

A shallow path into ActiveQuant

Ghost Rider <ghostrider@activequant.org>

June 3, 2011

Contents

I	Introduction	5
1	About ActiveQuant	7
2	Short history of ActiveQuant	9
3	Project web infrastructure	11
3.1	URLs	11
4	Technical introduction	13
II	Installation	15
5	... using maven	17
6	... using SVN	19
III	Technicals	21
7	Module structure	23
7.1	activequant-framework	23
7.2	activequant-adapter-ib	23
7.3	activequant-adapter-simple	23
7.4	activequant-dao-hibernate	23
7.5	activequant-filtering	23
7.6	activequant-regression	24
7.7	activequant-messaging-jabber	24
7.8	activequant-messaging-email	24

7.9	activequant-util-spring	24
7.10	activequant-util-math	24
7.11	activequant-util-charting	24
8	Spring and ActiveQuant	25
9	Domain model	27
9.1	Generic	27
9.1.1	InstrumentSpecification	27
9.1.2	Timestamp	27
9.1.3	Timeframe	28
9.1.4	Time Series	28
9.1.5	Series Specification	28
9.1.6	Candles	28
9.1.7	Quotes	28
9.1.8	TradeIndications	29
9.1.9	IndexComposition	29
9.1.10	IndexComponent	29
9.2	Domain model, trading related	29
9.2.1	Position	29
9.2.2	Portfolio	29
9.2.3	Account and BrokerAccount	29
9.2.4	BalanceBook	29
9.2.5	Order	30
9.2.6	OrderEvent	30
10	DAO - saving stuff to database	31
10.1	Configuring the DAO layer	31
10.2	Instantiating the DAO layer	31
10.3	Instrument Specifications	32
10.4	Ticks, Quotes - or in general MarketDataEntities	32
11	Data sources - listening to the world	35

CONTENTS	5
11.1 Theory	35
11.1.1 Event pattern	35
11.1.2 Pull sources	35
11.1.3 Pushing subscription sources	35
11.2 Practice	36
11.2.1 Pull sources	36
12 Logical building blocks	37
12.1 Trade System	37
12.2 Broker interfaces	37
12.2.1 Theory	37
12.2.2 Practice	39
IV P2	41
13 Introduction	43
14 Processes	45
15 Tools	47
15.1 Config Persister	47
15.2 Data Relay	47
15.2.1 Protocol used	47
15.2.2 Instrument specification	47
15.2.3 Quote lines	48
15.2.4 Trade Indication lines	48
15.2.5 Using and starting it	48
15.3 Random Data Generator	48
15.4 Quote Recorder	48
V Pushing the frontier further	51
16 AQ + R = AQ-R	53
16.1 Requirements for using AQ and R	53

16.2 Functionality by use cases 53

 16.2.1 Loading quotes from an AQ DB 53

Part I

Introduction

Chapter 1

About ActiveQuant

This document provides an overview over the activequant framework. ActiveQuant is a framework for many things related to quantitative finance, trading and reporting.

The main website is located at <http://www.activequant.org> .

ActiveQuant has always and will always follow a pragmatic and goal oriented approach. What is needed to get the job done. Overengineering and using technology for the sake of hippness has no place in ActiveQuant's down-to-the-metal approach.

Chapter 2

Short history of ActiveQuant

ActiveQuant (AQ) is an open source framework for automated trading, quantitative finance and many finance things with an active community and a growing user base. It contains java implementations as well as ruby and R parts. The history of AQ reaches back to 2002 when some people wanted to trade in an automated way but did not find any appropriate open source package to achieve this goal. Out of this necessity the original CCAPI¹ package was born and got its later name *ActiveQuant* after a late night chat between Erik and Ulrich.

