Concordia University

Department of Computer Science and Software Engineering

COMP 6231: Distributed Systems Design

STOCK BROKER SYSTEM: PROJECT MILESTONE #3

Team 3

Gay Hazan
Ross Smith
Patrick Cristofaro

July 28, 2015

Table of Contents

Overview

This project implements a stock exchange service, including a broker, a number of businesses, and an arbitrary number of clients.

"Stockbrokers receive requests from clients, buy/sell stocks on various stock exchanges. The application will have to query a real external stock quotes service, e.g., such as that of Yahoo, and simulate trades locally." [Mokhov]

The third project milestone re-factors the previous CORBA implementation into a web services architecture. Our team has implemented both SOAP and RESTful web services, as shown below in Illustration 1.

Architecture

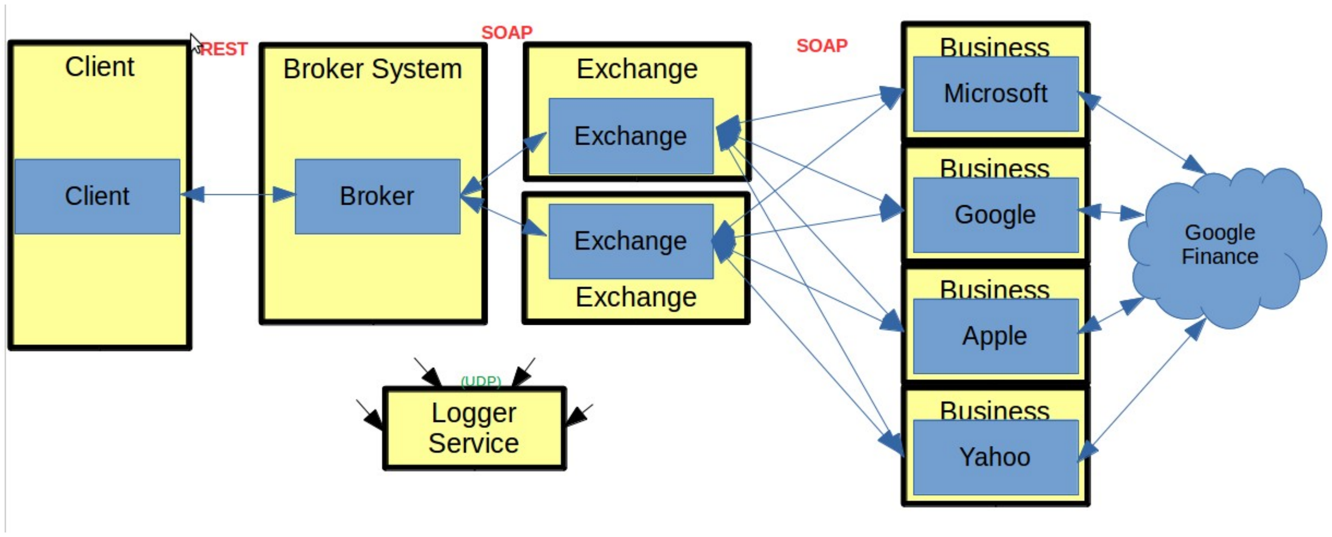


Illustration 1: Architecture of the Stock Exchange, adapted from [Mokhov]

The system is based on four different server types, all running concurrently.

Client Server initiates orders and purchase shares by connecting to the broker. The system must support an arbitrary number of clients. For our purposes, JUnit tests simulate clients.

Stock Broker implements the **broker** class. For this project milestone, we have separated the broker and exchange servers.

Exchange Servers implement the **exchange** class. Two exchanges must run concurrently for this milestone.

Business servers represent individual companies with shares to sell. The system must support an arbitrary number of businesses, and retrieve stock prices from an internet web service.

Logger server collects information and error messages, saving them locally in a log file.

All communications between servers must use a **Web Service** implementation, with the exception of the logger, which shall use **UDP**.

Design

In this section, we will take a more detailed look at each architectural component, including the rationale behind our design decisions. We will also address some of the overall improvements made since the previous iteration.

The main focus of the discussion will be on new implementation relevant to this milestone.

Business component

The business component is implemented using JAX-WS and SOAP. Only the methods used by the Exchange are implemented within the interface.

