1. Choose the correct answer (within the context of Scrum).

   a. The client is responsible for communicating user stories directly to the team.
      i. True
      ii. False

   b. Which of the following is the responsibility of the Product Owner?
      i. Managing the product backlog
      ii. Communicating with the clients
      iii. Communicating with the development team
      iv. All of the choices are correct

   c. Standup meetings only occur when something urgent happens that might prevent the team from meeting the sprint goal.
      i. True
      ii. False

   d. What is missing from the following user story? "*I should be able to change my password*"
      i. Actor/object
      ii. Action
      iii. Result
      iv. Actor/object and result

   e. Which of the following is not the responsibility of the Scrum Master?
      i. Coordinating the meetings
      ii. Addressing obstacles
      iii. Coaching
      iv. Assigning tasks to the team members

2. For each of the following features, indicate whether it is promoted by the Agile approach:

   a. Delivery of working software at short duration intervals

   b. Customer collaboration

   c. Responding to change

   d. Detailed documentation

3. Will the following test pass? Explain.

```java
public class Presenter{
    private Model model;
    private View view;

    public Presenter(Model model, View view){
        this.model = model;
        this.view = view;
    }

    public void registerStudent() {
        String studentID = view.getStudentID();
        String courseCode = view.getCourseCode();
        if(!model.studentExists(studentID))
            view.displayMessage("student not found");
        else if(!model.courseExists(courseCode))
            view.displayMessage("course not found");
        else if(model.alreadyRegistered(studentID, courseCode))
            view.displayMessage("student already registered");
        else{
            model.register(studentID, courseCode);
            view.displayMessage("student registered successfully");
        }
    }
}
```

```java
@RunWith(MockitoJUnitRunner.class)
public class ExampleUnitTest {
    @Mock
    View view;

    @Mock
    Model model;

    @Test
    public void testPresenter(){
        when(view.getStudentID()).thenReturn("100");
        when(view.getCourseCode()).thenReturn("CSCB07");
        when(model.studentExists("100")).thenReturn(true);
        when(model.courseExists("CSCB07")).thenReturn(true);
        Presenter presenter = new Presenter(model, view);
        presenter.registerStudent();
        presenter.registerStudent();
        verify(view, times(2)).displayMessage("student registered successfully");
    }
}
```

default value ⇒ false.

4. Does the following code violate OCP? Explain.

```java
abstract class Shape{
    public double computeArea() {
        if(this instanceof Circle) {
            Circle c = (Circle)this;
            return c.radius * c.radius * Math.PI;
        }
        else if(this instanceof Square) {
            Square s = (Square)this;
            return s.side * s.side;
        }
        else
            return 0;
    }
}
```

```java
class Circle extends Shape{
    double radius;
}

class Square extends Shape{
    double side;
}
```

5. Does the following code violate LSP? Explain.

```java
class Fraction{
    int numerator;
    int denominator;

    public Fraction(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public void setNumerator(int numerator) {
        this.numerator = numerator;
    }

    public void setDenominator(int denominator) {
        this.denominator = denominator;
    }

    @Override
    public String toString() {
        return numerator + "/" + denominator;
    }
}
```

```java
//represents fractions whose numerator is 1
//e.g. 1/2, 1/3, etc.
class UnitFraction extends Fraction{

    public UnitFraction(int denominator) {
        super(1, denominator);
    }

}
```

6. Does the following code violate DIP? Explain.

facade

```java
interface Strategy {
    String format(int day, int month, int year);
}

class StrategyMDY implements Strategy{
    public String format(int day, int month, int year) {
        return month + "/" + day + "/" + year;
    }
}
```

```java
class Date {
    int day;
    int month;
    int year;

    public Date(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    @Override
    public String toString() {
        Strategy strategy = new StrategyMDY();
        return strategy.format(day, month, year);
    }
}
```

7. Which design pattern is used for the following code?

```java
class CPU{
    void process(){
    }
}

class Memory{
    void read(){
    }
    void write() {
    }
}
```

```java
class Computer{
    CPU c;
    Memory m;
    void compute() {
        m.read();
        c.process();
        m.write();
    }
}
```

8. Which design pattern is used for the following code?

```java
class Student{
    String name;
    Course course;
    public Student(String name, Course course) {
        this.name = name;
        this.course = course;
        this.course.add(this);
    }
    public void displayExamDate() {
        System.out.println(course.examDate);
    }
}
```

```java
class Course{
    List<Student> students;
    Date examDate;
    public Course() {
        students = new ArrayList<Student>();
    }
    public void addStudent(Student student) {
        students.add(student);
    }
    public void scheduleExam(Date examDate) {
        this.examDate = examDate;
        for(Student s:students)
            s.displayExamDate();
    }
}
```

9. Which design pattern is used for the following code?

```java
class Date {
    int day;
    int month;
    int year;
    DateFormatter formatter;

    public Date(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public void setFormatter(DateFormatter formatter){
        this.formatter = formatter;
    }

    @Override
    public String toString() {
        return formatter.format(day, month, year);
    }
}

interface DateFormatter{
    String format(int day, int month, int year);
}

class MDYDateFormatter implements DateFormatter{
    public String format(int day, int month, int year) {
        return month + "/" + day + "/" + year;
    }
}
```

