# HSBC Interview Coding Exercise

Shu Ge

## Section 1. Requirement of the Coding Exercise and Design Consideration

The requirement of this coding exercise is to implement a server that responds to quote requests for market making purposes.

1. **Concurrency**

   The quote server is required to be capable of handling multiple clients in parallel. Under our design, we use one thread to handle each client. The maximum number of clients handled concurrently is configured via the configuration entry *engine.threads*. To achieve better performance, we use the thread pool to host all the worker threads.

   Potential improvement over this synchronous design is to deploy more advanced techniques like asynchronous I/O programming.

   Under real scenarios, we need to consider those idle clients, namely those with socket still connected but no longer sending any messages. We take care of this through a time out value on the socket read operation. The maximum idle time is configurable via the configuration entry *engine. client_max_idle_time*.

2. **Market Data Update**

   For this coding exercise, we are not able to get real time market data update.

   Under our design, the HSBCReferencePriceSource object polls for changes on file `dat/market_data_file`. Any incremental changes are placed into the file `market_data_file`, which represent market data changes.

   Also, the only market data considered for any security is price.

3. **Execution of calculateQuotePrice**

   We notice this requirement "This may take a long time to execute". For some securities, this is probably related with manual intervention on quote request.

   Under our design, we choose to auto respond to those requests covered in our security table (namely with a market price), or respond with NA if not. Whether this process takes 1 second or 10 minutes does not matter due to our design, i.e. not blocking other client's conversation.

4. **BUY/SELL Quote Calculation**

   We use the following simple formula to calculate the quote:

   buy_quote = referencePrice * (1 + buy_spread) * quantity;
   sell_quote = referencePrice * (1 - sell_spread) * quantity;

   Separate spreads are used for buy/sell quotes. Different spreads are used for different securities, which reflect the hardness of borrowing a particular security.

5. **Interface ReferencePriceSourceListener**

   We think it would be more appropriate to design this interface as a call back method within class ReferencePriceSource instead of a standalone class.

   Here is the justification: It is more intuitive to put the security price storage inside class ReferencePriceSource. If so, the coupling between ReferencePriceSource and ReferencePriceSourceListener is too tight, i.e. pointing to each other. After careful consideration, we put the security price storage inside ReferencePriceSourceListener. We do wrapper call on get method of ReferencePriceSource.

6. **Interface referencePriceChanged**

   We added one more price change prototype:

   ```
   referencePriceChanged(List<Tuple<Integer, Double>> list)
   ```

   This addition makes sense. Exchanges could send out multiple price updates in one shot. Updates to price data structure involve synchronization, which is costly. This additional interface supports bulk updates within one lock operation.

## Section 2. Compilation & Testing Server

We create a Microsoft Azure instance for this coding exercise. Both the compilation and testing are performed on this server.

|  |  |
|---|---|
| **Server IP**: | 23.97.78.99 |
| **User**: | geshu |
| **Password**: | 12345XScheme |

The following is a quick guide to launch the quote request server for testing:

1. **Launching 2 SSH sessions to the server above**

2. **In Session 1, starting the Quote Request Server**

   ```
   [geshu@java ~]$ pwd
   /home/geshu
   [geshu@java ~]$ java -cp /home/geshu/src/ com.example.marketmaker.HSBCQuoteEngine
   ```

3. **In Session 2, starting the automated script**

   ```
   [geshu@java ~]$ pwd
   /home/geshu
   [geshu@java ~]$ bin/script_test.php
   ```

## Section 3. Folder Structure & Compilation

The file structures are presented as follows:

```
[geshu@java ~]$ pwd
/home/geshu
[geshu@java ~]$ ls -l
total 16
drwxrwxr-x. 2 geshu geshu 4096 Jan  9 13:50 bin
drwxrwxr-x. 2 geshu geshu 4096 Jan  9 08:14 cfg
drwxrwxr-x. 2 geshu geshu 4096 Jan  9 07:25 dat
drwxrwxr-x. 3 geshu geshu 4096 Jan  9 15:14 src
```

| Path | Files |
|---|---|
| `bin` | Automated testing script. |
| `cfg` | The configuration file. |
| `dat` | Market data files, spread file. |
| `src` | All the source codes. |

**Table 1**. File Structure

The compilation steps are as follows:

```
[geshu@java ~]$ pwd
/home/geshu
[geshu@java ~]$ cd src/com/example/marketmaker/
[geshu@java marketmaker]$ javac -classpath /home/geshu/src/ HSBCQuoteEngine.java
```

## Section 4. Development Item Testing

The following testing cases are considered as unit testing, performed along with the development.

