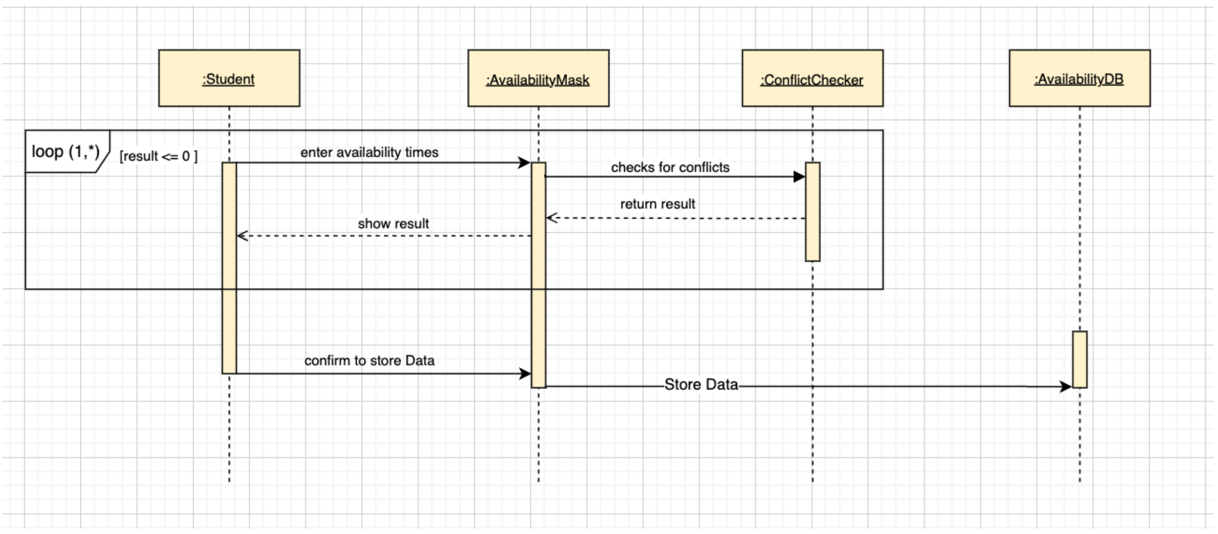
04_Homework
Ohad Ragolsky 7381351, Gruppe 4

Aufgabe1:



Aufgabe 2.a:

Object Constraint Language (OCL)

```
context Hashtable inv: numElements >= 0
context Hashtable::put(key,entry) pre: !containsKey(key)
context Hashtable::put(key,entry) post: containsKey(key) and get(key) = entry
```

Preconditions:

For the methode addCourse:

- The courseName should not be null.
- The course should not already be in the list.

context CourseManager::addCourse(courseName : String)
pre: courseName <> null and not courses->includes(courseName)

Postconditions:

For the methode addCourse:

- The new course shlould to be included in the list
- The List [course.Size()] should increase by 1.

context CourseManager::addCourse(courseName : String)
post: courses->includes(courseName) and courses->size() = courses@pre->size() + 1

Invariants:

For the CourseManager class:

- The course list should not contain any null values.

Context CourseManager inv: courses ->forAll (c | c <> null)

Aufgabe 2.b:

Single Responsibility Principle (SRP):
The CourseManager class seems to have a single responsibility—managing courses.
However, to fully comply, consider separating course validation logic from the course management functionality. No apparent violation