Programming Exercise 1

You are required to develop and test code that handles academic information related to professors, courses, and students. The steps to be done are as follows:

1. Define a class **Person** as follows:

   a. It has a field of type **int** named **sin** (social insurance number) and another field of type **String** named **name**. Both fields are private.

   b. It has a public constructor that takes one parameter of type **int** and another one of type **String**, and initializes **sin** and **name** accordingly.

   c. It overrides **equals**. Two objects of type **Person** are equal if and only if their **sin** values are equal.

   d. It overrides **hashCode**.

   e. It overrides **toString** by returning **name**

   f. It has a public method **compareName** that takes an object of type **Person** as an argument and returns the result of comparing the name of the calling object with that of the argument. This method should throw an **IllegalArgumentException** if the argument is **null**. Note that you can compare the names using the **compareTo** method of class **String**.

   g. It should be defined such that it would not be possible to instantiate it using the **new** operator.

2. Define class **Professor** as follows:

   a. It inherits from **Person.**

   b. It doesn't have any fields.

   c. It has a public constructor that takes one parameter of type **int** and another one of type **String**, and initializes **sin** and **name** accordingly.

d. It overrides **toString** by returning the name of the professor as **"Prof. [name]"**. For example, if the name is "X", the returned value would be "Prof. X"

3. Define class **Student** as follows:

   a. It inherits from **Person**.

   b. It has four package-private fields

      i. **cgpa** of type **double** (representing the cumulative grade points average)

      ii. **inCSCPOSt** of type **boolean** (indicating whether the student in enrolled in a CSC subject POSt)

      iii. **passedCSCA48** of type **boolean** (indicating whether the student has passed CSCA48)

      iv. **passedCSC207** of type **boolean** (indicating whether the student has passed CSC207)

   c. It has a public constructor that takes the following arguments and initializes the fields accordingly: **int** (to initialize **sin**), **String** (to initialize **name**), **double** (to initialize **cgpa**), **boolean** (to initialize **inCSCPOSt**), **boolean** (to initialize **passedCSCA48**), **boolean** (to initialize **passedCSC207**)

   d. It overrides **toString** by returning the name and cgpa information as **"[name], cgpa: [cgpa]"**. For example, if the name is "Sam" and the cgpa is 3.6, the returned value would be "Sam, cgpa: 3.6"

   e. It implements **Comparable**. Students should be ordered by **name** and if the names are equal, they should be ordered based on the **cgpa** values (lowest first).

4. Define class **Course** as follows:

   a. It has three package-private fields

      i. **code** of type **String** (representing the code of the course, e.g. "CSCB07")

      ii. **professor** of type **Professor** (representing the instructor of the course)

      iii. **students** of type **ArrayList<Student>** (representing the students registered in the course)

   b. It has a public constructor that takes one parameter of type **String** and another one of type **Professor**, and initializes **code** and **professor** accordingly. It also initializes **students** to an empty **ArrayList<Student>**

   c. It overrides **equals**. Two objects of type **Course** are equal if and only if their **code** values are equal.

   d. It overrides **hashCode**

   e. It has a public method named **isEligibile** that takes an object of type **Student** as an argument and returns a **boolean** value indicating whether the student is eligible to be registered in the course or not. To implement this method, you need to use the CSCB07 eligibility criteria. The student must satisfy the following three conditions:

        i. Passed CSCA48

        ii. cgpa>=3.5 or enrolled in a CSC subject POSt

        iii. Did not pass CSC207

  f. It has a public void method named **addStudent** that takes an object of type **Student** as an argument and adds it to **students** if it satisfies the following conditions:

        i. The student is eligible to be registered in the course.

        ii. The student is not already added to the list of students. You can use method **contains** of class **ArrayList** to check if that is the case.

  g. It has a public void method named **displayInfo** that takes no arguments and displays the course information in the following order: code, professor, students. Note that the list of students should be sorted before being displayed (you can use **students.sort(null);** to achieve that).

Programming Exercise 2

Modify the code of "Programming Exercise 1" using the appropriate design pattern(s) as follows:

1. Define class **Administration** such that:
    a. It should not be possible to have more than one instance of this class
    b. It has three fields whose types ensure that no duplicates are allowed and that the elements are stored in the same order they are added
        i. **professors** (a collection of all the instantiated professors)
        ii. **students** (a collection of all the instantiated students)
        iii. **courses** (a collection of all the instantiated courses)
    c. It has three methods of type **void**:
        i. **addProfessor** that takes an object of type **Professor** as an argument and adds it to the collection of professors
        ii. **addStudent** that takes an object of type **Student** as an argument and adds it to the collection of students
        iii. **addCourse** that takes an object of type **Course** as an argument and adds it to the collection of courses
    d. Every time an object of type **Professor**, **Student**, or **Course** is instantiated in the code, it should be added to the appropriate collection. Note that you might need to modify other classes to achieve that.
2. The problem with method **isEligible** in class **Course** is that it is does not account for courses other than CSCB07. Even if it is to be used solely for CSCB07, modifying the eligibility conditions later on would require modifying class **Course**. As such, you are required to make the necessary changes so that any future modifications to the eligibility criteria could be done without modifying class **Course**.