1. **Starting the Quote Request Server**

```
[geshu@java ~]$ pwd
/home/geshu
[geshu@java ~]$ java -cp /home/geshu/src/
com.example.marketmaker.HSBCQuoteEngine
HSBCQuoteEngine: initialize engine
HSBCQuoteEngine: initialize spread tables
initializeSpreadTable: loading 700,0.0005,0.00025
initializeSpreadTable: loading 2828,0.0005,0.00025
initializeSpreadTable: loading 2800,0.0005,0.00025
initializeSpreadTable: loading 678,0.01,0.0005
HSBCQuoteEngine: start to run engine
updating 700 195.1
updating 678 2.35
updating 1128 12.66
updating 1071 3.38
updating 2828 96.85
updating 2800 22.65
```

2. **Max Number of Clients**

   We use the Linux programme *telnet* as the client to connect to our Quote Request server. We set the configuration entry *engine.threads* as 2 in our configuration. Hence, we expect the server accepts a maximum of 2 clients in parallel.

   Firstly, we start 2 telnet sessions to connect to the server. The third telnet session is expected not to be accepted by the server. The testing evidence matches our expectation.

```
[geshu@java ~]$ telnet localhost 8888
Trying ::1...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host. (printed out immediately)
[geshu@java ~]$
```

3. **Idle Client Removal**

   We set the configuration entry as *engine.client_max_idle_time=10*. Hence, we expect the server removes clients idle for more than 10 seconds.

   We use *telnet* to connect to the serer. If we do not send any requests, the session is expected to be closed by the server after 10 seconds. The testing evidence matches our expectation.

```
[geshu@java marketmaker]$ telnet localhost 8888
Trying ::1...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host. (printed out after 10 seconds)
```

4. **BUY/SELL Spread Loading**

   This test involves checking the spreads at the server start (as indicated in **Point 1**).

5. **Market Data Loading/Update**

   The testing involves manually updating the file `dat/market_data_file` after the quote request server being started. When we change the file `market_data_file`, we should see the server immediately print something in screen similar to the following:

```
updating 2828 96.85
updating 2800 22.65
```

   This means the prices have just been reflected into the internal data structure.

6. **Quote Result Validation**

   This is the core part of the quote request engine. We manually verify the result for the following cases:

1) Buy/sell quote for security using default spread;
2) Buy/sell quote for security covered by spread file;
3) Buy/sell quote for security not with a reference price.

## Section 5. Testing

Our testing includes both manual testing and automated testing. All the testing is successful.

### 1. Manual Testing

We use *telnet* to connect to the Quote Request server, and send the manual commands directly to the server. The screenshot is as follows:

```
[geshu@java ~]$ telnet localhost 8888
Trying ::1...
Connected to localhost.
Escape character is '^]'.
700 BUY 100 (manually typed + PRESS ENTER)
19519.75 (server response)
700 SELL 100 (manually typed + PRESS ENTER)
19505.12 (server response)
```

### 2. Automated Testing

Our favorite scripting language is PHP. Hence, we write a simple script in PHP to automatically send massive quote requests to the Quote Server. The script is attached in the zip file.

The script randomly generates a quote request, based on security codes provided (bad security included as well). It then send the request to the quote request server, and read back the quote from the socket.

The screenshot is as follows:

```
[geshu@java ~]$ bin/script_test.php
REQUEST: 2800 BUY 800; QUOTE: 18129.06

REQUEST: 1128 SELL 800; QUOTE: 10122.94

REQUEST: 2800 BUY 600; QUOTE: 13596.79

REQUEST: 1071 BUY 800; QUOTE: 2706.70

REQUEST: 1071 SELL 1000; QUOTE: 3378.31

REQUEST: 2388 BUY 200; QUOTE: NA
```

## Section 6. Further Improvement

As this is a coding exercise, we do not spend major efforts on the following items. We believe these are not the key points for examination purposes. Anyway, these points could be solved after some research via google.

1. **System Logging**

   All the logs are sent to the standard output Systemt.out. Also, there is no synchronization considered on outputs from different threads. For any production quality software, output could be written to different log files for a simple design.

2. **Exception Handling**

   We put efforts on those application logic related exceptions, e.g. InvalidSecurityException, socket related.

   We do not provide high quality handlers for some rare exceptions, e.g. I/O Exception. These exceptions are only captured by the default Exception case.

3. **Software Packaging**

   We are not very familiar on the java software packaging. Hence, we do not create a JAR file to package all the java classes. Neither, we create a Make file to automate the compilation.