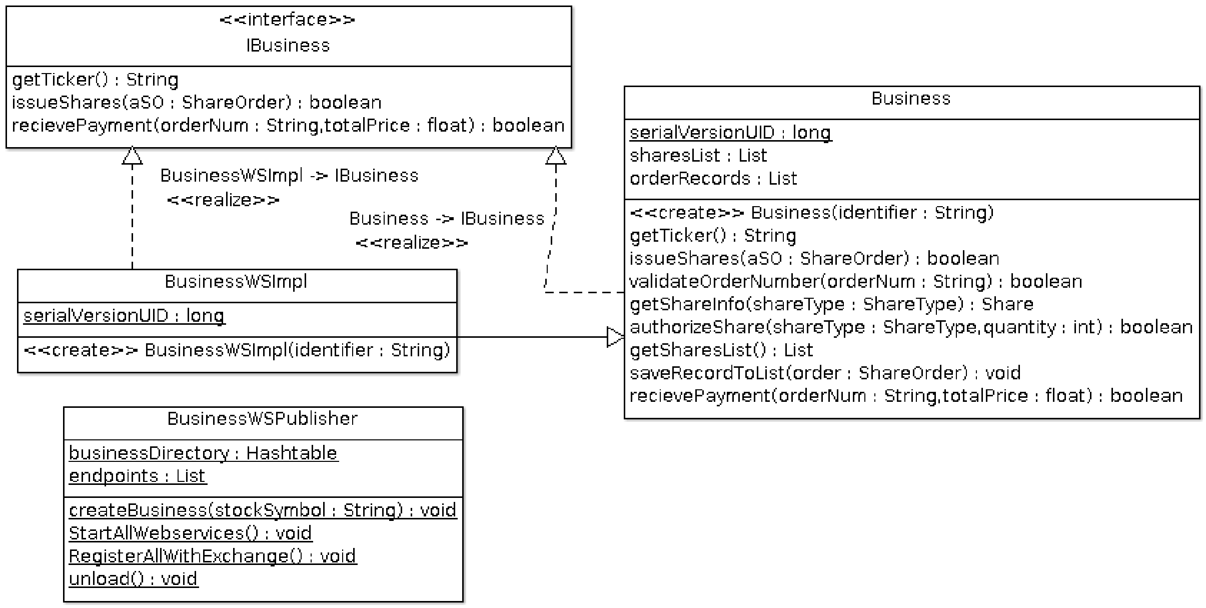

<<interface>>
IBusiness
getTicker() : String
issueShares(aSO : ShareOrder) : boolean
recievePayment(orderNum : String,totalPrice : float) : boolean
BusinessWSImpl -> IBusiness
<<realize>>
Business -> IBusiness
<<realize>>
Business
serialVersionUID : long
sharesList : List
orderRecords : List
<<create>> Business(identifier : String)
getTicker() : String
issueShares(aSO : ShareOrder) : boolean
validateOrderNumber(orderNum : String) : boolean
getShareInfo(shareType : ShareType) : Share
authorizeShare(shareType : ShareType,quantity : int) : boolean
getSharesList() : List
saveRecordToList(order : ShareOrder) : void
recievePayment(orderNum : String,totalPrice : float) : boolean
BusinessWSImpl
serialVersionUID : long
<<create>> BusinessWSImpl(identifier : String)
BusinessWSPublisher
businessDirectory : Hashtable
endpoints : List
createBusiness(stockSymbol : String) : void
StartAllWebservices() : void
RegisterAllWithExchange() : void
unload() : void

Illustration 2: Classes and methods of the business component

The BusinessWSImplService class, which is auto-generated by JAX-WS, has been modified to support multiple web servers. The constructor requires one parameter, the stock symbol, and automatically determines the correct endpoint and WSDL for that business. In this way, a single implementation can support unlimited instances of the Business class.

BusinessWSPublisher is a convenience class that allows businesses to be launched easily and without replicating code. Calling createBusiness() adds a business to a list of potential servers. Once the list is fully populated, calling StartAllWebservers() will instantiate endpoints for each business. Finally, calling RegisterAllWithExchange() will register all businesses on the exchange(s).

