false, it will ignore the order until next run.

Sample code:

```
public Boolean closeOrder(Order order, Double lastKnownPrice, Double diffPct, Long
openMinutes) {
  boolean close = false;

  // Close the order if we lost more than 5 pct
  if (diffPct < -5) {
   close = true;
  }

  // Automatically close if the order is older than 6 days (8640 minutes)...
  // Dont chase any further!
  if (openMinutes > 8640) {
   close = true;
  }

  return close;
 }
```

Strategy Interface

```
/*
  * Copyright (C) 2015 Stephan Grotz - stephan@myopentrader.org
  *
  * This program is free software: you can redistribute it and/or modify
  * it under the terms of the GNU General Public License as published by
  * the Free Software Foundation, either version 3 of the License, or
  * (at your option) any later version.
  *
  * This program is distributed in the hope that it will be useful,
  * but WITHOUT ANY WARRANTY; without even the implied warranty of
  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  * GNU General Public License for more details.
  *
  * You should have received a copy of the GNU General Public License
  * along with this program.  If not, see <http://www.gnu.org/licenses/>.
  *
  */


  package org.mot.common.iface;
import java.util.ArrayList;
import org.mot.common.objects.Expression;
import org.mot.common.objects.LoadValue;
import org.mot.common.objects.Order;
import org.mot.common.objects.SimulationRequest;
import org.mot.common.objects.SimulationResponse;
import com.espertech.esper.client.EventBean;
import com.espertech.esper.client.UpdateListener;
 /**
  * @author stephan
  *
  */
 public interface StrategyInterface extends UpdateListener {

  /**
   * This is the main method for receiving any tick updates from the CEP Engine.
   * Two arrays are passed in, one with the new events, another one with all old
events.
   *
   * (non-Javadoc)
   * @see
com.espertech.esper.client.UpdateListener#update(com.espertech.esper.client.EventBean[
], com.espertech.esper.client.EventBean[])
   */
  void update(EventBean[] newEvents, EventBean[] oldEvents);

  /**
   * This method is used to return any esper expressions, which want to be run as part
of the strategy.
   * As part of this, the code should provide a name and an expression. Use the name
in the update method
```

```
   * to access the values.
   */
  ArrayList<Expression> getEsperExpression();


  /**
   * This method is used to simulate back tests. These tests are important for
simulating the past
   * performance of a stock. This method will be called by the Backtest simulator...
   */
  ArrayList<SimulationResponse> processSimulationRequests(SimulationRequest sr);


  /**
   * This method is called by the simulation loader. Simply return any combinations of
simulation requests -
   * the simulation loader will then send to the message bus for processing.
   */
  ArrayList<SimulationRequest> getSimulationRequests(String symbol, String className,
String frequency, Integer quantity, String startDate, String endDate, Double txnPct,
Double minProfit) ;


  /**
   * This method is called by the watchdog scheduled service to check if a particular
order shall be closed
   * for "special" reasons. This could happen, if a trade is stuck or losses were too
large. This is an easy way to
   * protect for stop-lossing.
   *
   * @param order - should this order be closed?
   * @param lastKnownPrice - the last known price as per our internal database.
   * @param diffPct - the difference between the order buy price and the last known
price in percent
   * @param openMinutes - the time the order has been open in minutes (if order has
not been closed in x days/minutes - then close)
   * @return - indicate if you want the order to be closed (true) or not (default -
false)
   */
  Boolean closeOrder(Order order, Double lastKnownPrice, Double diffPct, Long
openMinutes);


  /**
   * The startup method is used as the main constructor. It is called once at
bootup/initialization phase.
   * This should be used to initiate any new strategy
   */
  void startup(String name, String symbol, LoadValue[] values, Boolean simulated,
Integer amount);

  /**
   * The shutdown method is not yet in use, but should be implemented in a later
release.
   */
  void shutdown();
```

```
}
```