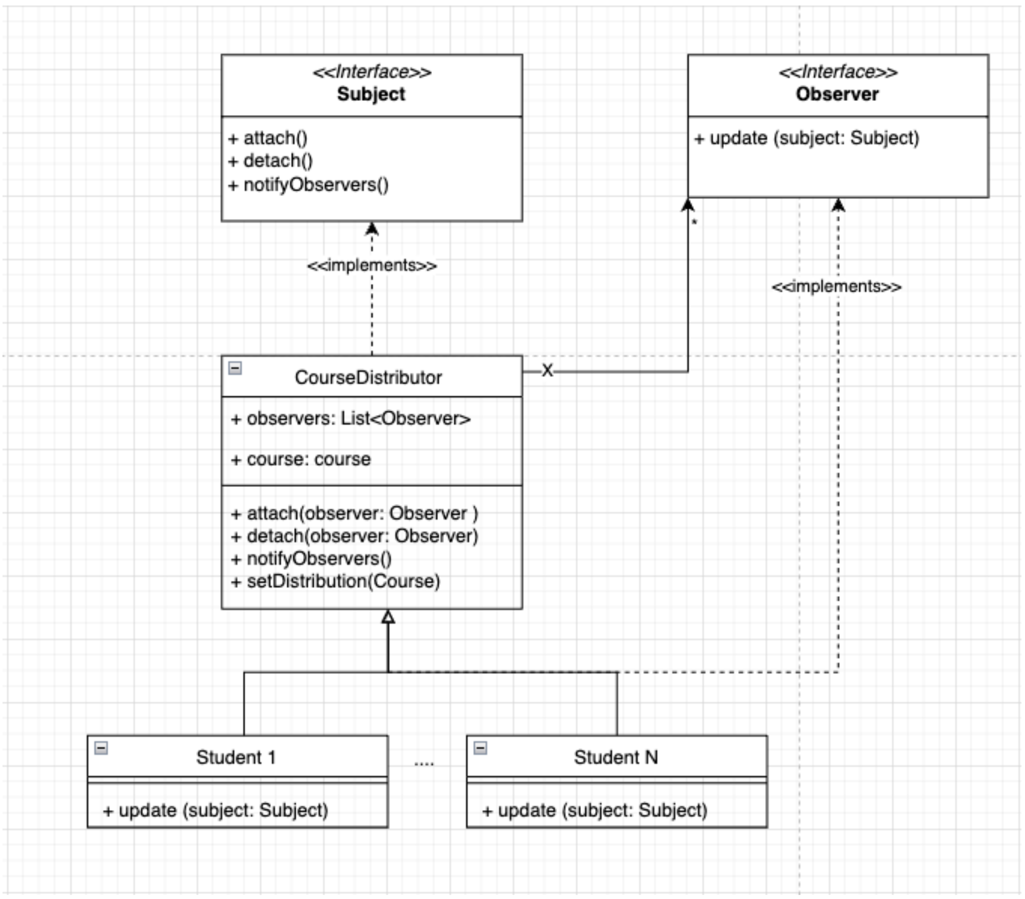
Open/Closed Principle (OCP):
The class allows for adding and removing courses. However, if the mechanism for storing or checking courses needs to change, you might need to modify the class.
For Example if we need to handle different types of courses differently, this class would require modification. we could handle this fby introducing a course interface and separate classes for different course types. It may violate OCP

Liskov Substitution Principle (LSP):
Since there are no subclasses, LSP does not directly apply here. Not applicable

Interface Segregation Principle (ISP): There are no interfaces in the given code.
To adhere more closely to the Interface Segregation Principle (ISP) in our CourseManager class, we could refactor the class to separate the responsibilities into more focused interfaces. This would allow different clients to implement only the interfaces that are necessary for their needs, rather than being forced to depend on a larger interface that includes methods they don't use.
No apparent violation

Dependency Inversion Principlve (DIP):
If a client uses the CourseManager, it might only need to add courses and not remove them or vice versa. However, since CourseManager has bundled these functionalities together, any client of this class must deal with both capabilities, even if only one is needed.
The class does not segregate these functionalities into smaller, more specific interfaces that clients could implement according to their needs.
The CourseManager class directly depends on a concrete implementation of ArrayList for storing courses. This represents a potential DIP violation because the class is directly dependent on a low-level module (ArrayList) instead of abstractions.
Fixing advice: Refactor the code to depend on an abstract List interface instead of directly on ArrayList. This change would make the class more flexible and easier to manage or test, as different types of lists could be injected if needed.