Exchange component

The exchange is one of the more complex components of the software system. It must keep a record of all businesses that can sell shares, it must keep a local cache of shares that are available for sale, and it must respond to purchase orders from the broker.

The exchange work flow is as follows:

- The exchange server starts up.
- A business registers with the exchange. The exchange asks the business to issue some shares. These shares have not been purchased yet; the exchange will hold them locally in a cache. The business keeps a record of this "debt" for later payment.
- The broker asks to buy some shares.
  - The exchange sells its cached shares first. **This allows for the fastest transaction time**.
  - If the exchange doesn't have enough shares, it will ask the business for more.
- Once the transaction is complete, the exchange will replenish the local cache, as required.
- When enough shares have been sold to pay back the business, the exchange will issue a payment to the business, who will record that the debt is now paid.

For this iteration, the Exchange communicates using both SOAP and RESTful web services as a client (for Businesses and the Broker, respectively). In addition, the Exchange itself implements a SOAP web service.

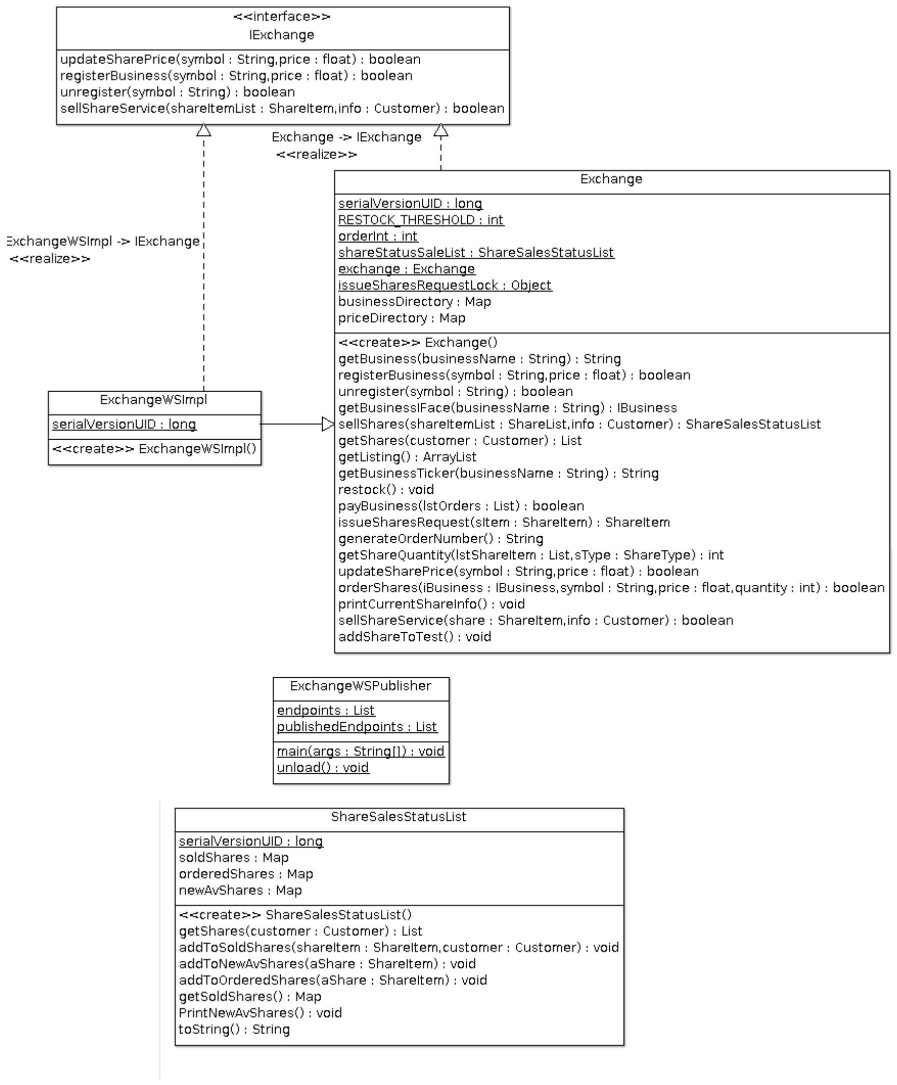

<<interface>>
IExchange
updateSharePrice(symbol : String,price : float) : boolean
registerBusiness(symbol : String,price : float) : boolean
unregister(symbol : String) : boolean
sellShareService(shareItemList : ShareItem,info : Customer) : boolean
Exchange -> IExchange
<<realize>>
ExchangeWSImpl -> IExchange
<<realize>>
Exchange
serialVersionUID : long
RESTOCK_THRESHOLD : int
orderInt : int
shareStatusSaleList : ShareSalesStatusList
exchange : Exchange
issueSharesRequestLock : Object
businessDirectory : Map
priceDirectory : Map
<<create>> Exchange()
getBusiness(businessName : String) : String
registerBusiness(symbol : String,price : float) : boolean
unregister(symbol : String) : boolean
getBusinessIFace(businessName : String) : IBusiness
sellShares(shareItemList : ShareList,info : Customer) : ShareSalesStatusList
