**SUTD 50.001 Introduction to Information Systems and Programming**

**Problem set 3 Part A**

Note: Please submit your answer to eDimension

**Cohort Questions (Week 5)**

**Session 1**

Title: **FindSmaller**

Q1. [4 points] Write a method, public static int findsmaller(int x, int[] arr).

It returns the total count of all the elements in arr, that are smaller than or equal to x. In case there are no smaller elements in arr, it returns 0. Write a suitable TestFindSmaller class that uses the Junit framework to test the correct working of findsmaller. Illustrate what happens when there is an error in the code. (For example, deliberately introduce an error in the findSmaller method.) Include failure / error messages in your submission.

**Input:**

FindSmaller f = new FindSmaller();

int[] a = {1,2,7,6,2};

assertEquals(“Failed Assertion for FindSmaller”, 3, findSmaller(4,a);

assertEquals(“Failed Assertion for FindSmaller”, 5, findSmaller(8,a);

assertEquals(“Failed Assertion for FindSmaller”, 0, findSmaller(-1,a);

Output:

OK (1 test)

Title: **Calculator**

Q2. [4 points] Write a class Calculator that handles only integers. It takes two integer arguments and returns an integer result. It has four methods add, sub, mul and divInt, which perform addition, subtraction, multiplication and integer division respectively. For the integer division method, throw and handle a suitable exception for divide by zero, otherwise return the integer quotient. Write a suitable TestCalculator class that uses the Junit framework to test the correct working of the calculator. Include test cases for each of the operations of add, sub, mul and divInt. Include a separate test case for division by zero. Starter code is provided.

**Input:**

Calculator c = new Calculator();

assertEquals("Failed Addition", 9, c.add(4,5));

assertEquals("Failed Subtraction", 2, c.sub(5,3));

assertEquals("Failed Multiplication", 42, c.mul(6,7));

assertEquals("Failed Division", 1, c.divInt(6,4));

Output:

OK (2 tests)

**Session 2**

Title: **SingletonTest**

Q3. [3 points] Design a class caller SingletonTest, to test correct functioning of the Singleton class discussed in the cohort session. The idea is to test that only one instance of this class should exist in a program. Typically, objects of this class may be instantiated by different modules. Design a test case that compares the results of two successive instantiation requests upon this class, and generates failure if they do NOT return the same object reference. Use the Singleton pattern for your implementation. Please use Junit to develop a test case for your program.

Input:

**s1** = Singleton.getInstance();  
**s2** = Singleton.getInstance();  
*assertEquals*(**true**, **s1** == **s2**);

Output:

OK (1 test)

Title: **StockGrabber**

Q4. [12 points] Using the Observer design pattern, develop a stock alert system that sends alerts to subscribers for any update of the stock prices. Develop StockGrabber class that keeps the list of subscribers for several stock prices, and notifies them for update. Develop StockObserver class that monitors changes in the stock prices. Develop a StockTest class to simulate the system. You may choose to do this using Junit if you like.

**Example test input:**  
StockGrabber IBM = **new** StockGrabber();  
StockObserver scott = **new** StockObserver(IBM, **"scott"**);  
StockObserver jim = **new** StockObserver(IBM, **"jim"**);  
IBM.stockUpdate(**"IBM going up by $0.05"**);  
StockObserver jeremy = **new** StockObserver(IBM, **"jeremy"**);  
IBM.stockUpdate(**"IBM is going down by $0.03"**);

**Output:**

**scott: Message received: IBM going up by $0.05**

**jim: Message received: IBM going up by $0.05**

**scott: Message received: IBM is going down by $0.03**

**jim: Message received: IBM is going down by $0.03**

**jeremy: Message received: IBM is going down by $0.03**

**Session 3**

Title: **TaxVisitor**

Q5. [12 points] Use the Visitor design pattern, compute the total tax payable on a list of items of the following types: Car, Electronics, Chocolate. The tax is payable under two conditions, whereby two taxing systems are in place:

TaxNormal:

Car: 40%, Electronics: 80%, Chocolate: 20%

TaxHoliday:

Car: 30%, Electronics: 50%, Chocolate: 10%

**Sample test class:**

**public class** MyClass {  
 **public static void** main (String[] args) {  
  
 ArrayList<Visitable> items = **new** ArrayList<Visitable>();  
 TaxVisitor tax = **new** TaxVisitor(**"TAXHOLIDAY"**);  
  
 items.add(**new** Car(100000));  
 items.add(**new** Electronics(2000));  
 items.add(**new** Chocolate(100));  
  
 **for** (Visitable o : items) {  
 o.accept(tax);  
 }

System.out.print("This is the Tax Visitor program:");  
 System.***out***.println (tax.getTotal());  
 }  
}

**Output:**

**This is the Tax Visitor program. Total tax:**

**31010.0**

**Homework Questions (Weeks 5 and 6)**

Title: **FindX**

Q1. [5 points] The class FindX has a method findX (int x, int[] a), that finds the index of the first occurrence of the integer x in an array a of integers, or returns -1 if x is not found. As an additional feature, the program also maintains a count of the number of times the object of class FindX has been visited. Construction of the object also counts as a visit.

The class FindX is provided in the starter code. You are to complete TestFindX that extends TestCase from the Junit framework. Design a suitable test that contains test cases for FindX that result in success and failure (i.e. number not found). Write a separate test that tests the incremental increase in the number of visits.