¹Cortal Consors API

Chapter 3

Project web infrastructure

3.1 URLs

The following URLs are seriously important:

- <http://www.activequant.org> - the main landing page
- <http://jdoc.activequant.org> - the JavaDocs of ActiveQuant
- <http://forums.activequant.org> - the discussion forums around ActiveQuant
- <http://developers.activequant.org> - the developer's space around ActiveQuant

Chapter 4

Technical introduction

AQ is a java framework heavily based on maven, spring, hibernate and some other minor libraries. Users should be familiar with maven and to a lesser degree with spring and hibernate. The software is distributed solely over the internet in the form of maven packages that can be referenced in any maven project with very little effort.

Part II

Installation

Chapter 5

... using maven

AQ is distributed as a maven library and thus requires the typical maven setup. Although it is also possible to directly download and reference the jars in projects, the proper integration over maven is highly recommended over using source code.

Assuming you are familiar with maven, the following paragraphs should give you a fairly straight forward instruction on how to reference the AQ libraries in your project.

First, announce the AQ repository in your project by adding AQ's repositories into the project's *<repositories>* section of its pom file.

Listing 5.1: Announcing the AQ repository to maven

```
<repository >
  <id>activequant-external-repository </id>
  <url>http://activequant.org/m2/external </url>
  <snapshots >
    <enabled>false </enabled>
  </snapshots >
</repository >
<repository >
  <id>activequant-repository </id>
  <url>http://activequant.org/m2/repo </url>
  <releases >
    <enabled>true </enabled>
  </releases >
  <snapshots >
    <enabled>true </enabled>
  </snapshots >
</repository >
```

Second, reference preferred AQ modules as dependencies in the project's *<dependencies>* section of its pom file. Several AQ modules have to be added depending on which parts of AQ are to be used in the project.

Listing 5.2: Adding AQ as a project dependency

```
<dependency>
  <groupId>org . activequant </ groupId >
  <artifactId >activequant –framework </ artifactId >
  <version >1.6 </ version >
</dependency >
```

With these steps done, the project is ready to be built, for example through the typical maven command *mvn build*.

Chapter 6

... using SVN

For the truly brave, the AQ website offers all SVN source code. In principle, all development takes place in trunk, which especially developers are encouraged to use. We assume you are familiar with SVN and maven.

Let's see what activequant-base contains ...

Listing 6.1: Checking out AQ from SVN

```
grider@sniardwy:~\$ svn ls \  
svn://activequant.org/opt/repositories/sandbox/activequant-base/trunk
```

HISTORY

LICENSE

README

```
activequant-adapter-ib/  
activequant-adapter-simple/  
activequant-dao-hibernate/  
activequant-filtering/  
activequant-framework/  
activequant-messaging-email/  
activequant-messaging-jabber/  
activequant-regression/  
activequant-util-charting/  
activequant-util-math/  
activequant-util-spring/  
pom.xml  
src/  
grider@sniardwy:~\$
```

The printed modules are explained at another place. Now, let's check out the full code tree.

Listing 6.2: Checking out AQ from SVN

```
grider@sniardwy:~\$ svn co \  
svn://activequant.org/opt/repositories/sandbox/activequant-base/trunk \  
aqt-base-trunk
```

```
A    aqt-base-trunk / activequant-framework
A    aqt-base-trunk / activequant-framework / src
A    aqt-base-trunk / activequant-framework / src / test
A    aqt-base-trunk / activequant-framework / src / test / java
A    aqt-ba ...
[... ]
A    aqt-base-trunk / activequant-filtering / src / main / java / org / ...
A    aqt-base-trunk / activequant-filtering / src / main / java / org / ...
A    aqt-base-trunk / activequant-filtering / pom.xml
U    aqt-base-trunk
Checked out revision 721.
grider@sniardwy:~\$
```

In order to build the checked out source, use maven.

Part III

Technicals

Chapter 7

Module structure

AQ consists in raw form of various modules, which you can but don't have to use at the same time. Each of these modules can be referenced explicitly in maven's pom file. In case a module depends on another one, maven will pull in the necessary dependencies.

Following a list and a short description.

7.1 activequant-framework

The framework package declares all interfaces, domain objects and central utilities. It is the base for all other packages.

