FIRST NAME:
LAST NAME:
NIA:
GROUP:

First midterm exam

Second Part: Problems (7 points out of 10)

Duration: 70 minutes
Highest score possible: 7 points
Date: March 24, 2021

Overall instructions for the exam:

- Books, notes, mobile phones, as well as other electronic devices are not allowed during the exam. Breaking this rule may result in expulsion from the examination
- Complete your personal information before starting the exam.

Problem 1 (5.5 / 7 points)

In the store of the course project, there is a need to implement new functionalities to control the shipments made to deliver the orders. In this context, you are required to implement several classes to model couriers and shipments. In addition, you are provided with the following interface, which contains the current functions that couriers carry out. Particularly, there is a method to compute the commissions that the courier will apply.

```
public interface CourierFunctions {
       double calculateCommission(double cost) throws LowCostException;
}
```

Section 1.1 (0.5 points) – Class Courier

Implement the class `Courier`. This class implements the aforementioned interface, and it stores the name of the courier company (`name`). The name can only be visible inside the package where this class is located. The class also has a constructor to initialize the name. However, there is not enough information to compute the commissions since they depend on how fast the delivery is done.

Section 1.2 (1 point) – Class FastDeliveryCourier

Implement the class `FastDeliveryCourier`. This class is used to model a courier which can deliver the order in a fast way. This class only stores the name of the courier company (`name`), and has two constructors. The first constructor initializes all the attributes, and the second constructor is an empty overloaded constructor, which calls the first one to initialize the name as `null`. In this case, the commissions can be calculated, and they are the 20% of the cost given. However, if the cost is less than 20€ (a value of 20), a `LowCostException` (which is already implemented) is thrown with the message **"`Minimum cost is not reached`"**.

Section 1.3 (1.5 points) – Class declaration and attributes of class Shipment

Implement the class `Shipment`. This class is used to model the shipments that are used to deliver the orders. This class stores the following information:

- The Order (`order`) that contains the `ProductList`, with their corresponding `StockableProduct`.
- The courier company that delivers the order (`courier`).
- The price of the shipment (`price`).
- The status of the shipment (`status`). This variable can take two values, depending on two constants, that are also defined in this class:
    - SENT = 1
    - DELIVERED = 2

- An ArrayList of String (`messages`) that keeps records of the events related to the shipment
- A variable to store the total number of shipments that have been done so far (`total_shipments`)
- A variable to store the total number of shipments that have been done with fast delivery so far (`total_fast`)

All the attributes, excepting the constants, can only be visible inside the class. Constants should be visible anywhere.

NOTE: Remember that there is a class `Order` to represent orders.

Section 1.4 (1.5 points) – Constructor of class Shipment

In addition, you need to implement the constructor of the class Shipment to initialize its attributes. When implementing the constructor, you should take into account the following instructions:

- You can initialize the `order` and `courier` using their corresponding sets (`setOrder(...)` and `setCourier(...)`). You can assume that these methods are already implemented for you.
- When a shipment is created, its initial status is *SENT* by default.
- The price of the shipment is the sum of the cost of the order and the commissions of the courier. Remember that there is a method `calculateCost()` in class `ProductList` that and Order extends `ProductList`.
- The array of messages is initialized with a message containing a timestamp and the text "Order sent". The message has this format: **"1614117309256 - Order sent"**. You can get the timestamp using the method `System.currentTimeMillis()`, which is an already implemented method in Java (class `System`).
- You need to update variables `total_shipments` and `total_fast` each time an object is created. However, `total_fast` is only updated if the courier is a `FastDeliveryCourier`. Hint: You may use *instanceof* to check this.

Section 1.5 (1 point) – Testing

Finally, another team has implemented a method to compute the commissions for the `SlowDeliveryCourier` class, the class used to model couriers that deliver their shipments in a slower way. In this class, you are given the code of the method `calculateCommission` to do the testing:

```
@Override
public double calculateCommission(double cost) throws LowCostException {
	if(cost < 20)
		throw new LowCostException("Minimum cost is not reached");
	else if(cost < 50)
		return 0.05*cost;
	else if(cost < 100)
		return 0.10*cost;
	else
		return 0.15*cost;
}
```

a) Write a method to test the `calculateCommission` method. This method should test all the equivalence classes, except for the equivalence class that corresponds to the highest costs.

b) What is the branch coverage achieved with the previous test?

Problem 2 (1.5 / 7 points)

A cipher is a secret text that represents a message. In this question, each character of the message is represented by a string of length 5 used in the cipher. For example, the text **"leo"**, which consists of 3 characters, is represented as a 15 characters string: **"akoakijgzelbkfa"**.

To decode the entire cipher, we decode the last 5 characters of the cipher into the first character of the original message and the second last 5 characters of the cipher into the second character of the original message and so on. To decode a chunk of 5 characters, you can use the method `char decipherChar(String s)`, which is already implemented. The following table outlines the process of decoding the cipher.

Cipher	"akoakijgzelbkfa"		
5-character chunks	"akoak"	"ijgze"	"lbkfa"
Decoded characters from chunks	'o'	'e'	'l'
Reconstructed message	"leo"		

Complete the **recursive** method `String decipher_recurs(String cipher)`, which takes the cipher as input and whose output is a String with the original message and with the following error checking:

1. If the input cipher is an empty String **""**, return the String **"no cipher"**.
2. If the input cipher does not have a length divisible by 5, return the String **"invalid cipher"**.

