COMS 417: Software Testing (Fall 2017)

LAB 4

Protocol Monitoring with Mockito

Due October 31, 12:30 pm

Objective and Background

In this lab you will write code to configure and validate a Mockito object so that it can be used to monitor and evaluate the pattern of calls *from* production code *to* the mock object. You will then use the configured Mockito object to check that the software under test makes the correct calls to the mocked object.

This kind of "bottom side" test is substantially different from everything else you have done in previous labs. All of our other Labs have focused on "top-side" testing (Figure 1). In the tests we've done so far, there was no way for the test method to determine which subordinate class methods had been called nor to deermine what order the SUT had called these methods.

Often, though, there are specific requirements for how subordinate methods are called. In this lab, the methods of the primary subordinate class (StockServiceSession) must be called in a specific order and not more than a limited number of times.

Mockito is well-suited for instrumenting this kind of Bottom Side test. We can configure the Mockito mock to accept and record the calls that are part of the protocol, and then after the SUT has finished it's work, we can use Mockito tools designed specifically for this kind of test to examine the record and evaluate whether the protocol was followed. (See Figure 2)

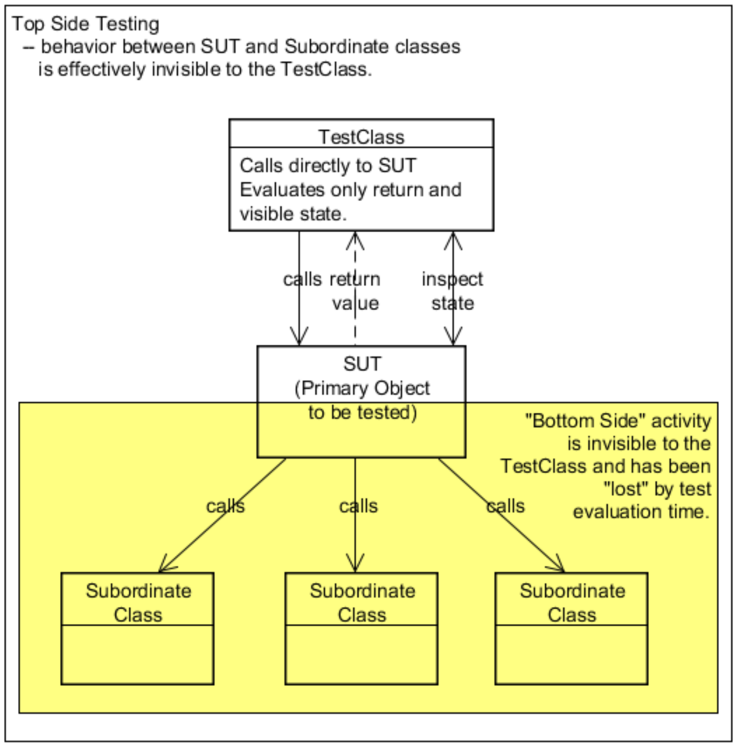

Top Side Testing
– behavior between SUT and Subordinate classes is effectively invisible to the TestClass.
TestClass
Calls directly to SUT
Evaluates only return and visible state.
calls
return value
inspect state
SUT (Primary Object to be tested)
"Bottom Side" activity is invisible to the TestClass and has been "lost" by test evaluation time.
calls
calls
calls
Subordinate Class
Subordinate Class
Subordinate Class

Figure 1 -- Top Side Testing

The arrows in Figure 2 are labelled with numbers indicating the order of their activation. Because the mocks record the activity at their interfaces, after the call to the SUT has returned (2:return value), the test class can still query the history of the subordinate objects.

