Java and Object Oriented Software Engineering
Semester 2 Coursework Laboratory 3
JAgora - An Electronic Stock Exchange

Friday 19th February, 2016

1 Coursework Overview and Working Arrangements

During the course of semester 2 you will work on the development of a single software project. Each laboratory will concentrate on a different aspect of the software development process:

Laboratory	Week	Beginning	Topic	Marks
1	19	25th Jan	Warm up	0
2	21	8th Feb	Problem domain analysis	25
3	23	22nd Feb	Unit and acceptance test harness	25
4	25	8th Mar	Design and implementation	25
5	27	22nd Mar	Applying design patterns	25

During semester 2, we will be using *pair programming* to improve the quality of software produced during the laboratories. You should prepare for your scheduled laboratory as normal, by reading the laboratory sheet problem description and undertaking background research. At the start of the scheduled laboratory, your tutor will pair you with a different student to work with in each of the five JOOSE laboratories. You must work with your assigned partner throughout the scheduled laboratory. Your tutor will monitor how you work as a development team as part of your assessment. Following the laboratory each partner should take a copy of the solution developed thus far and make further independent improvements as desired before submission.
Note: As pair programming is part of the assessment for each laboratory, non-attendance without good cause will result in a loss of all 5 marks available in this category. You must note down the name and matriculation number of your pair programming partner on your submission.

2 Domain Model and Activity Diagram

Consider the UML class diagram of the stock exchange system described in Laboratory 2 (Figure 1). The class diagram shows that a stock exchange comprises a number of different markets (one for each stock), with each market comprising two order books (one for sell and one for buy orders). The order book comprises a list of time stamped orders, sorted by price and time. Each order is associated with a trader.

The class diagram shows an inheritance hierarchy for the different types of order in the system, specialised first by the side of the trade (buy or sell) and then by the order type (limit, market and stop buy). A trade is a relationship class between one buy and one sell order (there may be many trades for each type so that an order can be completely filled). When a trade is executed it is stored as a time stamped trade by the stock exchange.

