

**Mohammad Usama**

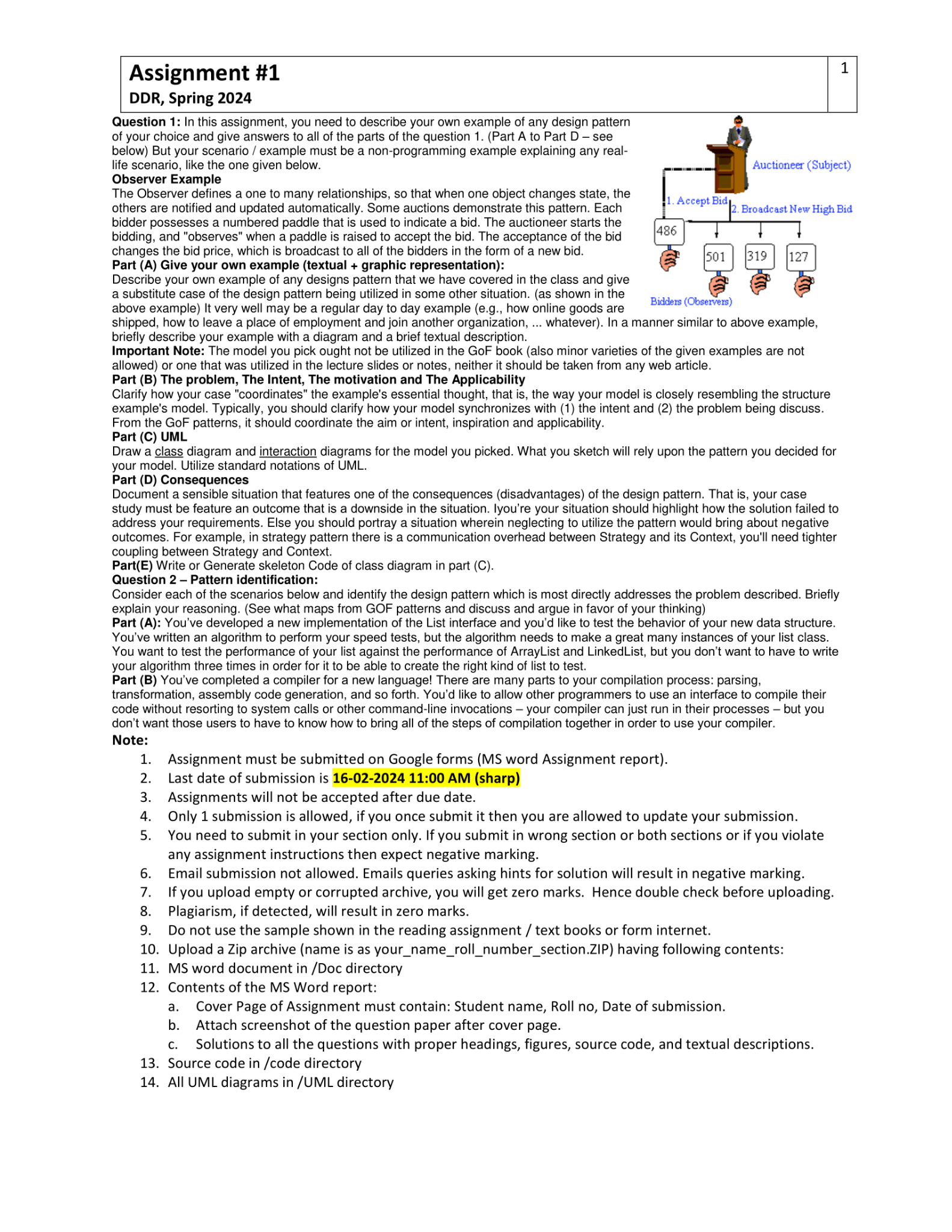
**20K-0190**

**Design Defects And Refactoring**

**Sec 8-A**

**Assignment 01**

**Date of submission: 16/02/2024**



**Question 01**

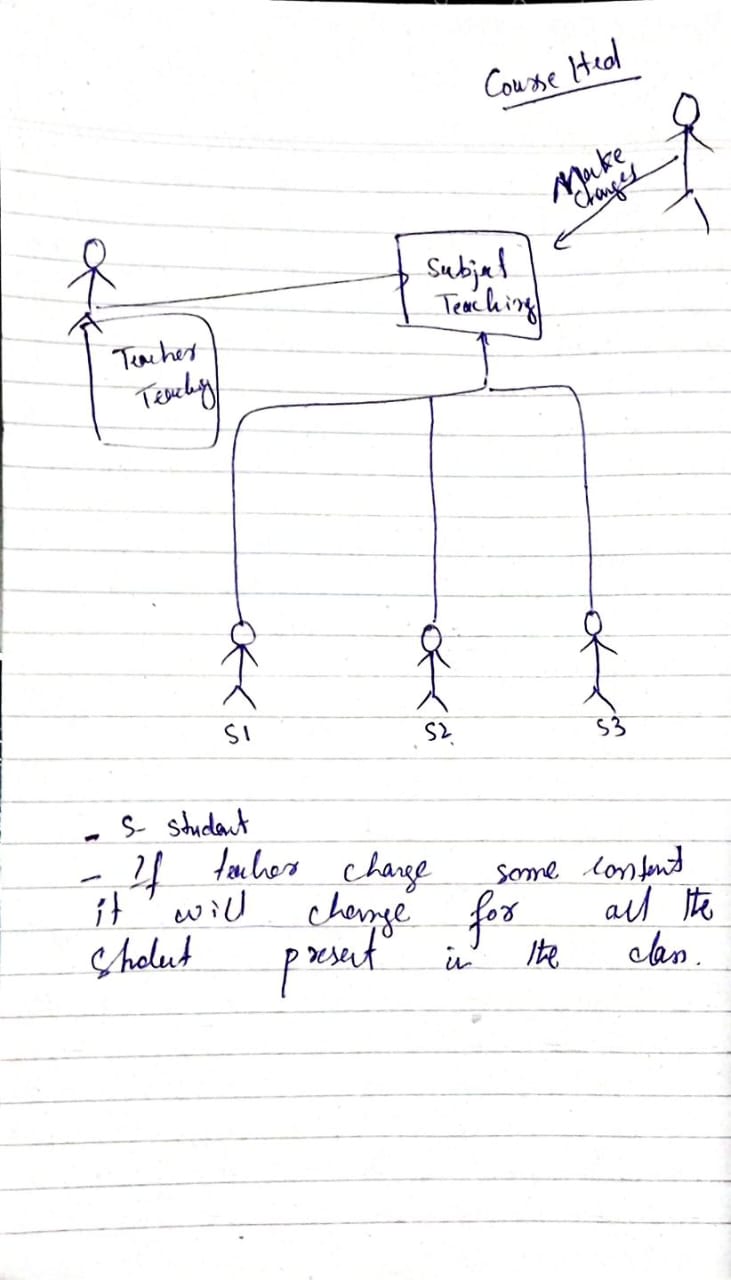
**Part (A) Give your own example (textual + graphic representation):**

**Observer Pattern**

The real life example of an observer pattern is a classroom scenario:

* The teacher represents the subject that is being observed.
* The students represent the observers.

When the teacher (subject) makes changes or takes actions, such as writing on the board, giving a lecture, or asking questions, all the students (observers) in the classroom are notified or made aware of these changes. This mirrors the observer pattern, where one object (the subject) changes state, and all the other objects (the observers) are notified and updated automatically.