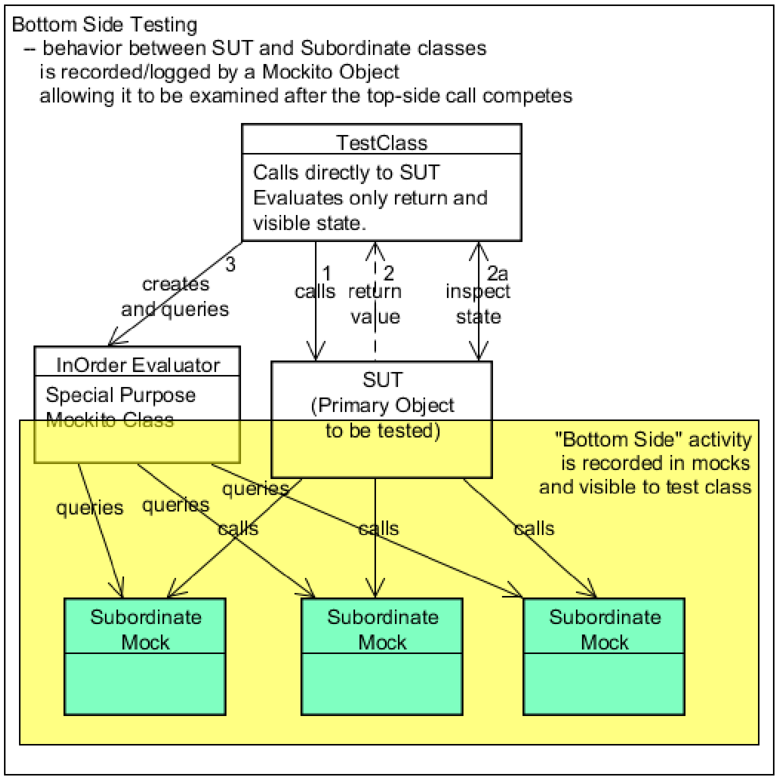

Bottom Side Testing
– behavior between SUT and Subordinate classes
is recorded/logged by a Mockito Object
allowing it to be examined after the top-side call competes
TestClass
Calls directly to SUT
Evaluates only return and visible state.
3 creates and queries
1 calls
2 return value
2a inspect state
InOrder Evaluator
Special Purpose Mockito Class
SUT (Primary Object to be tested)
"Bottom Side" activity is recorded in mocks and visible to test class
queries
queries
queries
calls
calls
calls
Subordinate Mock
Subordinate Mock
Subordinate Mock

Figure 2 -- bottom side testing

The Stock Service Protocol

In this lab, a portfolio manager calls a “remote” stock service to get current quotes (prices) for a collection of stocks known to the portfolio manager. As a security and fairness measure, this stock service requires that users authentic (log in) before making price requests, and that they make 15 or fewer price requests during each session.

The implication for the Portfolio manager is that for any portfolio of more than 15 stocks, the manger will need to establish more than one session to determine the value of the entire portfolio. In this lab, you will write the code to configure a Mockito mock suitable for monitoring this session behavior and write the code to evaluate whether each session meets the stock service’s requirements. Figure 3 shows how the primary classes interact.

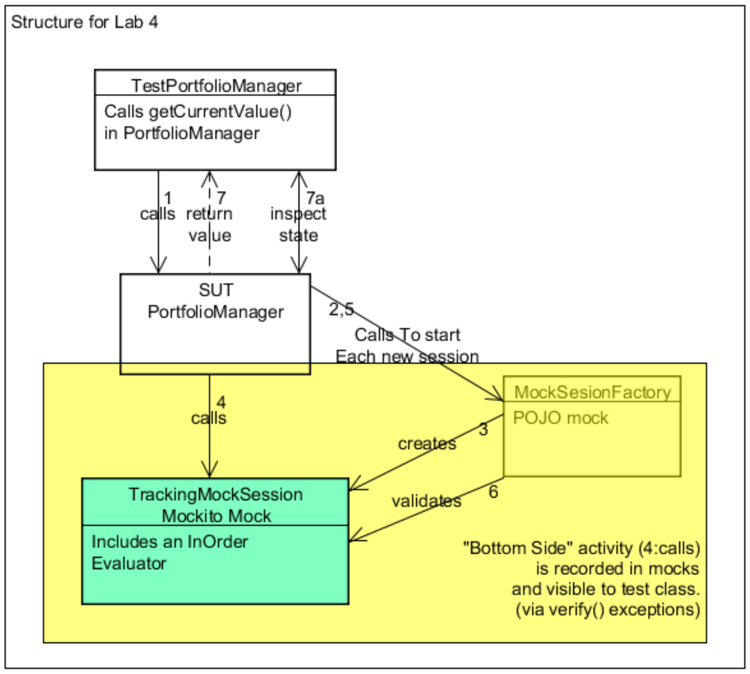

Structure for Lab 4
TestPortfolioManager
Calls getCurrentValue() in PortfolioManager
1 calls
7 return value
7a inspect state
SUT PortfolioManager
2,5 Calls To start Each new session
MockSesionFactory
POJO mock
4 calls
3 creates
6 validates
TrackingMockSession Mockito Mock
Includes an InOrder Evaluator
"Bottom Side" activity (4:calls) is recorded in mocks and visible to test class. (via verify() exceptions)

Figure 3 -- Lab 4 sructure

Figure 4 illustrates the Stock Service session protocol. You will also write test methods to check if the Portfolio correctly computes the value of different size portfolios, and test methods that exercise the Mockito object which you have completed. Figure 5 gives the class drawing for the entire project.