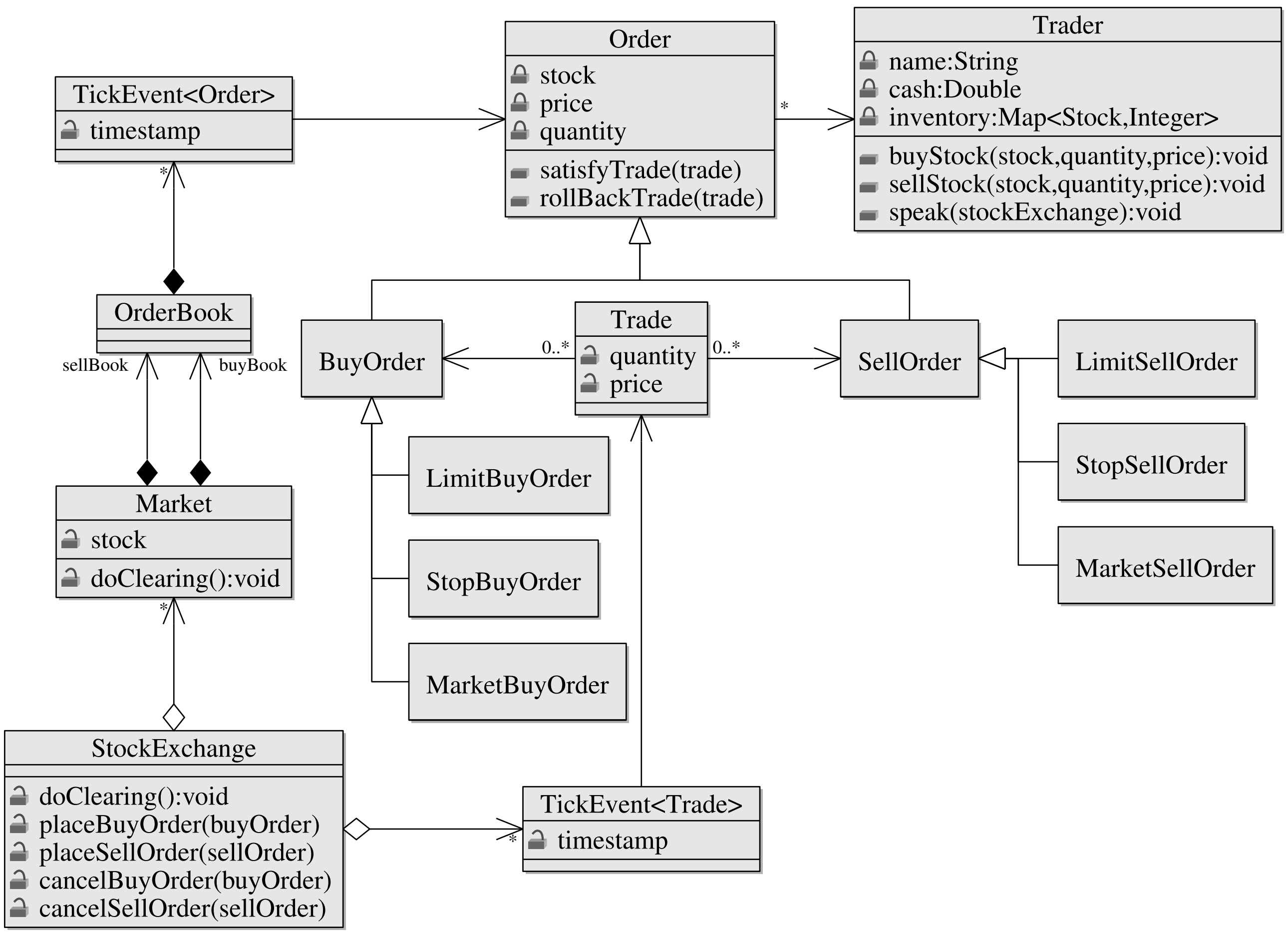

Figure 1: JAgora Class Diagram

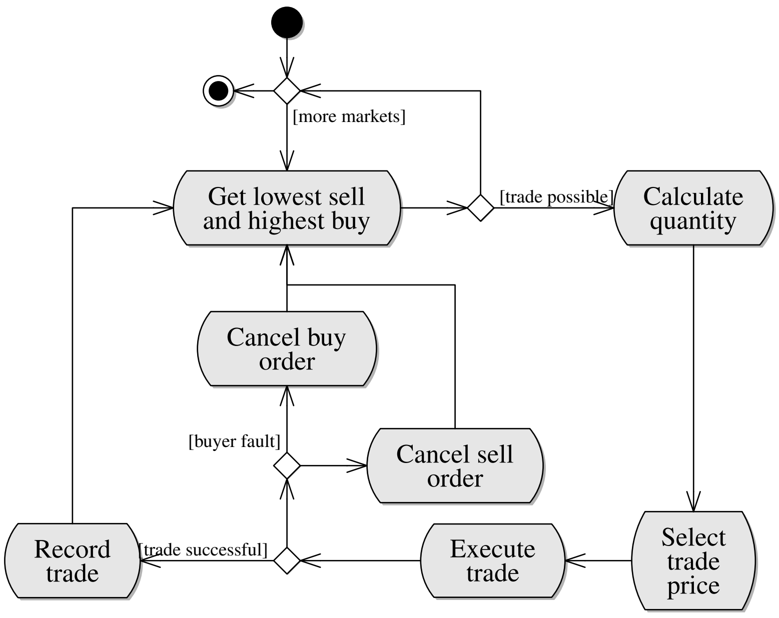

Figure 2: Clearing activity diagram

Separately, the activity diagram in Figure 2 shows the algorithm for clearing trades on a stock exchange comprised of continuous order driven markets. The algorithm performs clearing on each stock market in turn.