getShares(customer : Customer) : List
getListing() : ArrayList
getBusinessTicker(businessName : String) : String
restock() : void
payBusiness(lstOrders : List) : boolean
issueSharesRequest(sItem : ShareItem) : ShareItem
generateOrderNumber() : String
getShareQuantity(lstShareItem : List,sType : ShareType) : int
updateSharePrice(symbol : String,price : float) : boolean
orderShares(iBusiness : IBusiness,symbol : String,price : float,quantity : int) : boolean
printCurrentShareInfo() : void
sellShareService(share : ShareItem,info : Customer) : boolean
addShareToTest() : void
ExchangeWSImpl
serialVersionUID : long
<<create>> ExchangeWSImpl()
ExchangeWSPublisher
endpoints : List
publishedEndpoints : List
main(args : String[]) : void
unload() : void
ShareSalesStatusList
serialVersionUID : long
soldShares : Map
orderedShares : Map
newAvShares : Map
<<create>> ShareSalesStatusList()
getShares(customer : Customer) : List
addToSoldShares(shareItem : ShareItem,customer : Customer) : void
addToNewAvShares(aShare : ShareItem) : void
addToOrderedShares(aShare : ShareItem) : void
getSoldShares() : Map
PrintNewAvShares() : void
toString() : String

Illustration 3: Main classes of the Exchange

Broker component

The original Broker from milestone #1 is virtually unchanged, but a new class (BrokerREST.java) is added support RESTful web services.

The RESTful broker sits on top of Java servlets, with no other middle-ware used. All complex data types are marshaled and de-marshaled using JSON.

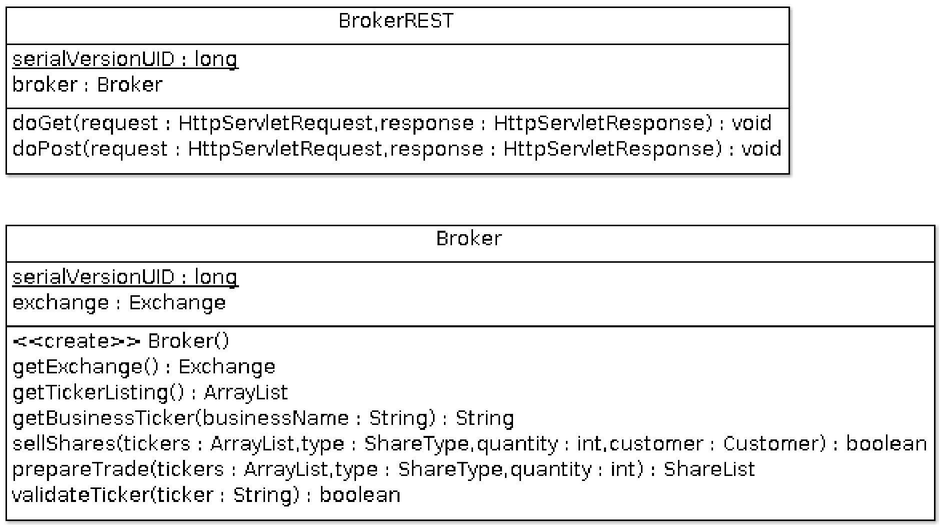

BrokerREST
serialVersionUID : long
broker : Broker
doGet(request : HttpServletRequest,response : HttpServletResponse) : void
doPost(request : HttpServletRequest,response : HttpServletResponse) : void
Broker
serialVersionUID : long
exchange : Exchange
<<create>> Broker()
getExchange() : Exchange
getTickerListing() : ArrayList
getBusinessTicker(businessName : String) : String
sellShares(tickers : ArrayList,type : ShareType,quantity : int,customer : Customer) : boolean
prepareTrade(tickers : ArrayList,type : ShareType,quantity : int) : ShareList
validateTicker(ticker : String) : boolean