**Graphic representation**

**Part (B) The problem, The Intent, The motivation and The Applicability**

**Intent:** Ensure that all students receive updates from the teacher without missing any information.

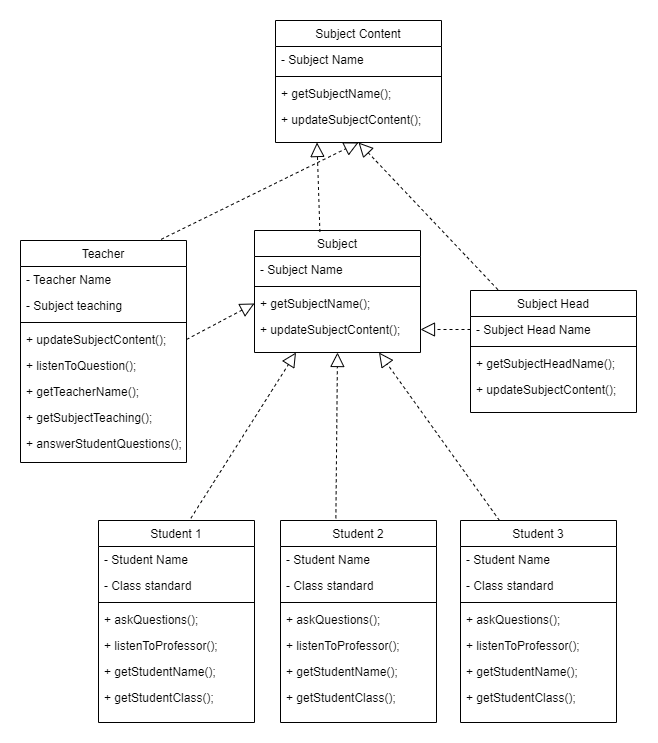
**Problem:** In traditional classrooms, it's challenging to ensure that every student stays informed about changes made by the teacher.

**Motivation:** The goal is to enhance the learning experience by keeping all students actively engaged and informed during class.

**Applicability:** This method is highly suitable for classrooms with large student populations where effective communication between the teacher and students is crucial.

**Part (C) UML**

**Class Diagram**



**Part (D) Consequences**

The observer pattern within the teacher-student setting helps reduce tight connections by enabling the teacher to share information without direct dependencies on individual students. However, it can also introduce complexities, such as the need for concrete class instantiation, potential confusion surrounding observer calls, and the possibility of intermingling concerns within the codebase.

**Part(E) Write or Generate skeleton Code of class diagram in part (C).**

abstract class SubjectContent {

protected String subjectName;

public String getSubjectName() {

return subjectName;

}

public abstract void updateSubjectContent();

}

class Subject extends SubjectContent {

@Override

public void updateSubjectContent() {

// Update subject content implementation

System.out.println("Subject content updated.");

}

}

class Teacher extends SubjectContent, Subject {

private String teacherName;

private Subject subjectTeaching;

public Teacher(String teacherName, Subject subjectTeaching) {

this.teacherName = teacherName;

this.subjectTeaching = subjectTeaching;

}

public String getTeacherName() {

return teacherName;

}

public String getSubjectTeaching() {

return subjectTeaching.getSubjectName();

}

public void listenToQuestions() {

System.out.println(teacherName + " is listening to student questions.");

}

public void answerToStudentQuestion() {

System.out.println(teacherName + " is answering student questions.");

}

@Override

public void updateSubjectContent() {

// Update subject content implementation for teacher

System.out.println("Subject content updated by the teacher.");

}

}

class SubjectHead extends SubjectContent, Subject {

private String subjectHeadName;

public SubjectHead(String subjectHeadName) {

this.subjectHeadName = subjectHeadName;

}

public String getSubjectHeadName() {

return subjectHeadName;

}

@Override

public void updateSubjectContent() {

// Update subject content implementation for subject head

System.out.println("Subject content updated by the subject head.");

}

}

class Student extends Subject {

private String studentName;

private String classStandard;

public Student(String studentName, String classStandard) {

this.studentName = studentName;

this.classStandard = classStandard;

}

public String getStudentName() {

return studentName;

}

public String getStudentClass() {

return classStandard;

}

public void askQuestion() {

System.out.println(studentName + " is asking a question.");

}

public void listenToProfessor() {

System.out.println(studentName + " is listening to the teacher.");

}

}

**Question 02 – Pattern identification:**

**Part (A):**

The Factory Method pattern facilitates object creation without specifying the exact class of the object. It defines an interface for creating objects while allowing subclasses to determine the specific type of objects to be created. In this scenario, the aim is to evaluate the performance of a custom list implementation against ArrayList and LinkedList without replicating the testing algorithm.

**Part (B):**

The Facade pattern offers a simplified interface to a subsystem's set of interfaces. It establishes a higher-level interface that streamlines the use of the subsystem. In this context, the compilation process involves multiple stages such as parsing, transformation, and assembly code generation. The goal is to enable other programmers to utilize the compiler without requiring an in-depth understanding of the intricacies of the compilation process.