**1.**

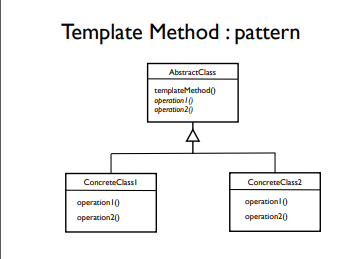
**a)**

**i)** Refactoring - improve the design of code without changing the functionality and this makes the code easier to extend and maintain. It also avoids an otherwise steep increase in the cost of making changes/adding new features as the code size grows - avoid technical debt.

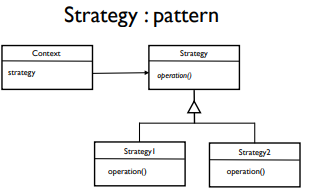
**ii)** Modules are open to extension, but closed to modification. We should aim to produce re-usable, extensible modules that are flexible, but can be specialised without changing their original source code. This minimises the risk of breaking existing code.

**b)**

**i)** Template method - uses inheritance

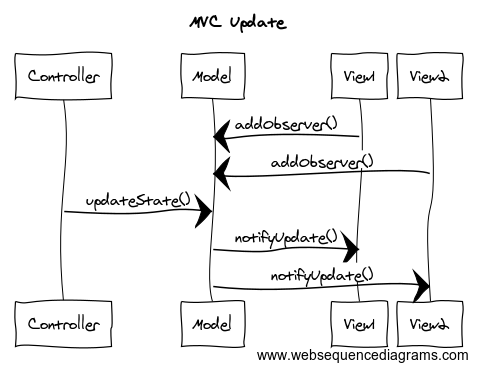


**ii)** Strategy pattern - uses class composition by delegating to a collaborator



**iii)** Strategy method, as in general we favour class composition over inheritance since this results in less coupling.

**c)**

Rough attempt:

**2.**

**a)**

Builder - identified by the chain of ‘with’ method calls (known as a fluent interface) and a final .build() method

Singleton - identified by .getInstance() method

**b)**

**i)** It would be difficult to switch as the code calls on specific classes and methods from the library (e.g. StockExchange, ConnectionException); these would all have to be individually replaced if we use an alternative library.

**ii)** We could use an adapter pattern to abstract away the third-party library by creating, for example, a StockExchange interface that is implemented by a concrete adapter class called LondonStockExchange, which in turn interacts with com.londonstockexchange.

**c)**

~~Create a class called StockPrice which replaces the com.londonstockexcahnge.StockPrice.~~

~~Create a new exception ConnectException which replaces the com.londonstockexchange.ConnectionException..~~

As above, create a StockExchange interface with buyRisingStocks() method, which takes in the 4 int parameters. Create a default implementation class LondonStockExchange and move the existing code from TradingAlgorithm to this class.

Also create a new exception class StockException and throw that in the buyRisingStocks() interface method, so we do not rely on the third-party library’s exception. Hence in LondonStockExchange, we catch the ConnectionException and throw our StockException instead.

TradingAlgorithm should have a new private final field StockExchange exchange, and a new constructor parameter for this. TradingSystem should catch our new exception in its main method.

**d)**

For this question assuming that TradingAlgorithm has the constructor of the following form:

public TradingAlgorithm(int accountNumber, int maxPrice, int minDelta, int numToBuy, Exchange exchange) {

this.accountNumber = accountNumber;

this.maxPrice = maxPrice;

this.minDelta = minDelta;

this.exchange = exchange;

}

otherwise, I can’t see of anyway to pass in the mocked object.

public class TradingAlgorithmTest {

@Rule

JunitRuleMockery context = new JUnitRuleMockery();

Exchange stockExchange = context.mock(StockExchangeAdaptor.class);

TradingAlgorithm tradingAlgorithm = new TradingAlgorithm(0, 100, 1, 0, stockExchange);

@Test

public void buyRisingStocksTest() {

context.checking(new Expectations() {{

oneOf(stockExchange).getLatestPrices(); will(returnValue(new String[0]));

oneOf(stockExchange).placeOrder(new String[0], 0 , 0);

}})