Illustration 4: The classes of the Broker component

Logger component

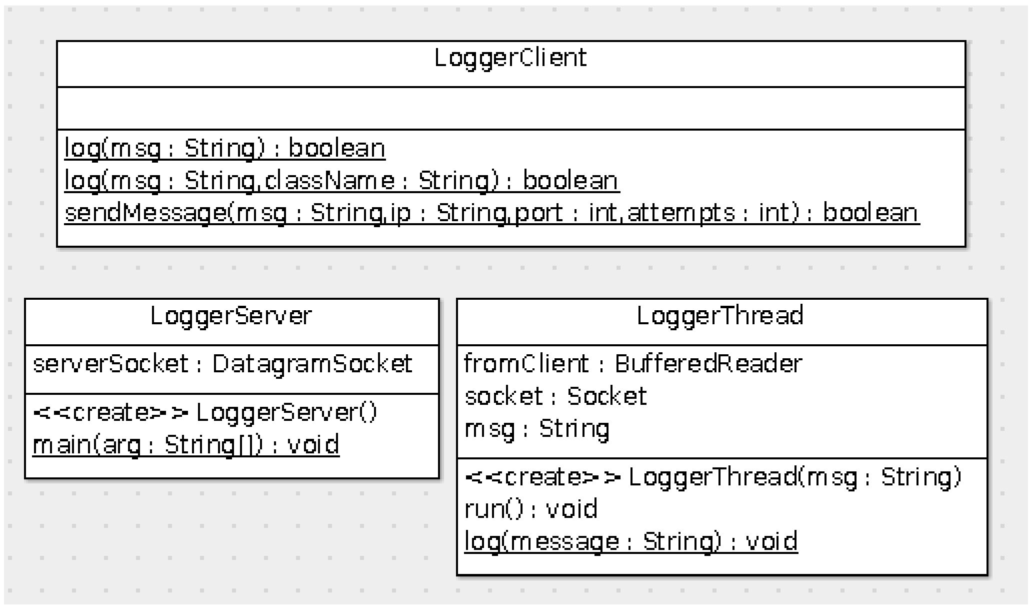

LoggerClient
log(msg : String) : boolean
log(msg : String,className : String) : boolean
sendMessage(msg : String,ip : String,port : int,attempts : int) : boolean
LoggerServer
serverSocket : DatagramSocket
<<create>> LoggerServer()
main(arg : String[]) : void
LoggerThread
fromClient : BufferedReader
socket : Socket
msg : String
<<create>> LoggerThread(msg : String)
run() : void
log(message : String) : void

Illustration 5: Classes that comprise the Logger

The logger component is virtually unchanged from the previous iteration, except for a small performance improvement regarding synchronization. It uses UDP to accept messages from all other system components, and saves them to file.

The logger can easily be run on any server with a known IP address. The user simply needs to update the Config.json file with the correct information.

Improvements

Our second iteration improved significantly on some weaker aspects of our design. For the third iteration, we continued to address issues of quality and performance, as outlined in this section.

LoggerThread should not lock LoggerThread.class

This issue has been resolved by creating a lock object only for the file write operation.

GenerateOrderNumber in Exchange uses method synchronization

The lock has been changed to a class-level lock.

Get rid of 33 warnings

The warnings that were present in the previous milestone have all been removed.

Show start and end time for MultiThread test

A timestamp has been added to the start and end of this test (displayed in the console window).

