Paper Number(s): **ISE2.1**

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE
UNIVERSITY OF LONDON

DEPARTMENT OF ELECTRICAL ELECTRONIC ENGINEERING
EXAMINATIONS 2000

SOFTWARE ENGINEERING 2

Friday, May 5 2000, 2:00 pm

There are 5 questions on this paper. Answer 3 questions.

Time allowed: 2:00 hours.

Examiner(s): Dr Porkolab

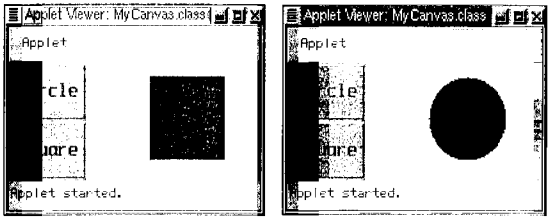
1 Consider an online retailer that has *Customers*. Some of the customers are *Corporate Customers*, while others are *Personal Customers*. (A *Customer* is either a *Corporate Customer* or a *Personal Customer*.) Each *Customer* has a name and address, and the *Customer*'s credit rating can be checked. The retailer keeps on file the customer's credit card number for each *Personal Customer*, and a contact name and credit limit for each *Corporate Customer*. The retailer can send the monthly bill to each *Corporate Customer* and also a reminder (if this becomes necessary). In addition, a *Corporate Customer* may be associated with an *Employee* who serves as a sales representative of the company. *Customers* may make (several) *Orders* over time. Each *Order* comes from a single *Customer* and may have several *Order Lines*, each of which refers to a single *Product*. *Orders* have various attributes, the date when they were received, an identifying order number, the price of the order and an indicator whether it has to be prepaid. (If the credit rating of the *Customer* who has made the order is not satisfactory, the prepaid attribute must be true.) During processing one can close and dispatch *Orders*. Finally, each *Order Line* has a price and quantity for the particular product it refers to, plus an indicator whether it can be satisfied or not.

a Draw a UML **class diagram** to model the above order processing system from an object-oriented point of view. Indicate also multiplicities for the associations (wherever it is applicable). [10 marks]

b Explain briefly the notion of **class inheritance** in object-oriented design, and demonstrate it by an example from the above described order processing system. [4 marks]

c Discuss briefly the similarities and differences of **abstract classes** and **interfaces** (in Java). Name a class from the above order processing system that is likely to be implemented as an abstract class, and justify your answer. [6 marks]

2 Write a Java applet that allows the user to select from two different shapes, circle and square, and draws either a filled circle or square depending on the user's choice. The user interface of the applet consists of two buttons with labels Circle and Square, plus a drawing area as it is shown on the figure below. [Hint: You may organize your Java code using the following three classes: *MyCanvas* to run the applet, *CustomCanvas* to draw the chosen shape and *ButtonHandler* to handle button events for the applet. The two buttons can be layed out on a *Panel* object by using *GridLayout* with two rows and one columns, and for drawing a filled circle or square you can use `g.fillOval(50,10,60,60)` and `g.fillRect(50,10,60,60)`, respectively.] [20 marks]



3 Consider the following (partial) definition of Java classes used in a supermarket application. [A customer object uses the field named *orders* to reference an array of outstanding orders placed by the customer. An order object uses the field *receiver* to hold a reference to a customer object representing a customer receiving that order, and uses another field named *items* to hold (the reference to) an array of ordered items. An item stores a reference to a type of merchandise and an integer field with name *quantity* that keeps track of how many of this merchandise type were requested. Each merchandise object has two attributes refering to its name and price.]

```
class Merchandise {
   String name;
   float price;

   Merchandise(String n, float p){
        name = n;
        price = p;
   }
}

class Item {
   Merchandise item;
   int quantity;

   Item(Merchandise m, int q){
        item = m;
        quantity = q;
   }

}

class Order {
   Customer receiver;
   Item[] items;
   ...
}

class Customer {
   String name, address, telephone;
   float balance;
   Order[] orders;
   ...
}
```

a To extend class *Order* with additional methods that also have exception handling facilities, one may want to have an exception class *NoSuchMerchandiseOrdered*, which signals the absence of a specified type of merchandise in an order. Define this exception class such that its constructor accepts a merchandise argument and an integer argument

(where the latter one refers to an intended new quantity of the merchandise). [4 marks]

b Add two methods into class *Order*:

- Method *findItem(Merchandise p)* tries to find a given type of merchandise in an order. If the mentioned type of merchandise cannot be found in the order, the method returns *null*.
- Method *changeQuantity(Merchandise p, int newQuantity)* uses the previous method and changes the quantity of a type of merchandise for an order. In the normal situation, the order contains the type of merchandise in an item; otherwise, the method throws a *NoSuchMerchandiseOrdered* exception.

[8 marks]

c Add the *main* method into class *Order* to test the above exception handling facility. The method should create at least two merchandises (e.g. "tomato" with price 0.19 and "potato" with price 0.99) and place an order for only one of the created merchandises. Then it should call method *changeQuantity* for both of the merchandises and inform the user about any failure of quantity change request. [8 marks]

4 Class *M-aryTree* implements multi-ary trees, each of which has a fixed arity m when it is created. Each node in an m-ary tree has **at most** m children that are assumed to be ordered so that they can be indexed.

a Define (in Java) class *Node* that implements a node for multi-ary trees. Provide different constructors that allow you to create a node with a given arity or with a given arity and data to be stored at the node. (The arity field of a node is used to limit the number of children for the node.) Furthermore, provide instance methods to access and update the fields of the node: methods to add a new child (as the last one) to the node, add/access a child with a specified index, and return the current degree (i.e. number of children) of the node. [12 marks]

b The *elements()* method of class *M-aryTree* returns an enumeration of the nodes of the tree by using the class *Traversal* that implements the *Enumeration* interface from the *java.util* package. This interface is defined as follows:

```
public abstract interface Enumeration {
    public abstract boolean hasMoreElements();
    public abstract Object nextElement();
}
```

Provide a Java implementation of class *Traversal* assuming that class *M-aryTree* uses field *root* to hold a reference to the root node of the tree. [8 marks]

5a Define class *Graph* to implement in Java an **undirected**, unweighted graph ADT. Assume that the maximum number of vertices in a graph is fixed after the graph object is created, but edges can be dynamically added and removed from the graph. Also assume that vertices are indexed, so the index of a vertex can also be used to access the vertex. The graph ADT has the following constructors

```
public Graph(),
public Graph(int vertexNumber),
public Graph(Object[] vertices),
```

and fields

```
boolean[][] adjacencyMatrix,
Object[] vertices,
int vertexNumber.
```

The class is tested with a main method which includes the following object creations:

```
String[] names = {"London", "Paris", "Rome"};
Graph stringsGraph = new Graph(names);
stringsGraph.addEdge("London", "Paris");
```

We also want to be able to set and access a vertex at a given index, and test whether two vertices (either both specified by indices or as objects) are adjacent. For simplicity, the implementation of methods for removing edges is not required. [14 marks]

b Define class *Digraph* in Java by using class *Graph* to implement a **directed**, unweighted graph ADT. (The same assumptions apply as above in part a.) [Note, in directed graphs, edges (also called as arcs) are directed, so the existence of an edge from vertex i to vertex j does not imply (in contrast to undirected graphs) the existence of the reverse/opposite edge from vertex j to vertex i.] [6 marks]

End of Paper