NOTE 1: Your method must be **recursive.** Non-recursive solutions will not be considered.

NOTE 2: You may use auxiliary methods if you need them

NOTE 3: You may use the method `substring` of class String to return part of a String. This is an overloaded method with two possible implementations: (1) `substring(int beginIndex)` which returns the substring that begins at index `beginIndex`, and (2) `substring(int beginIndex, int endIndex)`, which begins at `beginIndex` and ends at index `endIndex-1`.

REFERENCE SOLUTIONS (several solutions are possible)

PROBLEM 1

Section 1.1 (0.5 points)

```
public abstract class Courier implements CourierFunctions {
	String name;

	public Courier(String name) {
		this.name = name;
	}

}
```

Evaluation criteria:

- 0.20: Class declaration including abstract and implementing the interface.
- 0.10: Attribute name with correct visibility
- 0.10: Signature of the constructor
- 0,10: Initialization of the attribute
- Significant errors are subject to additional penalties

Section 1.2 (1 point)

```
public class FastDeliveryCourier extends Courier {

	public FastDeliveryCourier(String name) {
		super(name);
	}

	public FastDeliveryCourier() {
		this(null);
	}

	@Override
	public double calculateCommission(double cost) throws LowCostException {
		if(cost < 20)
			throw new LowCostException("Minimum cost is not reached");
		return 0.2*cost;
	}

}
```

Evaluation criteria

- 0.10: Class declaration extending Courier
- 0.20: Constructor with parameters
- 0.20: Constructor without parameters
- 0.10: Signature of method calculateCommission, including throws
- 0.10: Check the condition
- 0.10: Throw LowCostException
- 0.20: Compute (0.1) and return (0.1) the commission
- Significant errors are subject to additional penalties

Section 1.3 (1.5 points)

```
public class Shipment {
	private Order order;
	private Courier courier;
	private double price;
	private int status;
	private ArrayList<String> messages;

	public static final int SENT = 1;
	public static final int DELIVERED = 2;

	private static int total_shipments;
	private static int total_fast;
```

Evaluation criteria

- 0.10: Class declaration
- 0.30: Attributes order and courier (0.15 each)
- 0.20: Attributes price and status (0.1 each)
- 0.10: Attribute messages
- 0.40: Constants (0.2 each)
- 0.40: Static variables (0.20 each)
- Significant errors are subject to additional penalties

Section 1.4 (1.5 points)

```
public Shipment(Order order, Courier courier) throws LowCostException {
		setOrder(order);
		setCourier(courier);
		status = SENT;
		price =
order.calculateCost()+courier.calculateCommission(order.calculateCost());

		messages = new ArrayList<String>();
		messages.add(System.currentTimeMillis() + " -  Order sent");

		total_shipments++;
		if(courier instanceof FastDeliveryCourier)
			total_fast++;
}
```

Evaluation criteria

- 0.20: Signature of the constructor with correct parameters (0.1) and throws (0.1)
- 0.20: Call to set methods to initialize order and courier (0.1 each)
- 0.10: Set initial status to SENT
- 0.40: Calculate price using methods from class Order and Courier
- 0.10: Initialize ArrayList
- 0.20: Add message in ArrayList (0.1) using System.currentTimeMillis() appropriately (0.1)
- 0.20: Increment static variables (0.1 each)
- 0.10: Check if courier is FastDeliveryCourier using instanceof
- Significant errors are subject to additional penalties

Section 1.5 (1 point)

```
@Test
void test() throws LowCostException {
	SlowDeliveryCourier s = new  SlowDeliveryCourier("Courier SL");
	assertThrows(LowCostException.class, ()->{s.calculateCommission(5);});
	assertEquals(s.calculateCommission(20), 1);
	assertEquals(s.calculateCommission(60), 6);
	 // Branch coverage = 5/6
}
```

Evaluation criteria

- 0.10: Signature of the method including @Test and throws
- 0.10: Creation of the object
- 0.20: Case with assertThrows
- 0.20: Case when prize is between 20 (included) and 50 (not included)
- 0.20: Case when prize is between 50 (included) and 100 (not included)
- 0.20: Branch coverage
- Significant errors are subject to additional penalties

PROBLEM 2 (1.5 points)

Section 2.1 (1.5 points)

```
public static String decipher_recurs(String cipher) {
	// Perform error checking
	if (cipher.equals("")) return "no cipher";
	if (cipher.length() % 5 != 0) return "invalid cipher";
	return decipher_recurs(cipher,"");
}

public static String decipher_recurs(String cipher, String result) {
	if(cipher.equals(""))
		return result;

	return decipher_recurs(cipher.substring(5),result) +
decipherChar(cipher.substring(0, 5));
}
```

Evaluation criteria:

- 0.20: Error checking (0.1 each condition)
- 0.20: Call to the auxiliary method with the correct parameters
- 0.10: Signature of the auxiliary method
- 0.20: Base case
- 0.80: Recursive case
    - 0.40: Recursive call
    - 0.20: Call to deciperChar
    - 0.20: Return ensuring that the order is correct
- Significant errors are subject to additional penalties