EXAMINATION QUESTION PAPER: Semester 2, 2007/8

Module code:	**CSP032N**
Module title:	**Object-oriented Software Design and Development**
Module leader:	**Dr. Quan Dang**

Date:	**13 May 2008**
Day / evening:	**Day**
Start time:	**14:00**
Duration:	**2 hours**

Exam type:	**Unseen, Closed**
Materials supplied:	**None**
Materials permitted:	**None**
Warning:	**Candidates are warned that possession of unauthorised materials in an examination is a serious assessment offence.**

Instructions to candidates:	**Answer any Four (4) of the SIX questions.** **For each question 25 marks are available.** **No credit will be given for attempting further questions.** **DO NOT TURN PAGE OVER UNTIL INSTRUCTED**

QUESTION 1

(a) What is a "*use case*"? Explain the purpose of use cases in software design.

(5 marks)

(b) Provide an example of a use case from a software development scenario of your own choosing, listing its sequence of actions and variations.

(7 marks)

(c) What is a CRC card? Describe the purpose of each compartment of the card.

(5 marks)

(d) *"CRC cards can be used to negotiate the use case's operations with classes' responsibilities in object-oriented software design"*. Identify at least two classes from the scenario in your answer to sub-question (b), then draw CRC cards for the two classes to explain the statement.

(8 marks)

QUESTION 2

(a) In the context of object-oriented design

(i) Name three types of relationship between classes.

(3 marks)

(ii) For each type of relationship of your choice, provide a definition, an example and its standard UML notation in a UML class diagram.

(6 marks)

(b) Inspect the code of a simple calculator in Java provided in Appendix A at the end of this paper.

(i) Draw a UML class diagram of the classes in the code, explaining the relationship between the classes.

(4 marks)

(ii) Re-write the provided code to minimise the dependency between the classes.

(8 marks)

(iii) Explain why your re-written code is better than the original version.

(4 marks)

QUESTION 3

"Encapsulation refers to the bundling of data with the methods that operate on that data. Often that definition is misconstrued to mean that the data is somehow hidden. In Java, you can have encapsulated data that is not hidden at all." (Author: Paul Rogers, JavaWorld.com, 18/05/2001)

(a) Distinguish *Encapsulation and Information hiding* in OO programming to elaborate the point of the above author.

(4 marks)

(b) Why would the use of encapsulation and information hiding result in more robust software?

(4 marks)

(c) Provide an example in Java code to illustrate the author's point "*In Java, you can have encapsulated data that is not hidden at all"*.

(9 marks)

(d) The same author argues that *"hiding data is not the full extent of information hiding"*. Would you agree with the argument? Discuss and provide an example to illustrate your answer.

(8 marks)

QUESTION 4

(a) From a practical programming viewpoint, polymorphism manifests itself in three distinct forms in Java:

- Method overloading
- Method overriding through inheritance
- Method overriding through the Java interface

Provide an example of, and explain in more detail, each form of polymorphism.

(9 marks)

(b) Consider the code of the Car class provided in Appendix B. What class is the parent class of the Car class? Which method does the Car class inherit from its parent class and override?

(7 marks)

(c) Rewrite the code of the Car class to allow its instances to be compared by their engine size in an ascending order.

(9 marks)

QUESTION 5

(a) Explain what an *abstract class* is. Can an abstract class include a method with implementation? Provide an example in your answer.

(7 marks)

(b) Explain what a Java *interface* is and its purpose. Compare the Java interface and the abstract class.

(8 marks)

(c) What is a *software pattern*? What are the major benefits of using software patterns in software development? Name and concisely describe three software patterns which you know.

(10 marks)

QUESTION 6

(a) Explain in any suitable manner, what "*class design with loose coupling*" means, providing a concrete example in your answer.

(8 marks)

(b) In the context of object-oriented programming the law of Demeter states that a method should only use objects that are: instance fields of its class, parameters or objects that it constructs with new methods.

(i) Inspect the code below and explain why the code in the class Timetable violates the law of Demeter.

```
public class Timetable {
   public String showModuleLeader(Module myModule)
   {   Module aModule= myModule;
       return aModule.getModuleLeader().getStaffName();     }
  // there are other methods ….
}

class Module {
   Staff moduleLeader;
   public void setModuleLeader(Staff aStaff)
   {   moduleLeader = aStaff;      }

   public Staff getModuleLeader()
   {       return moduleLeader;       }
}//~class Module...

class Staff {
   String name;
   public Staff(String StaffName)
   {       name = StaffName;    }

   public String getStaffName()
   {       return name;    }
}//~class Staff...
```

(8 marks)

(ii) Re-write the code of classes Timetable and Module to conform to the law of Demeter, preserving the same effect of the showModuleLeader() method.

(9 marks)

Appendix A

```
import javax.swing.JOptionPane;
/** simple calculator */
class Calculator {
   public static void main(String args[])
   {       AddOperation performAddition = new AddOperation();
      int sum = performAddition.getSum();
```

```
        System.out.println("The result is: " + sum);
    }
}

/** a class to get user input */
class Keypad {
    public int getInput()
    {   String input = JOptionPane.showInputDialog("Please enter an integer?");
        if (input != null) return Integer.parseInt(input);
        else return 0;
    }
}

/** an operation to calculate the sum of 2 integers */
class AddOperation {
    public int getSum()
    {   Keypad myKeypad = new Keypad();
        int num1=0, num2=0;
        num1 = myKeypad.getInput();
        num2 = myKeypad.getInput();
        return num1 + num2;
    }
}//~class Add...
```

Appendix B

```
public class Car implements java.lang.Comparable
 {
    double carLength = 0; double carEngineSize = 0;
    String carMake =”no make”;
    public Car(String make) { carMake = make; }

    public void setCarLength(double length) { carLength = length; }
    public void setCarEngineSize (double EngineSize) { carEngineSize = EngineSize;}

    public double getCarEngineSize () { return carEngineSize; }
    public double getCarLength() { return carLength; }
    public String getCarMake(){ return carMake; }

    public String toString() { return carMake + " " + carLength + "metres long"; }

    public int compareTo(Object anotherCar)
    {
        if (this.getCarLength() < ((Car)anotherCar).getCarLength())
        return -1;
        else
            if (this.getCarLength() == ((Car)anotherCar).getCarLength()) return 0;
            else return 1;

    }//~public int compareTo(Obj...
}//~class car...
```