7.2 activequant-adapter-ib

feed and broker implementation, using connection to TWS station from Interactive Brokers

7.3 activequant-adapter-simple

misc web-based feed implementations (Yahoo, Google, etc).

7.4 activequant-dao-hibernate

SQL persistence layer implemented via Hibernate.

7.5 activequant-filtering

filter-style processing of data series.

7.6 activequant-regression

regression utilities.

7.7 activequant-messaging-jabber

Contains various jabber related messaging modules. Very handy to build reporting bots.

7.8 activequant-messaging-email

Wrapper functions to build reporting messaging services, for example daily email reports.

7.9 activequant-util-spring

context-building utilities using Spring

7.10 activequant-util-math

The Math utilities module contains various math related functionality, for example some statistical methods, regression, some indicator implementations. More specialized finance mathematical computations are available in other source code modules, for example JQuantlib.

7.11 activequant-util-charting

Activequant also contains some medium advanced charting functionality, mostly to visualize real time or historical number series. High level functionality like chart interaction, drawing trendlines or similar aspects are reduced to the interactivity features coming along the JFreeChart module.

Existing functionality includes methods that work nicely with the data structures and procedures present in ActiveQuant. For example, it is easily possible to generate a candle stick chart, once a CandleSeries object is available.

(Almost-) Realtime charts are also possible, of course within the limits of a computer's performance. An example of this is the ActiveQuant JMS charting user interface GUI1.

Chapter 8

Spring and ActiveQuant

Spring has become the standard framework for all sort of Java applications. ActiveQuant recognized the need for flexibility very early and switched to Spring at around 2007. As a result of this, developers familiar with Spring have a significant advantage. Spring has been the framework to ram concepts like IoC and dependency injection into the world's development departments.

Chapter 9

Domain model

Several building blocks and preassumptions have to be cleared before we can dive deeper. ActiveQuant has a relatively consistent and stable domain model. This domain model is used throughout the whole codebase, so get familiar with it to fully leverage and understand all code.

The domain model is split into a generic type of objects and purely trading related objects. Almost all objects contain some sort of an ID, especially when they need to be stored.

9.1 Generic

9.1.1 InstrumentSpecification

An instrument specification is a definition of an asset in a generic way. At the present, these objects are not inherited from other classes and contain most parameters required for specifying which future, bond, option, stock to trade and where¹. Although it might be a bit out of context, it is highly recommended that instrument specifications are always used and passed on as references.

AQ already contains successor classes for different instrument types, namely AQFuture, AQIndex, AQListing, AQStock, etc., but these types are not in wide use due to the large codebase affected by introducing such a drastical improvement.

9.1.2 Timestamp

All timestamps in AQ are coming in nanosecond precision, hence the specific class for this. Not all data sources provide nanosecond precision, these data sources should use the UniqueTimeStampGenerator to generate non-ambiguous timestamps with nanosecond precision².

¹see <http://jdoc.activequant.org/org/activequant/core/domainmodel/InstrumentSpecification.html>

²see <http://jdoc.activequant.org/org/activequant/util/tools/UniqueDateGenerator.html>

9.1.3 Timeframe

The time frames inside AQ are all standardized through the class `TimeFrame`. The time frame class contains a `Unit`, which is an enumeration. Inside the `TimeFrame` instance, several ready made, public static instances exist. Examples of predefined time frames are :

Listing 9.1: Examples of `TimeFrame` enums

```
TimeFrame . TIMEFRAME_1_TICK  
TimeFrame . TIMEFRAME_1_SECOND  
TimeFrame . TIMEFRAME_5_MINUTES  
TimeFrame . TIMEFRAME_1_DAY
```

Time frames are typically used in series specifications, but can also be of use in configurations, etc.

9.1.4 Time Series

The time series object is the base class for all derived time series like `QuoteSeries`, `TradeIndicationSeries` and `CandleSeries`³. A timeseries is linked to a series specification. As such, the resolution of a specific time series can be found in the series specification. The concrete implementation of a time series domain model, the `TimeSeries` class, also contains several helper functions to remove entities by a timestamps and to get a sublist of entities within a timestamp range.