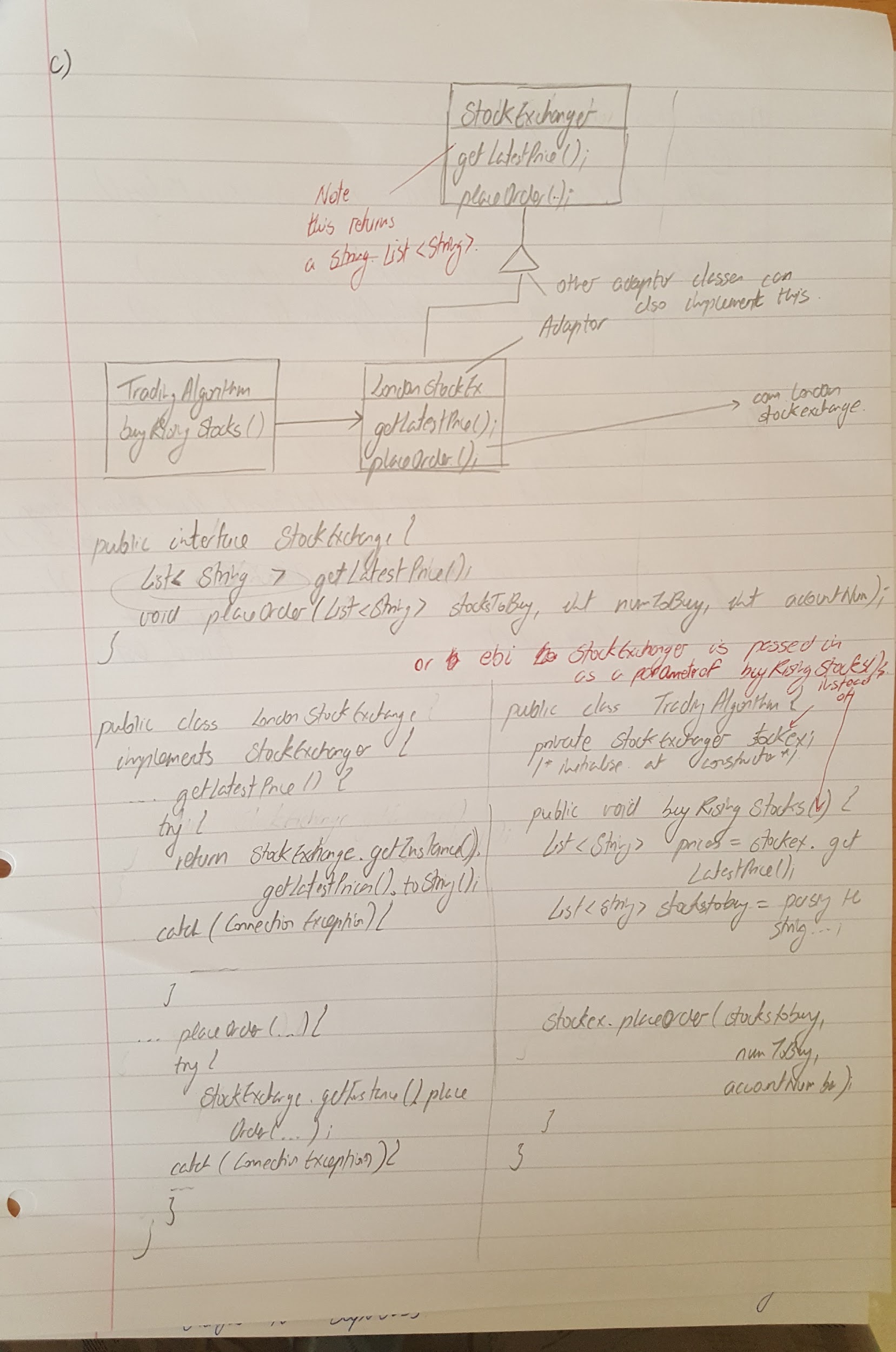
tradingAlgorithm.buyRisingStocks();

}

}

**e)**

Hexagonal Architecture (/Ports and Adapters): we separate core application logic from services upon which it depends by only accessing them through a set of adapters. This makes it easy for us to swap one implementation of a third-party service for another without impacting the core of our application.. (The only one that we have learnt..)



**d)**

public class TradingAlgorithmTests {

@Rule

public JUnitRuleMockery context = new JUnitRuleMockery();

SEAdapter adapter = context.mock(SEAdapter.class);

TradingAlgorithm ta = new TradingAlgorithm(0,0,0,0, adapter);

@Test

public void buyStockTest() {

context.checking(new Expectations() {{

exactly(1).of(adapter).getLatestPrice();

exactly(1).of(adapter).placeOrder();

}});

ta.buyRisingStocks(adapter);

}

}