First, the lowest sell (best offer) and highest buy (best bid) prices on a market are obtained from the respective order books. If a trade is possible between these two orders then a quantity is calculated for the trade (taking the smallest quantity of the two orders). Next the sell price is selected for the trade and the trade is executed.

If the trade is successful it is recorded in the stock exchange's trade history. If the trade was unsuccessful, the culprit is determined and their order cancelled. In either case, the clearing process will re-check the buy and sell order books to get the new best offer and bids, and attempt to execute the next trade. The clearing process continues in each market until no more trades are possible.

3 Task

You have been provided with a *three* related software projects in the laboratory handout.

- `jagora-api` contains a set of interfaces that define the application programming interface of the JAgora application. Each of the interfaces is accompanied by JavaDoc documentation describing the semantics of the operational signatures. Note that these interfaces are for a design and so vary somewhat from the domain model. *You must not make any modifications to this project.*
- `jagora-impl` contains a set of (unimplemented) classes that realise the interfaces in the `jagora-api` project. *You must not make any modifications to this project.*
- `jagora-test` contains an example JUnit test case called `DefaultTraderTest` for the (unimplemented) `DefaultTrader` class in the `jagora-impl` project. You will work exclusively in this project for this laboratory.

Working with your lab partner, you should extend the test harness for the JAgora system by adding JUnit test cases to the `jagora-test` project, based on the interfaces described in `jagora-api` and using the `jagora-impl` implementation classes as targets. The purpose of this exercise is to demonstrate your understanding of the intended *behaviour* of the JAgora system before you commit to implementing it. *Consequently, you should not attempt to write any code that implements system classes.* Doing so may cause you to embed unintended assumptions into your test cases.

Follow these steps to get started:

1. Start by creating *three* new projects in Eclipse for each of the three projects supplied in the laboratory archive. For example, for the project called `jagora-api` you should create a project called `jagora-api` with a root directory of `lab3-distrib/jagora-api`. You may find it convenient to change the Eclipse workspace to the `lab3-distrib` directory first.
2. The source code for the `jagora-api` and `jagora-impl` projects is contained in the directory `src/main/java`. Make sure that this directory is used as a source folder in both projects so that the code can be compiled in Eclipse. Eclipse may do this for you automatically, but if not you should: *Right click on the source directory* (e.g. `src/main/java`) the choose *Build Path* and *Use as Source Folder*.
3. The source code for tests in the `jagora-test` project is contained in a directory called `src/test/java`, so that we can separate test and application code. Make sure that this directory is also used as a source folder following a similar step to above.
4. You may also need to add the JUnit libraries in the `lib` directory to the `jagora-test` project build path. To do this, select the three libraries then *Right click*, *Build Path* and *Add to Build Path*.

5. Finally, you need to create the dependencies between projects. These are as follows:

   - `jagora-impl` depends on `jagora-api`
   - `jagora-test` depends on `jagora-api` and `jagora-impl`

   To configure the dependencies for`jagora-impl` right click on the project, then *Build Path* and *Configure Build Path...*. Select the *Projects* tab and click on *Add*. Make sure that the `jagora -api` project is checked, then click *Ok* and *Ok*.

   Repeat this process for the `jagora-test` project.

6. At this point your code should compile. Try running the `DefaultTraderTest` as a JUnit test case, as you did in Laboratory 1. Note that the `DefaultTraderTest` test case fails because the `DefaultTrader` class has not been fully implemented.

7. Now proceed to implement your own first unit test case. Start by choosing a relatively simple interface from the `jagora-api` project, such as `Stock`. Read the documentation of the interface, including the operation signatures and any available JavaDoc. *You may need to spend some time studying the interfaces to understand the behaviour they describe before you proceed to write a test case for them.*

8. Once you understand the interface's intended behaviour create a JUnit test class in the `jagora -test` project. Give the test case class an appropriate name (`DefaultStockTest` for example) and place it in the correct package (`uk.ac.glasgow.jagora.test`).