A time series object is a generic object that can hold an arbitrary amount of `MarketDataEntity`⁴ objects.

9.1.5 Series Specification

A series specification extends the instrument specification with a resolution and a time frame (start and stop timestamp)⁵. Resolutions are specified through a `TimeFrame` instance.

9.1.6 Candles

Candles are essentially OHLC data sets with additional volume data. The AQ candles are a bit special as they also can (but not must) contain a high timestamp and a low timestamp that can be used to show the internal price flow of the candle (whether high or low was reached first).

Candles are linked to an instrument specification reference and contain a timestamp.

9.1.7 Quotes

Quotes are market quotations of prices that buyers and sellers put into an exchange. This is the so called bid/ask. Quote objects can also contain an arbitrary number of additional quotes (level 5, 10, etc.). However, the first quotes are always best bid and best ask.

³see <http://jdoc.activequant.org/org/activequant/core/domainmodel/data/TimeSeries.html>

⁴see <http://jdoc.activequant.org/org/activequant/core/domainmodel/data/MarketDataEntity.html>

⁵see <http://jdoc.activequant.org/org/activequant/core/domainmodel/SeriesSpecification.html>

Quotes are linked to an instrument specification by reference and contain a timestamp.

9.1.8 TradeIndications

TradeIndications are objects that represent actual trades. These objects are in some contexts called LastTraded, Tick or otherwise. Whenever a trade happens at an exchange, a TradeIndication is generated in ActiveQuant (only if the broker/data interface supports it).

TradeIndications are linked to an instrument specification by reference and contain a timestamp.

9.1.9 IndexComposition

A container to describe an index composition. Good to fully describe how the index is weighted. Contains index components.

9.1.10 IndexComponent

A component of an index composition. Contains the actual weight in the encapsulating index.

9.2 Domain model, trading related

Trading related domain model objects include all parts that are directly related to trading.

9.2.1 Position

A position that is held in an InstrumentSpecification.

9.2.2 Portfolio

A list of all positions with some special functionality.

9.2.3 Account and BrokerAccount

An account object is what one could (!) associate with a clearing account. In AQ an account can contain multiple BrokerAccounts, which can keep individual positions. The account object shows an aggregated view on all broker accounts.

Every account, whether plain or broker account, contains a portfolio, an order book and a balance book.

9.2.4 BalanceBook

The balance book contains balance book entries, which resemble changes in value, for example due to cash payments or portfolio value changes. Mind that there is no direct link between the a balance book entry and the

reason that caused the balance book entry.

The balance book's entries can have a description field, very similar to what a bank account shows.

9.2.5 Order

An order is a central object for trading. An order is essentially an object that tells a counterparty what should be bought or sold, at which price, at which exchange and with which other conditions⁶.

9.2.6 OrderEvent

Base class for all events that can happen during an order lifecycle. Orders contain list of events that document the different states, such as `OrderAccepted`, `OrderSubmitted`, `OrderCancelled`, `OrderPartialFill`, `OrderFilled`, `OrderRejected`.

`OrderEvents` can resemble a terminal state, which means that no other events are expected beyond that terminal state - it's a final state.

⁶See Hull for details.

Chapter 10

DAO - saving stuff to database

The DAO layer is at the heart of all storage related aspects. Basically everything within AQ can be saved to database, or any other persistence layer when you implement it. By default, the hibernate DAO layer brings everything in. The DAO layer implements the well known CRUD functions.

The DAO module implements the dao interfaces. DAO stands for Data Access Objects. At the moment only a hibernate implementation is available. In the past, several other implementations existed, but because of maintenance constraints, these have been removed. DAO interfaces exist for most of the domain model objects.

