Name: ________________________________

Student ID:________________________________

Signature:________________________________

SEMESTER 2, 2018 EXAMINATIONS

COMP 2511

OBJECT ORIENTED DESIGN AND PROGRAMMING

(SAMPLE - PAPER)

1. TIME ALLOWED – 3 HOURS
2. READING TIME – 10 MINUTES
3. TOTAL NUMBER OF QUESTIONS – PART A (10), PART B (30), PART C (60)
4. ANSWER **ALL** QUESTIONS
5. TOTAL MARKS AVAILABLE – 100
6. MARKS AVAILABLE FOR EACH QUESTION ARE SHOWN IN THE EXAMINATION PAPER
7. NO MATERIALS MAY BE TAKEN INTO THE EXAMINATION ROOM

ALL ANSWERS MUST BE PROVIDED ONLINE EXCEPT WHERE THEY ARE EXPRESSLY REQUIRED TO BE DONE IN A SEPARATE SHEET OF PAPER. PENCILS MAY BE USED ONLY FOR DRAWING, SKETCHING OR GRAPHICAL WORK

PART A (Multiple Choice) – 10 marks

EACH QUESTION IS WORTH **1** MARK..

1. **Different parts of the code contain identical groups of variables (e.g., parameters connecting to a database". This symptom is a sign of which code smell?**
   a. Data Clumps
   b. Large class
   c. Long Parameter List
   d. Lazy class
   e. Inappropriate Intimacy

2. **The code smell "message chains" violates the design principle:**

   a. open closed principle
   b. single responsibility principle
   c. liskov's substitution principle
   d. program to a super-type, not an implementation
   f. principle of least knowledge

3. **A design pattern that promotes loose coupling of components is:**

   a. Decorator Pattern
   b. Visitor Pattern
   c. Observer Pattern
   d. Singleton Pattern

4. **In the context of threading in Java, choose the incorrect statement.**

   a. A java thread in a *new* state implies that the thread has been allocated and waiting for the method *start()* to be invoked
   b. A java thread in a *runnable* state implies that the thread can begin execution
   c. Call the *run()* method in the class that implements the interface *Runnable*, to put the thread in *runnable* state
   d. The approach of creating a Java thread by extending the Thread class is not usually recommended
   e. A thread in a blocked state implies that the thread is waiting for an event e.g. IO

5. **Which of the following is untrue about method overriding ?**
   a. Constructors cannot be overridden
   b. During method overriding, the overridden method in the sub-class can specify a weaker access modifier
   c. If a static method in the base class, is redefined in the sub-class, the later hides the method in the base class

d. In method overriding, run-time polymorphism ensures that instantiated, the call to any method in the base class will be resolved to the correct method, based on the run-time type of the object instantiated.

6. Which of the following is untrue about method overriding ?

a. The principle of least knowledge reduces dependencies between objects and promotes loose coupling

b. The code below is a good example of the principle of least knowledge:

```
Driver driver = car.getDriver()
Address driverAddress = driver.getAddress()
```

c. According to the principle of least knowledge, accessing the methods on objects returned by a method call is invalid

d. The principle of least knowledge states that accessing methods of objects passed in as parameters or instantiated inside the method is valid

7.

8.

9.

10.

Part B (Short Answer) – 30 marks

Note: The marks for each question varies.

Question 1 (6 marks)

a. Briefly explain the concept of encapsulation in OO design and its benefits
b. What does it mean for a class to be *immutable*? Explain why we choose for classes to be immutable (3 marks)

Question 2 (6 marks)

Name the design pattern you would use to make it easy to make the following changes and in each case draw a corresponding UML class diagram to describe this pattern

a. Change the algorithm that an object uses
b. Change the kind and number of objects that react to changes to the state of another object

Answer:
a. Strategy Pattern
b. Observer Pattern

Question 3 (6 marks)

a. Explain the terms code smell and design smell (2 marks)
b. Briefly describe the code smells "Divergent Change" and "Shotgun Surgery" (4 marks)

Answer:
"Divergent Change" is a code smell that occurs when you have to change a class in many different ways for different reasons. This is a violation of the single responsibility design principle.Any change to handle a variation should change a single class, and all the typing in the new class should express the variation.

- **To clean this up you identify everything that changes for a particular cause and use Extract Class to put them all together**

"Shot Gun surgery" is a code smell where one small change in the code forces lots of little changes to different classes.
  – **Use Move Method or Move Field to put all the changes into a single class**
Often you can use Inline Class to bring a whole bunch of behaviour together.

The above two code smells are the opposite of each other. Divergent change is one class that suffers many kinds of changes, and shotgun surgery is one change that alters many classes

Question 4 (6 marks) –

The Gof design patterns can be classified into three categories.

a. Name these categories
b. Describe the purpose of each category of patterns and give one example in each

Answer:
Refer tutorial 11 solution

Question 5 (6 marks)
-

Part C – 60 marks

Note: The marks for each question varies.