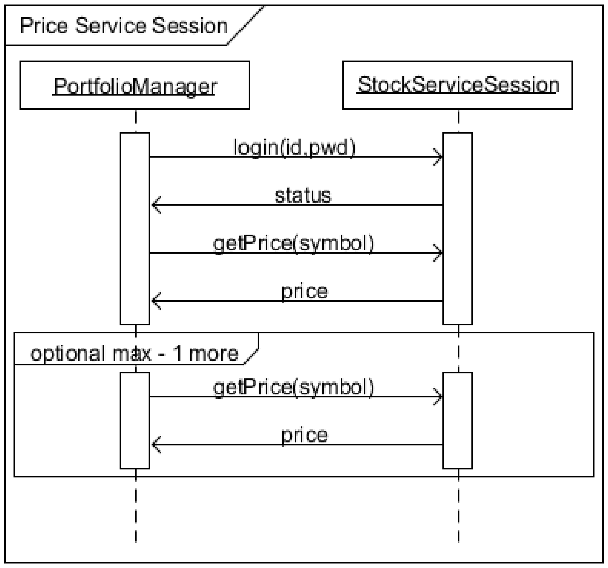
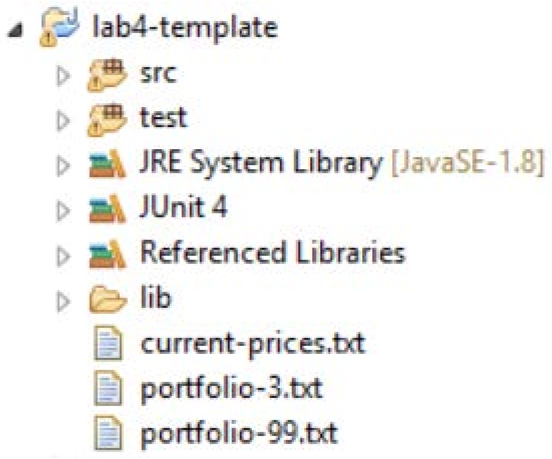