Aufgabe 3:
The Observer pattern allows an object, known as the subject, to notify other objects, known as observers, about changes in its state. This pattern is particularly useful for the scenario, where the course distribution system needs to inform participants about changes automatically.



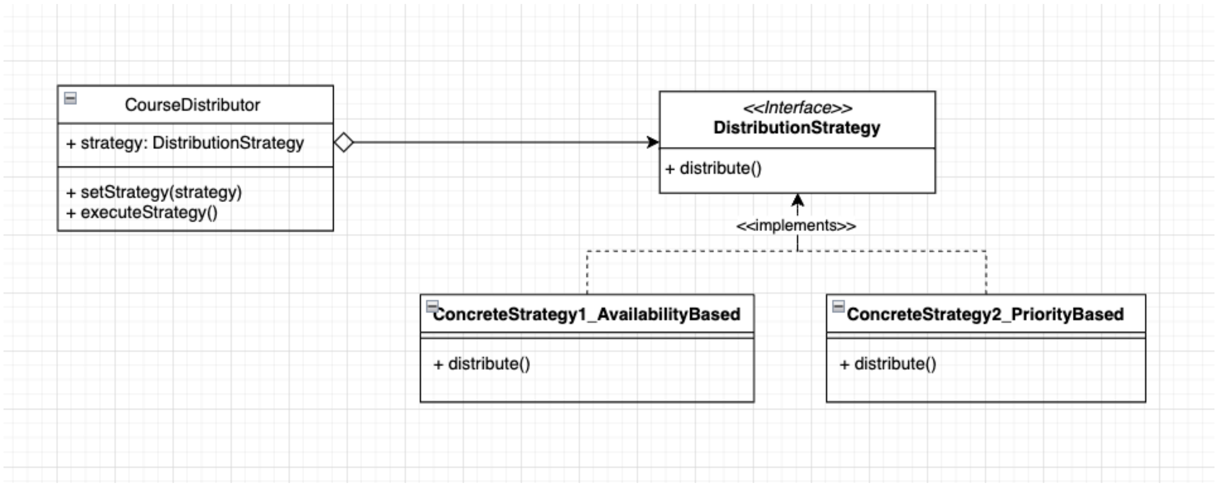
<<Interface>>
Subject
+ attach()
+ detach()
+ notifyObservers()
<<Interface>>
Observer
+ update (subject: Subject)
<<implements>>
*
<<implements>>
X
CourseDistributor
+ observers: List<Observer>
+ course: course
+ attach(observer: Observer)
+ detach(observer: Observer)
+ notifyObservers()
+ setDistribution(Course)
Student 1
+ update (subject: Subject)
....
Student N
+ update (subject: Subject)

Explanation of the Diagram:

- **Subject Interface**: Describes the operations that the subject must implement, including methods to attach, detach, and notify observers.
- **Observer Interface**: Defines the update method that every observer must implement to respond to notifications from the subject.
- **CourseDistributor Class**: Implements the Subject interface and includes methods to manage its list of observers, notify them of changes, and set the distribution of courses.
- **Students Classes**: Implement the Observer interface and handle updates from the CourseDistributor when the course distribution changes.

Aufgabe 4:

The Strategy pattern is ideal for situations where we need to dynamically alter the behavior of an application. It allows lecturers to change the distribution paradigm of courses, such as switching between "availability-based" and "priority-based" distributions. Future paradigms can be addable without code changes to existing systems, which indicates a need for a highly flexible system.


CourseDistributor
+ strategy: DistributionStrategy
+ setStrategy(strategy)
+ executeStrategy()
<<Interface>>
DistributionStrategy
+ distribute()
<<implements>>
ConcreteStrategy1_AvailabilityBased
+ distribute()
ConcreteStrategy2_PriorityBased
+ distribute()