Question 1: Object Oriented Design Principles (10 marks)

(a) In the context of object oriented design, briefly explain the concepts of inheritance and composition. (4 marks)

(b) With the help of UML class diagrams, provide an example and explain why achieving reuse through the use of composition may be better than achieving reuse through inheritance. You do not need to provide any Java implementation for this question, but your class diagram must clearly show relevant fields/methods for your example. (6 marks)

Question 2: Object Oriented Design (15 marks)

Consider a university enrolments system with the following requirements.

- Students enrol in courses that are offered in particular semesters.
- Students receive grades (pass, fail, etc.) for courses in particular semesters.
- Courses may have prerequisites (other courses) and must have credit point values.
- For a student to enrol in a course, s/he must have passed all prerequisite courses.
- Course offerings are broken down into multiple sessions (lectures, tutorials and labs).
- Sessions in a course offering for a particular semester have an allocated room and timeslot.
- If a student enrols in a course, s/he must also enrol in some sessions of that course.

a. Draw a UML class diagram for an object-oriented system to implement the above requirements (6 marks)
b. Define a use case for a student enrolling in a course that has a prerequisite that s/he has passed (3 marks)
c. Define a walkthrough using a sequence diagram demonstrating how this would be handled in your system. (6 marks)

Question 3: Programming By Contract (15 marks)

a. Explain the meaning of the concepts *programming by contract* and *class invariant*
b. In the supplied code, we have provided an interface **Set<E>** for sets that can handle elements of a generic type **E**. Complete the class **ArrayListSet<E>** such that it uses an **ArrayList<E>** to store elements. Override the **equals()** method and provide an implementation. Do not modify the **Set<E>** interface.
c. Explain how your code enforces the class invariant that all elements of the set are distinct.

Question 4 (10 marks)

a. Name the code smell in the code fragments below. Name the refactoring techniques to be applied and rewrite the code with the necessary refactoring applied. (4 marks)

```
public static void main(String[] args) {
        int[] intArrayA = {1,2,3,4,5};
        int[] intArrayB = new int[5];

        int sum = 0;
        for (int i=0; i< intArrayA.length; i++) {
              sum += intArrayA[i];
        }
        System.out.println("Sum of ArrayA: " + sum);

        for (int i=0; i< intArrayB.length; i++) {
              sum += intArrayB[i];
        }
        System.out.println("Sum of ArrayB: " + sum);
   }
```

Answer:
Code Smell: Duplicate code – occurrence of same code structure more than once. Here, we have two for loops, duplicating the computation of the sum of an array of numbers. The solution for this code smell is to apply the refactoring technique “Extract method” which will encapsulate the computation and then invoke the code from both the places.

The revised solution is:

```
public static void main(String[] args) {
            int[] intArrayA = {1,2,3,4,5};
            int[] intArrayB = new int[5];

            int sum = computeSum(intArrayA);
            System.out.println("Sum of ArrayA: " + sum);

            sum = computeSum(intArrayB);
            System.out.println("Sum of ArrayB: " + sum);

    }

    private static int computeSum(int[] anArray) {
            int sum = 0;
            for (int i=0; i< anArray.length; i++) {
                  sum += anArray[i];
            }
    return sum;
    }
```

b. Examine the code snippet below. This code exhibits a typical problem that occurs when threads share memory and concurrently modify data. (6 marks)
(i) Briefly explain the problem in this code snippet
(ii) Re-write the code with a solution that fixes this problem. This code can be found in your submission folder as Counter.java

```
public class Counter implements Runnable {

   public static long count = 0;

   // This class implements Runnable interface
   // Its run method increments the counter three times
   public void increment() {

   // increments the counter and prints the value
   // of the counter shared between threads
         count++;
         System.out.println(Counter.count + " ");
   }
   public void run() {
        increment();
        increment();
        increment();
   }

   public static void main(String args[]) {
        Counter c = new Counter();
        Thread t1 = new Thread(c);
        Thread t2 = new Thread(c);
        Thread t3 = new Thread(c);
        t1.start();
        t2.start();
        t3.start();
   }

}
```

Answer:

The program above could result in a concurrent access problem known as race-condition. When two or more threads are trying to access a variable and modify them, at the same-time, this section of code that is commonly accessed and modified by more than one thread is known as the *critical section*. If this critical section is not protected to ensure that only thread can execute this critical section, it could result in a race-condition.

In the code above, the lines of code

```
        count++;
        System.out.println(Counter.count + " ");
```

constitute the critical section and these two lines of code i.e. the read and the write operation must be executed together (as an atomic operation). However, as the program above has no measures to ensure that these two operations are executed together, it can result in a race-condition, resulting in the incorrect update and read of the variable count. To avoid the race-condition, you ensure that the critical section is executed only by one thread through the use of the Java keyword *synchronised* that helps in thread synchronization.

The revised code with the use of the synchronized keyword is provided below, which ensures that the correct output is produced.

```
public void increment() {
   // These two statements perform read and write operations
   // on a variable that is commonly accessed by multiple threads.
   // So, acquire a lock before processing this "critical section"
   synchronized(this) {
   Counter.count++;
   System.out.print(Counter.count + " ");
   }
}
```

Question 5 (10 marks)

-

END OF EXAM PAPER