10.1 Configuring the DAO layer

The DAO layer looks by default in JAVA's classpath for a file called *database-activequant.properties*, but you can also specify your own database configuration file, by passing a custom spring xml config file to the DAO factory's constructor and referencing in that file a different database configuration file. The configuration file must have the form of an ordinary Java properties file.

A typical properties file looks like this:

Listing 10.1: Example of a data bank configuration file

```
database.driver=com.mysql.jdbc.Driver
database.url=jdbc:mysql://192.168.0.177:3306/activequant?user=root&
    password=&create=true
database.dialect=org.hibernate.dialect.MySQL5Dialect
database.showSql=true
```

The filename of the properties filename is specified in a file called *activequantdao/config.xml*, which is compiled into the jar.

10.2 Instantiating the DAO layer

The DAO layer can be fairly trivially instantiated by a couple of lines of code, as shown below.

Listing 10.2: Instantiating AQ's DAO layer

```
IFactoryDao factoryDao = new FactoryLocatorDao("activequantdao/config.xml");
```

As visible, the first parameter of the DAO factory is the configuration's filename. It is possible to point to another config file, have a look at the original config file name to see how to point to different bean implementations.

Once the DAO layer is instantiated, you are free to use all parts or only some parts of it. To retrieve the specification dao, fetch it from the factory dao.

Listing 10.3: Obtaining the instrument spec dao facade

```
ISpecificationDao specDao = factoryDao.createSpecificationDao();
```

A complete list of all interfaces offered by the factory is available in the JavaDocs¹.

10.3 Instrument Specifications

Instrument specs are easy to save and search. You can also use the find function to look for specific instrument specs. The following code snippet shows how to load the spec for a specific instrument ID.

Listing 10.4: loading an instrument specification

```
InstrumentSpecification spec = specDao.find(1L);
```

AQ also offers functionality to look for a specs that match a certain pattern. For example the call to load all specs with symbol DAX, which would include all options and futures of all expiries, looks like this:

Listing 10.5: loading a list of specs

```
// search for all DAX specs
InstrumentSpecification[] specs = specDao.findAll(new Symbol({} 'DAX'));
// ... or simply load all specs
InstrumentSpecification[] specs = specDao.findAll();
```

10.4 Ticks, Quotes - or in general MarketDataEntities

Ticks, Quotes, etc. are all extending the same base class, namely MarketDataEntities. As AQ implements basic CRUD functions for all data types, using the DAO layer is trivial once more.

To save a quote, call something like:

Listing 10.6: Saving a quote

```
Quote q = new Quote();
q.setBidPrice(100.0); q.setAskPrice(100.0);
```

¹see <http://jdoc.activequant.org/org/activequant/dao/IFactoryDao.html>

```
q.setInstrumentSpecification(spec);  
// finally save it.  
quoteDao.update(quote);
```

Loading a list of all ticks between a certain time frame can raise serious requirements on your computer's memory. To load all ticks for a specific instrument between two dates, it is sufficient to construct a series specification that encapsulates the instrument specification, a start and end date and a time frame, that is passed to a trade indication dao instance.

Listing 10.7: loading all ticks between two dates

```
TradeIndication[] ticks = tradeIndicationDao.find(  
    new~SeriesSpecification(  
        1L,  
        new~TimeStamp(20100101),~  
        new~TimeStamp(20100102),  
        TimeFrame.TICK  
    )  
);
```

Chapter 11

Data sources - listening to the world

Data sources are using in AQ for all sorts of information. Quotes, Ticks, Candles and other things. All data source implementations extend interfaces that are defined in AQ, too.

AQ distinguishes between two types of data sources, pull sources and pushing subscription sources. The first are relatively trivial and are for example implemented as DAO objects. The second are a bit trickier, but nonetheless easy to use once you got it.

11.1 Theory

11.1.1 Event pattern

Crucial to understanding data sources is the event pattern in AQ. The event pattern was introduced by Erik Nijkamp. The event pattern is a very straight forward implementation example of an observer pattern.