Figure 4 -- Stock Service Session Protocol

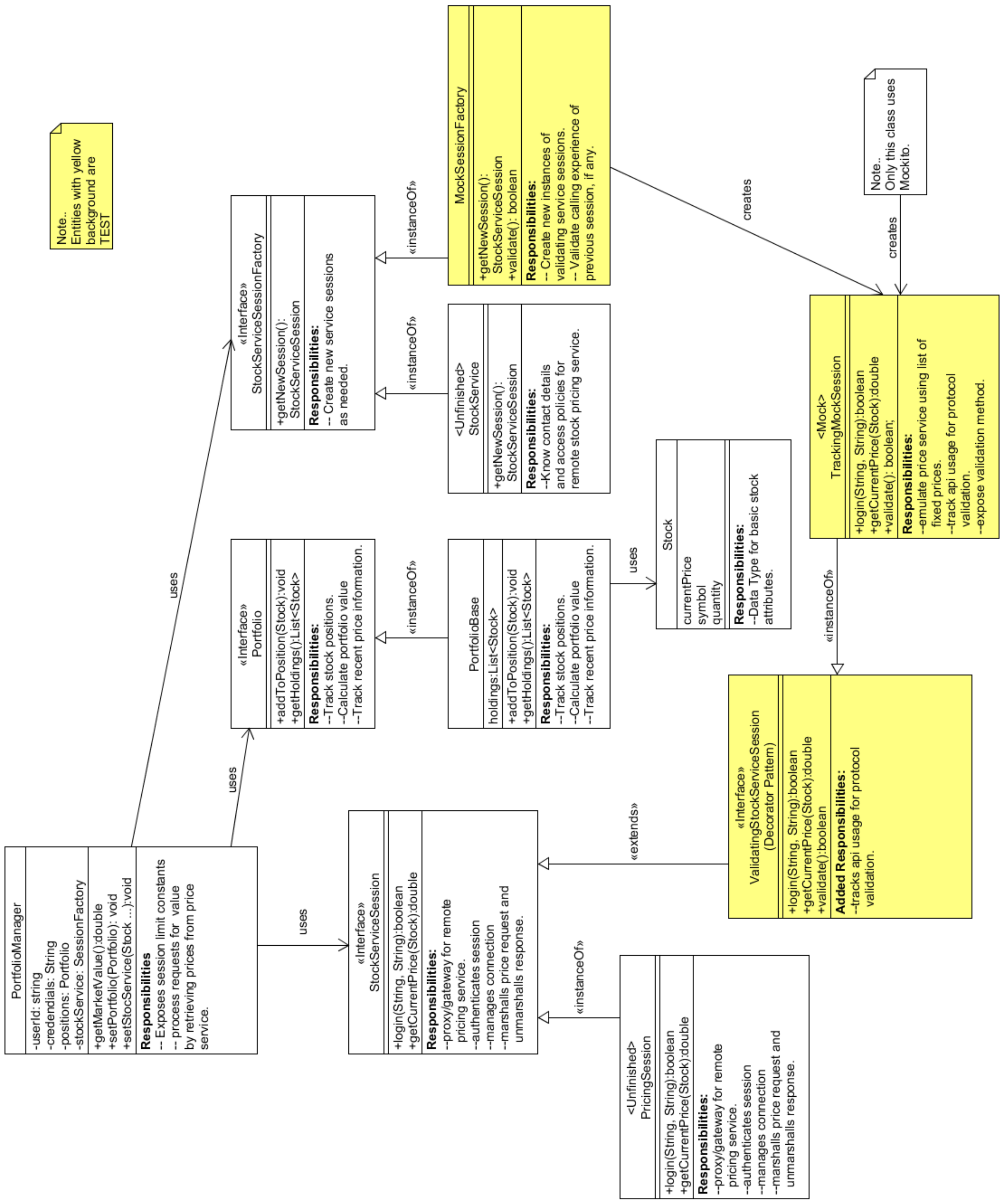
Lab Outline

1. Import (copy paste, whatever) the Lab 4 template into a normal Java project in your workspace. We do not package jar files in the template, so you may need to adjust your build path to make JUnit4 and the JRE available.
2. You will also need to find and download mockito-all-1.9.5.jar. There is a newer version of mockito, but I do not recommend using it. Once you have the mockito jar, copy it into the project's lib folder and add it to the build path as an "external library".
3. When you have the components properly assembled, the top level project structure should look like this, and the project should compile (but not run.) Note that there are three data files at the top level. The "portfolio" files contain stock symbols and quantities. These are the holdings you should load into the portfolio manager for testing (see the example test!). The portfolio-3.txt file is used by the existing (minimal) PortfolioManagerTest method.



```
lab4-template
    src
    test
    JRE System Library [JavaSE-1.8]
    JUnit 4
    Referenced Libraries
    lib
    current-prices.txt
    portfolio-3.txt
    portfolio-99.txt
```

The current-prices.txt file has prices for 99 symbols. These are the prices that the stock service should return when called. Again see the existing tests to see how it is loaded.



Note..
Entities with yellow background are TEST
«Interface»
StockServiceSessionFactory
+getNewSession(): StockServiceSession
Responsibilities:
-- Create new service sessions as needed.
MockSessionFactory
+getNewSession(): StockServiceSession
+validate(): boolean
Responsibilities:
-- Create new instances of validating service sessions.
-- Validate calling experience of previous session, if any.
«instanceOf»
«instanceOf»
<Unfinished>
StockService
+getNewSession(): StockServiceSession
Responsibilities:
--Know contact details and access policies for remote stock pricing service.
creates
creates
Note..
Only this class uses Mockito.
<Mock>
TrackingMockSession
+login(String, String):boolean
+getCurrentPrice(Stock):double
+validate(): boolean;
Responsibilities:
--emulate price service using list of fixed prices.
--track api usage for protocol validation.
--expose validation method.
uses
«Interface»
Portfolio
+addToPosition(Stock):void
+getHoldings():List<Stock>
Responsibilities:
--Track stock positions.
--Calculate portfolio value
--Track recent price information.
«instanceOf»
PortfolioBase
holdings:List<Stock>
+addToPosition(Stock):void
+getHoldings():List<Stock>
Responsibilities:
--Track stock positions.
--Calculate portfolio value
--Track recent price information.
uses
Stock
currentPrice
symbol
quantity
Responsibilities:
--Data Type for basic stock attributes.
«instanceOf»
uses
«Interface»
ValidatingStockServiceSession
(Decorator Pattern)
+login(String, String):boolean
+getCurrentPrice(Stock):double
+validate():boolean
Added Responsibilities:
--tracks api usage for protocol validation.
«extends»
PortfolioManager
-userId: string
-credendials: String
-positions: Portfolio
-stockService: SessionFactory
+getMarketValue():double
+setPortfolio(Portfolio): void
+setStocService(Stock ...):void
Responsibilities
-- Exposes session limit constants
-- process requests for value by retrieving prices from price service.
uses
«Interface»
StockServiceSession
+login(String, String):boolean
+getCurrentPrice(Stock):double
Responsibilities:
--proxy/gateway for remote pricing service.
--authenticates session
--manages connection
--marshalls price request and unmarshalls response.
«instanceOf»
<Unfinished>
PricingSession
+login(String, String):boolean
+getCurrentPrice(Stock):double
Responsibilities:
--proxy/gateway for remote pricing service.
--authenticates session
--manages connection
--marshalls price request and unmarshalls response.