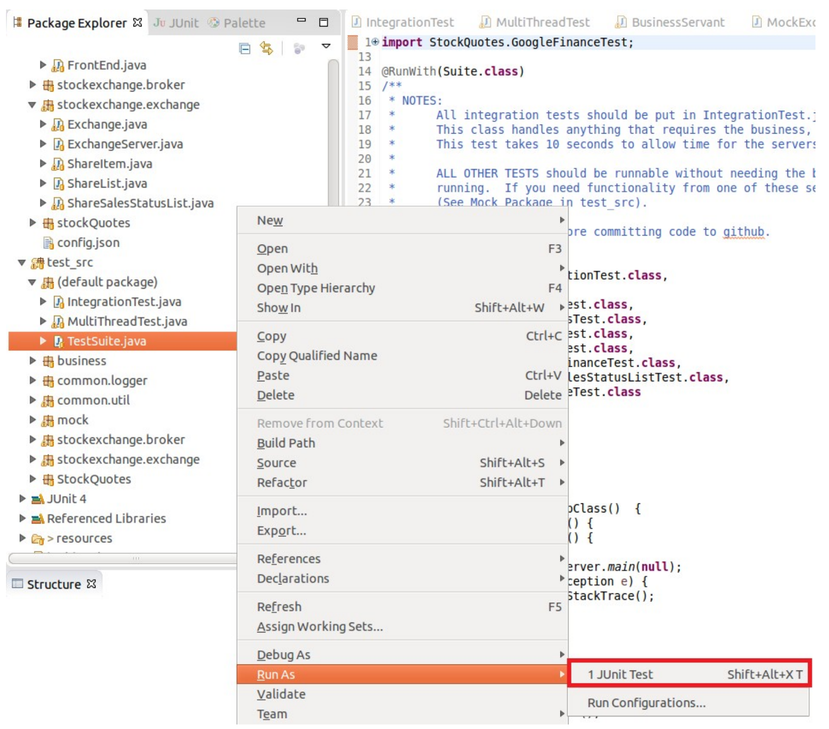
Testing

This section will outline the tests we have included in the project submission. All tests use JUnit4. For all tests, the user must ensure that Tomcat is running locally with the Broker service correctly set up, as outlined in the README.md file.

TestSuite.java

The TestSuite JUnit conveniently sets up all the servers and runs all other JUnit tests. Simply launch it from Eclipse and observe the output on the console, or in the *files/iteration_3.log* file.

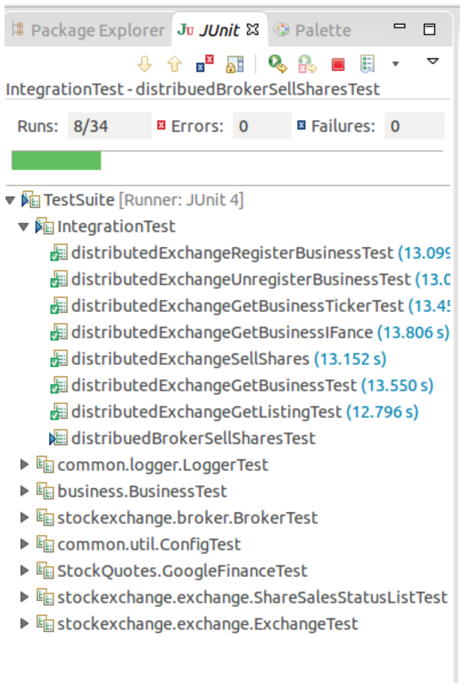
The TestSuite includes all integration and unit tests, but not the stress test.



```
import StockQuotes.GoogleFinanceTest;

@RunWith(Suite.class)
/**
 * NOTES:
 *      All integration tests should be put in IntegrationTest.
 *      This class handles anything that requires the business,
 *      This test takes 10 seconds to allow time for the server
 *
 *      ALL OTHER TESTS should be runnable without needing the
 *      running.  If you need functionality from one of these se
 *      (See Mock Package in test_src).
```

```
ore committing code to github.

tionTest.class,

est.class,
sTest.class,
est.class,
est.class,
inanceTest.class,
lesStatusListTest.class,
eTest.class

Class()  {
() {
() {

erver.main(null);
ception e) {
StackTrace();
```