Input (First Test Case):

FindX f = **new** FindX();  
**int**[] a = {4,5,6};  
*assertEquals*(**"Failed Assertion for FindX"**, 0, f.findX(4,a));  
*assertEquals*(**"Failed Assertion for FindX"**, 1, f.findX(5,a));  
*assertEquals*(**"Failed Assertion for FindX"**, 2, f.findX(6,a));  
*assertEquals*(**"Failed Assertion for FindX"**, -1, f.findX(9,a));

Input (Second Test Case):

FindX f = **new** FindX();  
*assertEquals*(**"Failed Assertion for Counting Number of Visits"**, 1, f.getVisitNumber());  
**int**[] P = {4,5,6};  
**int** v1 = f.findX(4,P);  
*assertEquals*(**"Failed Assertion for Counting Number of Visits"**, 2, f.getVisitNumber());  
**int** v2 = f.findX(5,P);  
*assertEquals*(**"Failed Assertion for Counting Number of Visits"**, 3, f.getVisitNumber());

**Output:**

**OK (2 tests)**

Title: **TestTotalPay**

Q2. [10 points]: Add a method called getTotalPay() that returns void, to the TestEmployee class discussed in your cohort session. Design appropriate tests to make sure that:

* the total pay is correctly calculated and returned
* in case an employee does not have any salary record, your program should generate an exception, which is normal and hence will not be reported as failure. If the exception is not caught however, your program should report failure.

Please use Junit framework. Starter code for classes Employee, SalaryComponent and TestEmployee have been provided. You are to modify and complete all three classes.

**Output:**

**OK (3 tests)**

Title: **TrafficAlert**

Q3. [10 points] Use the Observer design pattern to develop a traffic alert system that provides instantaneous alerts to all the subscribed road users about the conditions on the road. Copy and suitably modify Observer pattern samples discussed in the cohort activities. Create a Traffic class to implement interface Subject. Create a RoadUser class to implement Observer.

**Input:**

Traffic sgTraffic = new Traffic();

RoadUser u1 = new RoadUser("man", sgTraffic);

RoadUser u2 = new RoadUser("andrew", sgTraffic);

sgTraffic.changeCondition("Heavy Traffic on PIE");

sgTraffic.unregister(u1);

sgTraffic.changeCondition("Roadworks on Keppel Road");

**Output:**

**man receives alert: Heavy Traffic on PIE**

**andrew receives alert: Heavy Traffic on PIE**

**andrew receives alert: Roadworks on Keppel Road**

Title: **GrandTotalVisitor**

Q4. [10 points] Use the Visitor design pattern, compute the grand total price of a list of grocery items: Milk, Fish, Jacket. The sub-total of each grocery item is calculated as follows:

* For Milk, the sub-total is its price.
* For Jacket, the sub-total is its price and an additional 20% tax.
* For Fish, the sub-total is the product of its price per kg and its weight in kg.

The starting code has been provide. Modify the starting code:

• insert code for interface Visitable

• insert code for interface Visitor

• insert code for concrete class GrandTotalVisitor, which calculates and accumulates the sub-total of visited items

• modify Milk, Jacket and Fish classes to implement Visitable

**Example:**

new Milk(10);

new Jacket(20);

new Fish(4,2);

**Output:**

**The grand total is: 42.0**

Title: **AirPollutionAlert**

Q5. [15 points] Use the Observer design pattern to develop an air pollution alert system that sends the air pollution index to all the subscribed users if the air pollution index is more than 100.

Copy and suitably modify the starting code for Observer pattern cohort exercises. Create an AirPollutionAlert class to implement interface Subject. If your implementation of Observer Design Pattern is correct, you should see following results.

**Input:**

AirPollutionAlert singaporeAlert = new AirPollutionAlert();

Subscriber man = new Subscriber("man", singaporeAlert);

Subscriber simon = new Subscriber("simon", singaporeAlert);

singaporeAlert.setAirPollutionIndex(200);

singaporeAlert.setAirPollutionIndex(50);

singaporeAlert.setAirPollutionIndex(120);

singaporeAlert.unregister(man);

singaporeAlert.setAirPollutionIndex(300);

**Output:**

**man received notification: 200.0**

**simon received notification: 200.0**

**man received notification: 120.0**

**simon received notification: 120.0**

**simon received notification: 300.0**

Title: **Geometric Shapes**

Q6. [15 points] Use the Visitor design pattern, compute the total area of a list of items of the following types: Circle, Square, Rectangle. Also compute the total perimeter of the items using another Visitor instance. Copy and suitably modify the starting code for Visitor pattern cohort exercises as follows:

• insert code for interface Visitable

• insert code for interface Visitor

• insert code for concrete class AreaVisitor, which calculate and accumulate the total area of visited items

• insert code for concrete class PerimeterVisitor, which calculate and accumulate the total perimeter of visited items

• modify Circle, Rectangle and Square classes to implement Visitable

**Input:**  
shapes.add(**new** Circle(10));  
shapes.add(**new** Square(4));  
shapes.add(**new** Rectangle(2,4));

**Output:**

**Hello World! Testing Geometry Visitor.**

**Area: 338.1593**

**Perimeter: 90.8319**