11.1.2 Pull sources

Series data sources are named that way to indicate that this data source will return a time series of information. Propably the most widest known series data source is the YahooCandleSeriesSource. This source leverages the yahoo data source, it downloads O/H/L/C/V information from Yahoo and converts these into candles, hence also the name pull source. For almost all timeseries related domain model objects exist some series data sources, for example IBQuoteSeriesSources. DAO interfaces usually are more of this sort as well, as series data sources resemble on another page, pull sources, contrary to subscription sources that are considered to be push based.

11.1.3 Pushing subscription sources

Subscription sources are by definition data sources at which a listener has to subscribe at to receive certain data. Subscription sources exist not only on the tick level, the most precise level, but also at other resolutions. Using some special classes of ActiveQuant it is possible to use a list or an array of data as the underlying data of a subscription source. A possible scenario is backtesting.

11.2 Practice

11.2.1 Pull sources

For the sake of these examples, let's assume we want to download ordinary OHLC time series. The relevant interface definition is contained in `ICandleSeriesSource`¹ and in `ISeriesDataSource`. By using the interface in your applications, you can transparently switch from one implementation to another without even touching the rest of the code. By the time of this writing, AQ knows only series related pull data sources, but not other data sources.

The logical steps to fetch something from a pull source are:

- Construct an instrument specification
- Construct a series specification, encapsulating the instrument spec
- instantiate a pull data source, for example `YahooCandleSeriesSource`
- Call the series source's `fetch` method and obtain a series object.

... yahoo, google, etc.

The following code fragment shows how to fetch historic daily OHLC dataset of MSFT from Yahoo. Yes, it is a very boring and overly simple example, but it fits here as an introduction very well, as it presents the concept of pull sources. If this is too simple, check out the Google or Ameritrade pull sources. And to go even further, use the `InteractiveBroker` quote history pull source to get the full picture.

Listing 11.1: fetching MSFT OHLC data from Yahoo

InteractiveBrokers

Trivial - that describes it best. Why? Thanks to proper software architecture. The following code fragment shows how to use the IB quote source to fetch historical quotes from your IB connection. It is left to you to imagine how to fetch ticks or OHLC data sets.

Listing 11.2: fetching historical MSFT quotes from IB

¹see <http://jdoc.activequant.org/org/activequant/data/retrieval/ICandleSeriesSource.html>

Chapter 12

Logical building blocks

ActiveQuant is built out of several building blocks and modules, each fitting a certain purpose.

12.1 Trade System

A trade system is a code part that is ran in a trade system container. It does not necessarily have to trade, it can also passively track or compute things. Thinking of it as a plugin for the trade system container could help.

Annotations can be used to configure and tag certain parts of a trade system. For example, a "start" annotation declares a function in a tradesystem as a method that needs to be ran after trade system initialization.

12.2 Broker interfaces

12.2.1 Theory

The broker interfaces provide functionality to access your broker or exchange account through a variety of means. Mind that AQ has a defined java interface for all broker interface implementations, which makes it a seamless act to move from simulation, via the paper broker, to production via the IB broker implementation. All that is needed to exchange the used broker in the framework is a change in the spring configuration file.

By the time of this writing, the Interactive Broker interface is very well fleshed out and the FixML is less complete. The FixML implementation can be considered as a starting point to implement own interfaces.

In order to synchronize an account object, which includes a portfolio, former orders and executions as well as an account balance, the AccountManagingBrokerProxy can be wrapped around a broker interface. Unless this is done, the broker interface acts as a very raw and direct access class to the broker.

Note: It is not recommended to directly use the broker interface implementations, as functionality contained in the AccountManagingBrokerProxy is lost. Better always wrap the account managing broker proxy around some broker implementation.

The life of an order ...

Orders pass through several states in their life time as an object instance. The corresponding broker implementations follow well defined interfaces of the *IBroker* interface. First of all, an order has to be instantiated. Second, the order has to be prepared by calling the corresponding function of any broker implementation. The result is an *OrderTracker* instance, that typically contains a reference to the broker and functions for more information about the order's state and other functionality. The order tracker also has an order event source to which interested parts of an application can subscribe to. Once all is set up, the order can get submitted to the broker through the tracker's submit function from where on information about order events will flow back to the tracker through the event source.