9. Create a fixture in the test case that assigns an instance of the default implementation class for the interface in the `jagora-impl` project to a field, using a method with the `@Before` annotation. For example:

   ```
   private Stock stock;

   @Before
   public void setUp () throws Exception {
     stock = new DefaultStock("lemons");
   }
   ```

10. Now create a series of methods with the `@Test` annotation that exercise the behaviours of the `DefaultStock` as they are defined in `Stock`. For example:

    ```
    @Test
    public void testName(){
      assertEquals("Expecting␣lemons.", "lemons", stock.getName());
    }
    ```

    You should devise the methods for each of your test cases based on the strategies for black box testing described in lectures. In particular, you will need to consider the pre and post conditions for each method in an interface, as well as the partition of the input space for equivalent data ranges.

11. Finally, you need to make sure that the fixture is cleaned up after the test.

    ```
    @After
    public void tearDown (){
      stock = null;
    }
    ```

12. When you run the test case all the test methods should fail or generate errors. If this doesn't happen it *suggests* that the test cannot distinguish between an empty (i.e. defective) and complete implementation of the target class as well as it might.

You should now go on to implement unit test cases for each of the other interfaces and associated default implementations in the `jagora-impl` project. Note that:

- It may not be worth testing all the classes and/or methods. Think carefully about where to focus your testing efforts based on the interface specifications. For example, the `Stock.getName` method is likely to have a very simple implementation (although it was a useful example for the above tutorial). Other methods (that belong to super classes or are auto-generated, for example) may not be worth creating tests for.
- You may also need to create a number of *stub* classes to capture some of the outputs from your target tests. Refer to the technique in lectures for how to do this. Place any stub classes in a package called `uk.ac.glasgow.jagora.test.stubs` in the `jagora-test` project.

4 Assessment

Submissions for this exercise are due by 10am two days after your lab during week 23 (week beginning 23rd Feb). You should submit your solution on Moodle in the appropriate upload slot for the laboratory. You must *only* submit a zip compressed archive containing the `jagora-test` project that you have implemented. *Do not submit either of the other two projects: these will not be considered in your assessment.* The root of this zip archive must contain a single directory called `jagora-test`, with the source code and other files for the project below this node.

Call your submission file `<matriculation>.zip`, where `<matriculation>` should be replaced with your matriculation number, for example `2001234.zip`.

Your submission will be marked on a basis of 25 marks for the assignment. Credit will be awarded for:

section	**marks**
Pair programming	5
Test case design	10
Test case coverage	10

Note that attendance at the laboratory is mandatory: any student who does not attend the laboratory and work in a pair without good cause will suffer a 5 mark penalty.

The test case design will be assessed through inspection by your tutor based on the guidance given in lectures. They will take into account the organisation of test cases (including package location); the implementation of test cases including the correct use of fixtures and test methods; the appropriate documentation of test cases and test methods, including method names; the implementation of stub classes as appropriate and the organisation of test data.

The test case coverage will be assessed using an automated tool called Pitest. Pitest will measure the coverage of your test cases against a reference implementation of JAgora. Pitest will report the percentage of *mutants* of the reference implementation (i.e. versions with deliberate bugs introduced by Pitest) that are detected by your test harness. Your tutor will round this percentage to the nearest 10% and then divide by 10 to calculate the number of marks to award. For example, a Pitest coverage of 88% will be credited with 9 marks out of 10. Note that Pitest has been configured to exclude some target methods from the coverage assessment, as they are not considered worthwhile creating tests for. The reference test suite provided to tutors achieves 98% mutation coverage.

As per the Code of Assessment policy regarding late submissions, submissions will be accepted for up to 5 working days beyond this due date. Any late submissions will be marked as if submitted on time, yielding a band value between 0 and 22; for each working day the submission is late, the band value will be reduced by 2. Submissions received more than 5 working days after the due date will receive an H (band value of 0).