Illustration 6: Running the test suite



```
IntegrationTest - distribuedBrokerSellSharesTest

Runs:  8/34     Errors:  0     Failures:  0

TestSuite [Runner: JUnit 4]
  IntegrationTest
    distributedExchangeRegisterBusinessTest (13.09
    distributedExchangeUnregisterBusinessTest (13.0
    distributedExchangeGetBusinessTickerTest (13.4
    distributedExchangeGetBusinessIFance (13.806 s)
    distributedExchangeSellShares (13.152 s)
    distributedExchangeGetBusinessTest (13.550 s)
    distributedExchangeGetListingTest (12.796 s)
    distribuedBrokerSellSharesTest
  common.logger.LoggerTest
  business.BusinessTest
  stockexchange.broker.BrokerTest
  common.util.ConfigTest
  StockQuotes.GoogleFinanceTest
  stockexchange.exchange.ShareSalesStatusListTest
  stockexchange.exchange.ExchangeTest
```

Illustration 7: Test suite in progress

MultiThreadTest.java

Concurrency is a critical feature of distributed systems. To test concurrency in our software package, we include the MultiThreadTest JUnit. The test launches four businesses, fifteen clients and tries to issue 90,000 purchase orders per client for a total of 1.35 million transactions.

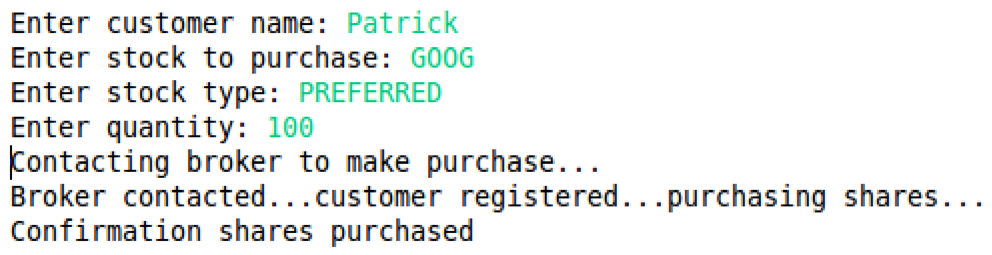
The orders are random, and some are intentionally invalid. The system must respond to the stress resiliently and recover from invalid orders cleanly.

To run the test, simply launch it the same way as TestSuite.java.

FrontEnd package

The FrontEnd acts as a client interface, connecting to the Broker web service and simulating the experience of using the entire system as a customer.

Simply run FrontEnd.java and use the console to interact with the system. Valid stock symbols are GOOG, MSFT, AAPL and YHOO.

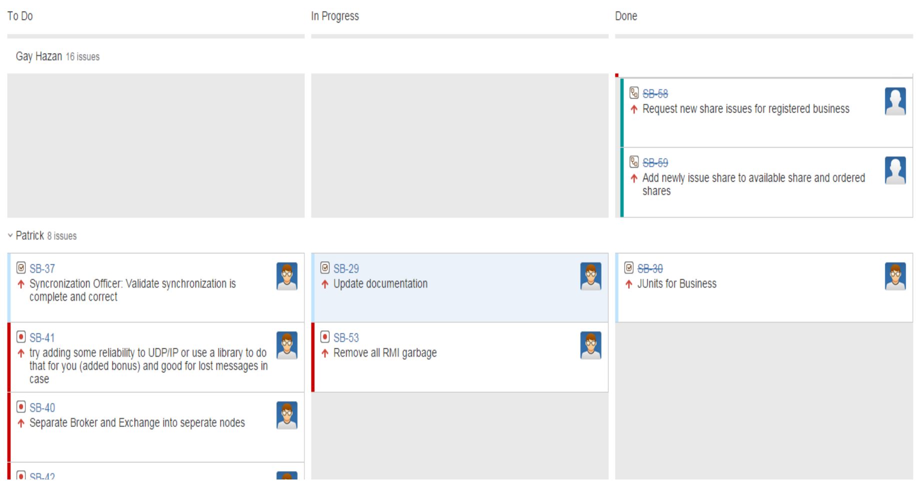


```
Enter customer name: Patrick
Enter stock to purchase: GOOG
Enter stock type: PREFERRED
Enter quantity: 100
Contacting broker to make purchase...
Broker contacted...customer registered...purchasing shares...
Confirmation shares purchased
```

Illustration 8: Using the FrontEnd client interface

Management

For this milestone, we continued to use JIRA Agile as our primary tool for coordinating tasks between team members. The server is located at http://jira.homelinux.org. Access can be provided on request.