Figure 5 -- Lab 4 class diagram. Yellow items are test classes.

4. Complete the method configure() in TrackingMockSession. This method is where you tell the mock how to respond to login() and getCurrentPrice() calls. I strongly suggest you work step by step. Start with versions that accept any parameters at login and always return the same price (maybe 1.00). Then make them work better. There is already a test (in TestPortfolioManager) that you should be able to run once you have specified a return value for getCurrentPrice(). See https://turgaykivrak.wordpress.com/2009/03/04/return-dynamic-result-from-mock-with-mockito/ for the basic pattern you need to follow in order to be able to return a dynamically selected price for each stock. If you are not familiar with anonymous class declarations, you may need to study
https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html
and related materials.

5. Once you are returning valid (dynamic) prices for each stock, then begin work on the validation portion of the mock object. Start by detecting when login() is called with other than the correct credentials ("Tom", "123"). Then start working on verifying the order of the calls. See https://www.tutorialspoint.com/mockito/mockito_ordered_verification.htm for the basics.
6. As you develop these protocol validation, add test cases to TestTrackingMockSession. Try to work in TDD fashion. Think of a way the protocol could be violated. Construct a test. If your validation method can't detect the error (doesn't throw the appropriate exception), then you need to do some more work on the validation function.

You should not make any changes to this project except:

- completing TrackingMockSession.configure()
- completing TrackingMockSession.validate()
- adding test methods to TestPortfolioManager.
- adding test methods to TestTrackingMockSesion.
- copy extracts from the test data files into files that support particular tests more conveniently. (e.g., a portfolio of 16 stocks).

Complete coverage of PortfolioManager (or even more correct operation) of PortfolioManager is *not* the objective of this Lab. The objective is to learn how to build a bottom side instrumentation mock and to gain experience with some of the more advanced capabilities of the Mockito library.

Hints

- Notice the includes in TrackingMockSession. The includes are a big hint about which Mockito methods/objects you will need to understand to complete this lab.
- While many of the in
- order examples show method calls against different objects, you can construct an inOrder verifier that monitors only a single object.

- Despite how most of the documentation reads, when inOrder verifies that one method call happens *before* the other. It does *not* verify that the earlier method call happens *first* on the object instance.
- In our case, showing that login() happened first is equivalent to showing that login() happened before getCurrentPrice() and that getCurrentPrice() never happened before login().
- How many times should login() happen in a single session? Test for that.
- You can define more than one InOrder validator for the same object instance.
- You can call as many standalone verify methods as you like without interfering with the behavior of InOrder.
- While you want the validation method to throw an exception when the protocol is violated, you want your test method to pass only when the required exception is thrown. I think it works best to have the test fail while you are trying to get the validation to work and then add "expected" once the validator is working properly so that the test passes.
- You ABSOLUTELY CANNOT determine if the validator is working by calling only PortfolioManger.getMarketValue(). You must write tests that call the mock object directly so that you can call it wrongly. That is why we have two different test classes. One tests whether the protocol monitor is working correctly. The other (given a working session), tests whether the PortfolioManger returns the correct value.

Other Notes

- A Test Utility class (PMTestUtil) is not included in the class drawing. It houses some methods to load data to a Portfolio and to a prices map.
- This project ignores the stock name field.

Submission Instructions

You should submit your complete project (minus bin directory and standard jars) as a zip file, with file hierarchy. We will take off points if we need to reorganize files or supply missing files or deal with unnecessarily large zips. Name the zip "<last name>_<first name>_417lab3.zip". Submit via BlackBoard.

We will accept projects that are less than 24 hours late, with a 10 percent (of possible points) penalty.