The following listing provides an example of how to set up an order. The mentioned `longLimitOrder(..)` is a helper function that does nothing else than instantiating a new order object and filling it with some ready made default values.

Listing 12.1: set up and submit orders

```
Order o = longLimitOrder(spec, limit, positionDifference);
IOrderTracker t = algoEnv.getBroker().prepareOrder(o);
t.getOrderEventSource().addListener(new IEventListener<OrderEvent>()
{
    @Override
    public void eventFired(OrderEvent event) {
        if (event instanceof OrderExecutionEvent) {
            // log.info("Execution at " + ((
                OrderExecutionEvent) event).getPrice());
        }
    }
});
// good practice to store the order trackers in a hashmap
orderTrackers.put(o, t);
t.submit();
```

Account Managing Broker Proxy

The AMBP keeps track of net positions, cash, etc. and stores these value for you to access. All other broker implementations do not, but just emit events.

Paper Broker

The paper broker implementation is good for testing trading systems. It will accept orders and emit executions and fills, it supports various order types like stop, limit, market and trailing stop orders, which it executes after a configurable delay. Especially the last point, configurable delay, can make simulated trading and backtests very realistic.

Interactive Brokers

The IBBroker implementation follows of course as well AQ's IBroker interface. It is AQ's best implemented broker module.

12.2.2 Practice

Note these words of warning, don't do this with your live account unless you know what you do. As usual, the authors reject all liability for everything, even for your dog's short tail.

For the sake of usability, most examples will use the IB broker interface implementation. In order to use another broker implementation, the platform offers several approaches. Depending on the chosen configuration approach, it can be as simple as changing a bean definition in Spring to recompiling source code.

Placing orders

An example of how to send orders is listed at `ref:sendOrders`.

Part IV

P2

Chapter 13

Introduction

P2 - the flagship application suite. Just take the following sections as they are and make up your mind about them.

Chapter 14

Processes

Chapter 15

Tools

15.1 Config Persister

The class ConfigPersister can be used to generate a simulation config or algo env config and to save it to disk. All parameters can be set in the java code of this class, it's length of about 30 lines makes it easy to understand and use.

It is a very small and short utility class that uses the persistence layer to save it somewhere.

15.2 Data Relay

The data relay is a generic tool to relay quotes received on a TCP socket in ASCII/CSV format to a JMS channel. It is very handy when you have a non java data source and want to make the quotes available over JMS. Of course, this relaying introduces a little bit of latency, usually in the dimension of some milliseconds.

By default it listens on TCP port 22223 for incoming connections and reads then line wise from the socket.

The data relay needs to be linked to the activequant master database. Whenever an instrument specification arrives, it will try to resolve an instrument specification instance from the activequant master database. If a matching instrument cannot be found, it will generate entries for new instrument specifications.

15.2.1 Protocol used

The data relay can accept ticks and quotes. The first character (Q or T) indicates the type of the transmitted line.

15.2.2 Instrument specification

Instrument specification are separated by commas instead of semicolons. They are nested inside Quotes or TradeIndication lines.

Spec information must have the format:

<InstrumentName>, <Exchange>, <Currency>, <Vendor>

15.2.3 Quote lines

CSV lines must have the format:

Q;<NanoSecTimeStamp>;<InstrumentSpecification>;<BidPrice>;<BidVolume>;<AskPrice>;<AskVolume>;

15.2.4 Trade Indication lines

CSV lines must have the format:

Spec information must have the format:

T;<NanoSecTimeStamp>;<InstrumentSpecification>;<Price>;<Volume>;

15.2.5 Using and starting it

The DataRelay can be started through the runDataRelay.sh bash script, after a successful compilation.

```
sh runDataRelay.sh
```

15.3 Random Data Generator

The random data generator is a generic tool to generate random quotes for three instruments. All quotes are published in the AQ specific format to the JMS connection configured in the spring configuration files (see src/main/resources/inmemoryalgorithmruntime.xml). The three instruments are called Random1, Random2, Random3. Upon start, the application will load these instruments from a database and if not present, these will be generated and saved in the database.

