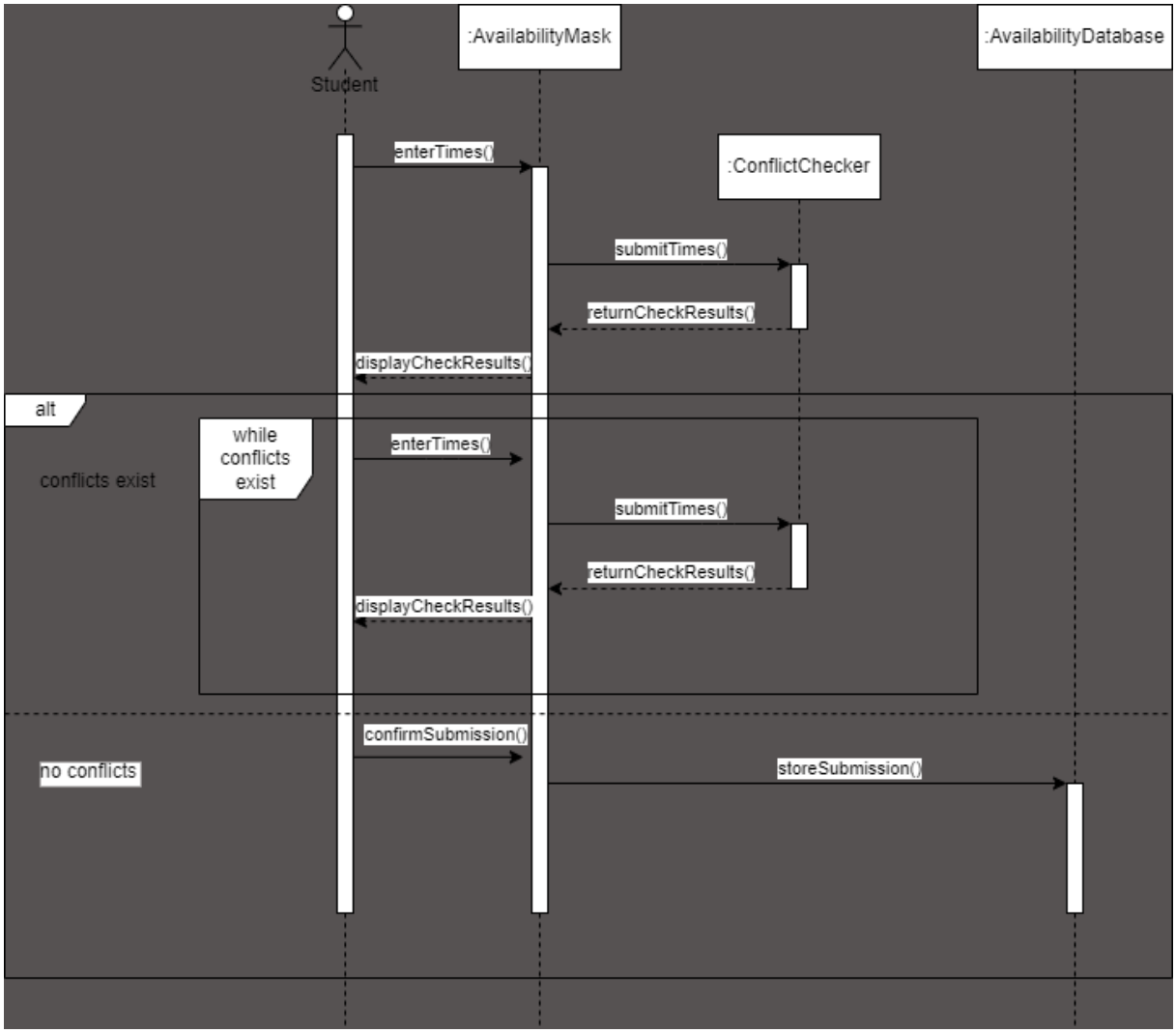
Please submit a single PDF file. That makes the correction much faster for us. Otherwise, I always have to generate a PDF file first. Thanks!