To Do	In Progress	Done
Gay Hazan 16 issues		
		~~SB-58~~ Request new share issues for registered business
		~~SB-59~~ Add newly issue share to available share and ordered shares
Patrick 8 issues		
SB-37 Syncronization Officer: Validate synchronization is complete and correct	SB-29 Update documentation	~~SB-30~~ JUnits for Business
SB-41 try adding some reliability to UDP/IP or use a library to do that for you (added bonus) and good for lost messages in case	SB-53 Remove all RMI garbage	
SB-40 Separate Broker and Exchange into seperate nodes		
SB-42		

Illustration 9: JIRA Agile continues to be the project's task-management system

Release Notes

Please see the README.md in the project root folder for important details about this release, including basic instructions on running and links to the detailed Javadoc.

References

[Mokhov]	Mokhov, Serguei A. COMP6231: Distributed Systems Design TENTATIVE PROJECT DESCRIPTION (Revisions 2.2), June 2015.
[Oracle]	http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html, June 27, 2015.
[IBM]	http://www.ibm.com/developerworks/library/ws-restful/
[Nordic]	http://nordicapis.com/rest-vs-soap-nordic-apis-infographic-comparison/
[Oracle2]	http://www.oracle.com/technetwork/articles/java/restful-142517.html
[JavaT]	http://www.javatpoint.com/soap-vs-rest-web-services
[Oracle3]	http://www.oracle.com/technetwork/java/tutorial-138750.html

Appendix A: Response to Questions

Question: When should REST be used, and when is SOAP better? What are the pros and cons of each?

Although both technologies concern web services, REST and SOAP are dramatically different. REST is an architectural style, while SOAP is a protocol [JavaT].

RESTful services are stateless, expose directory structure-like URIs and transfer data in XML, JSON or an arbitrary format [IBM]. Standard HTTP calls such as GET, POST, PUT and DELETE are the core of RESTful web services. The footprint of RESTful web services reduces reliance on middleware which in turn simplifies load balancing. Links can be bookmarkable (in the case of GET requests).

SOAP uses WSDL to define and share what services are available and how to interact with them. It uses XML exclusively.

RESTful services can incorporate SOAP services, but not the other way around [JavaT].

One should use RESTful web services for applications operating on a web environment that does not require access to the data objects themselves. RESTful is also advantageous when bandwidth is limited, such as for mobile applications. [Nordic]

SOAP is preferable when direct access to server objects is needed, or when greater security is necessary. [Nordic] Whenever specific contracts are required between server and client, or capability beyond simple CRUD (Create/Read/Update/Delete) operations is desired, SOAP is the better choice [Oracle2]. This flexibility comes at a cost, however, and SOAP operations have a high overhead bandwidth cost.

REST embraces the open standards of the internet, and if done correctly, has implementations that are relatively intuitive to understand. [IBM] For example, a RESTful GET request to http://www.payments.com/invoicelist/2015/05/18 can be expected, with little thought, to return the list of invoices for May 18, 2015. However, RESTful services may not be the best choice if your server objects are not stateless. A good test would be to consider if your objects would survive a server restart [Oracle2]. If not, they might not be right for REST.

Question: How are webservices implemented in JSP?

A Java servlet is a lot like a Java applet, except instead of being executed by a local user on a machine, it is executed in response to an HTTP request (much like a CGI script). [Oracle3]

Each time a client calls is made to the JSP object, a new thread is launched to run the servlet. The servlet can do virtually everything a standard Java applet can do, such as create objects, talk to databases and print messages. [Oracle3]

To create a servlet, the javax.servlet.Servlet interface must be implemented. In most cases, the desired transport is HTTP, and Java provides the HTTPServlet class that can be extended in order to simplify such an implementation. The HTTPServlet class has a number of methods that can be overwritten to support HTTP operations, such as doGet() or doPost(). [Oracle3]

Once implemented, a typical lifecycle of a servlet would be the following: A user visits a webpage and completes an HTML form. Once they click submit, the form data is POSTed and the Java servlet is invoked. The servlet takes the data, processes it, and produces an HTML output page which is sent back to the client and viewed on his or her local machine. [Oracle3]

Appendix B: Change Log

```
Please refer to GITHUB for the changelog history.
```