Quotes are emitted at intervals of $(\text{rnd}() * 1000 + 10)$ milliseconds, so, somewhere between 0.01 and 1.01 seconds between quotes.

A plain start script for the random data generator is shipped in the main folder of p2.

15.4 Quote Recorder

Mike's power horse

Quote recorder records data in the new archive format into a target folder. It loads through the instrument specification dao, the complete list of specs and compares the vendor of all specs against the vendor identifier, which can be set either in the spring configuration file or it can be left blank, in which case all loaded specs will be recorded. Per instrument id, one folder will be created, in which per day a subfolder is created. In there a file quotes.csv contains recorded quotes. Data is appended to the end of the file.

The following xml snippets are from the quoterecorder.xml configuration file.

By default it uses the JMS data source:

```
<bean id="quoteSourceJms" class="org.activequant.util.tempjms.JMSQuoteSubscriptionSource">
    <constructor-arg><value>83.169.9.78</value></constructor-arg>
    <constructor-arg><value>7676</value></constructor-arg>
</bean>
<!-- main class -->
<bean id="context" class="org.activequant.util.QuoteRecorder"
    init-method="record">
    <property name="source" ref="quoteSourceJms" />
    <property name="dao" ref="quoteArchiveDao" />
    <property name="specDao" ref="specificationDao" />
</bean>
```

In order to use IB, you have to replace the above snippet with:

TO BE ADDED.

To record instruments from a specific vendor only (that's matched against the instrument specifications from the master database) modify quoterecorder.xml as shown below. Only specs that match are considered for subscribing. Usually it's null, so it will record everything.

```
<!-- main class -->
<bean id="context" class="org.activequant.util.QuoteRecorder"
    init-method="record">
    <property name="source" ref="quoteSourceJms" />
    <property name="dao" ref="quoteArchiveDao" />
    <property name="specDao" ref="specificationDao" />
    <property name="vendorIdentifier" value="IB" />
</bean>
```

Starting the QuoteRecorder can happen through the start script in the base folder of P2:

```
sh runQuoteRecorder.sh
```

The quote recorder saves all quotes to a specific target folder, which can be specified in the corresponding spring configuration file, in the quote recorder dao section:

```
<bean id="quoteArchiveDao" class="org.activequant.util.RecorderQuoteDao">
    <constructor-arg>
    <value>/home/share/archive</value>
    </constructor-arg>
</bean>
```

The good thing is that the modularity of spring and the way ActiveQuant is written, allows you to plug in any other QuoteDao, you can easily implement your own recorder to save to a specific custom file format or even to FTP if you like. A database recorder dao is shipped in ActiveQuant. By default, the quote recorder will use the HibernateQuoteDao, which will record everything to the configured database. Mind that you will have to recompile and assemble the app in case you want to use another database file, unless you edit the startup script to include the property files from some specific other place (Note: different start script and more documentation appreciated).

Part V

Pushing the frontier further

Chapter 16

AQ + R = AQ-R

UNDER CONSTRUCTION

The open source S-clone R has become very popular over the years. It's open sourcedness and scriptability have made it a real success story in many cases. A special subproject of the AQ project provides access functions for R, so that an AQ database can be used as a fundament for research. In this section we will have a look at the AQ/R bridge, examples will show how to get most out of an AQ database with R.

16.1 Requirements for using AQ and R

In order to use the AQ/R bridge, several R plugins have to be installed. First of all, the mysql package has to be up and running. The most common package for this is RMySQL¹.

16.2 Functionality by use cases

16.2.1 Loading quotes from an AQ DB

To be written.

¹see <http://biostat.mc.vanderbilt.edu/wiki/Main/RMySQL>