Tip: See pandoc for md -> pdf conversion.

Exercise 1



2a)

OCL for Class Invariants:

```
context CourseManager
inv noNullValues: courses->forAll(course | course <> null)
inv noDuplicates: courses->isUnique(course)
```

✓

Preconditions and Postconditions for addCourse:

Preconditions:

- The input `courseName` must not be null.
- The `courseName` must not already exist in the courses list.

Postconditions:

- If the method succeeds, `courseName` is added to the list.
- The size of the courses list increases by one.

OCL for addCourse:

```
context CourseManager::addCourse(courseName: String)
pre courseNotNull: courseName <> null
pre courseNotAlreadyInList: not courses->includes(courseName)
post courseAdded: courses->includes(courseName)
post sizeIncreased: courses->size() = courses@pre->size() + 1
```

✓

2b)

1. **Single Responsibility Principle (SRP): Violation**
   - **Problem:** The `CourseManager` class handles both course management logic and the storage mechanism for courses (`ArrayList<String>`).
   - **Fix:** Introduce a separation of concerns. Use a dedicated repository class or service to handle storage and retrieval of courses. For example:

     ```
     class CourseRepository {
           private List<String> courses;
           // Methods for adding, removing, and retrieving courses.
     }
     ```

     (✓)

     The one responsibility could be "managing courses", thus this wouldn't be violated.
2. **Open/Closed Principle (OCP): Adherence**
   - The class is open for extension (you can add new methods) but closed for modification (existing functionality need not change). This is due to proper encapsulation.

   f
   methods are final
   -0,5P.
3. **Liskov Substitution Principle (LSP): Adherence**
   - The `CourseManager` class does not inherit from another class, so LSP is trivially satisfied.

   ✓
4. **Interface Segregation Principle (ISP): Uncertain**
   - **Further Information Needed:** If `CourseManager` is part of a broader system and clients only need parts of its functionality (e.g., some only read courses, others modify them), this principle could be violated. A possible solution is to extract smaller interfaces like `CourseReader` and `CourseWriter`.

   ✓
5. **Dependency Inversion Principle (DIP): Violation**
   - **Problem:** The class depends on a concrete implementation (`ArrayList<String>`) instead of an abstraction.

   ✓

Σ 4,5/5P.

- **Fix:** Use the `List<String>` interface instead of `ArrayList<String>` for courses. This makes the code more flexible and adheres to DIP.

  ```
  private List<String> courses;
  ```

Improved Code

```
import java.util.ArrayList;
import java.util.List;

public class CourseManager {

    private final List<String> courses; // Use abstraction

    public CourseManager() {
        this.courses = new ArrayList<>();
    }

    /**
     * Adds a course to the list. It should only do so
     * if courseName is not null and the course is not
     * already in the list.
     *
     * @param courseName The name of the course to add.
     * @throws IllegalArgumentException if the courseName is null or
     ↪  already exists.
     */
    public final void addCourse(String courseName) {
        if (courseName == null || courses.contains(courseName)) {
            throw new IllegalArgumentException("Invalid course name");
        }
        courses.add(courseName);
    }

    /**
     * Removes a course from the list. It should only do so
     * if courseName is not null and the course is in the list.
     *
     * @param courseName The name of the course to remove.
     * @return true if the course was removed, false otherwise.
     */
    public final boolean removeCourse(String courseName) {
        if (courseName == null) {
```

```
            throw new IllegalArgumentException("Course name cannot be
            ↪  null");
        }
        return courses.remove(courseName);
    }
}
```

The Observer Pattern matches the requirements perfectly. This pattern decouples the subject (CourseDistributor) from its observers (Student, or other potential user classes). Observers can dynamically register, and the subject notifies them of changes.

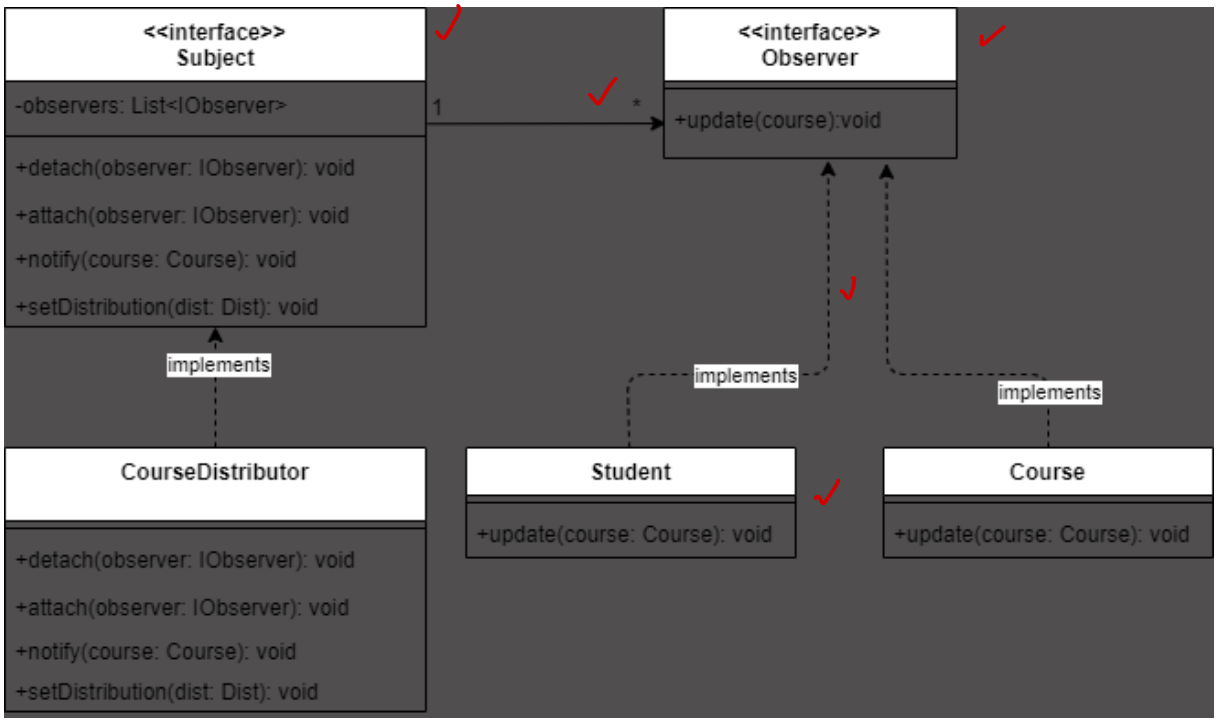


Figure 1: Alt text

Appropriate Design Pattern: Strategy Pattern

The Strategy Pattern is a perfect fit for this problem. It allows you to define a family of algorithms (in this case, course distribution paradigms) and makes them interchangeable. The strategy pattern enables you to:

- Dynamically select a specific course distribution paradigm (e.g., “availability-based” or “priority-based”).
- Add new paradigms in the future without modifying existing code.

Key Components of the Strategy Pattern

- **Strategy Interface**:
  - Defines a common interface for all distribution paradigms.
  - Each paradigm implements this interface.
- **Concrete Strategies**:
  - Specific implementations of the strategy interface (e.g., `AvailabilityBasedDistribution`, `PriorityBasedDistribution`).
- **Context Class**:
  - Maintains a reference to a Strategy object.
  - Delegates course distribution tasks to the selected strategy.

Proposed Design

1. **Classes and Interfaces**
   - `CourseDistributionStrategy` (Strategy Interface): Declares a method `distributeCourses()`.
   - **Concrete Strategies**:
     - `AvailabilityBasedDistribution`: Implements distribution based on availability.
     - `PriorityBasedDistribution`: Implements distribution based on priorities.
   - `CourseDistributor` (Context):
     - Maintains a reference to a `CourseDistributionStrategy`.
     - Allows lecturers to dynamically change the strategy via a setter method (`setStrategy`).
     - Delegates the `distributeCourses()` task to the current strategy.
2. **UML Class Diagram**