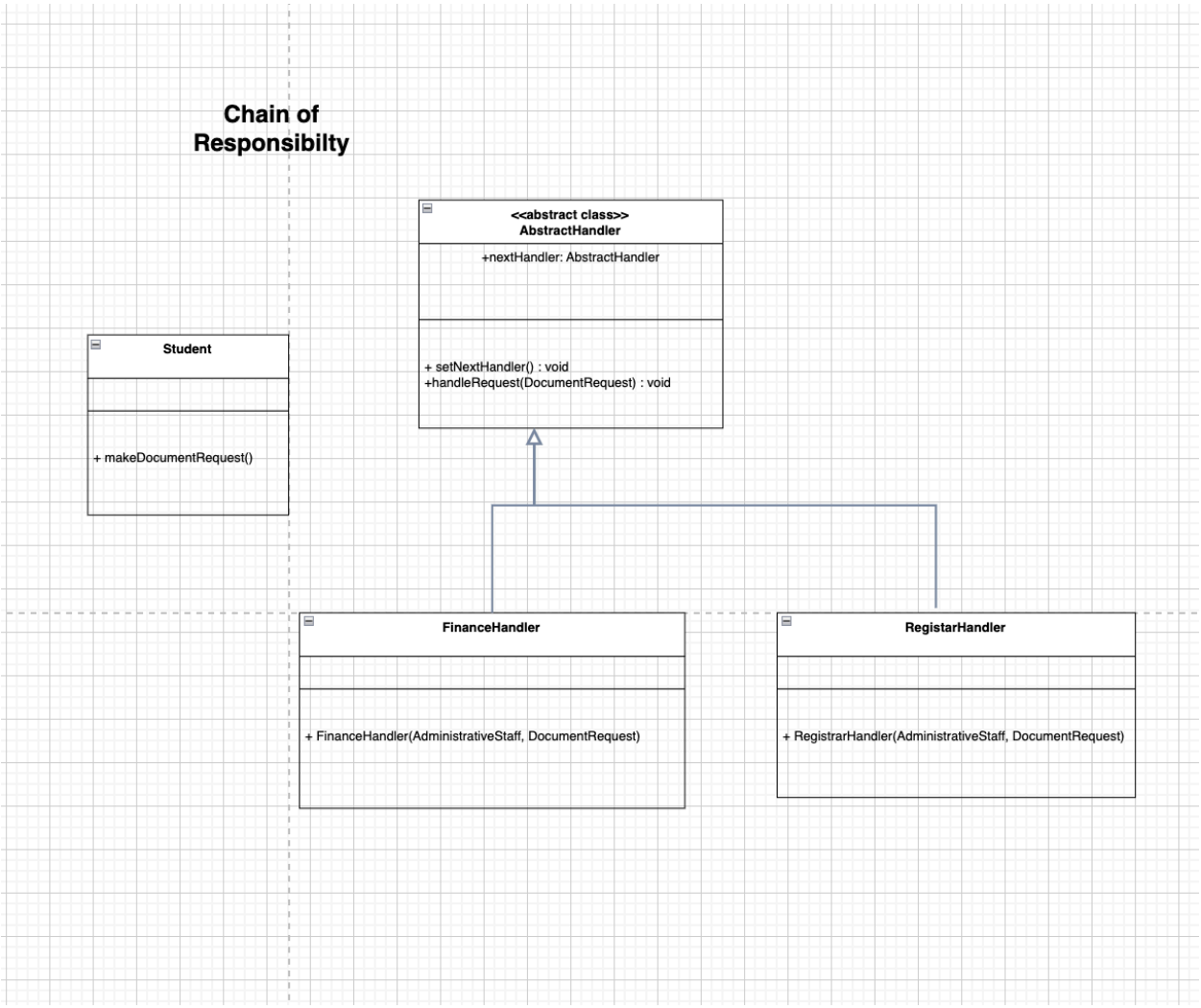
Chain of responsibility

Since the document request use case involves a sequential chain of events involving various actors, a chain of responsibility design pattern might simplify the code organization. A student makes a document request, starting a chain reaction of events: the request first goes to the Finance Office and after finance approval it goes to the Registrar's Office to get handled. So we have an AbstractHandler interface and the concrete FinanceHandler and RegistrarHandler.

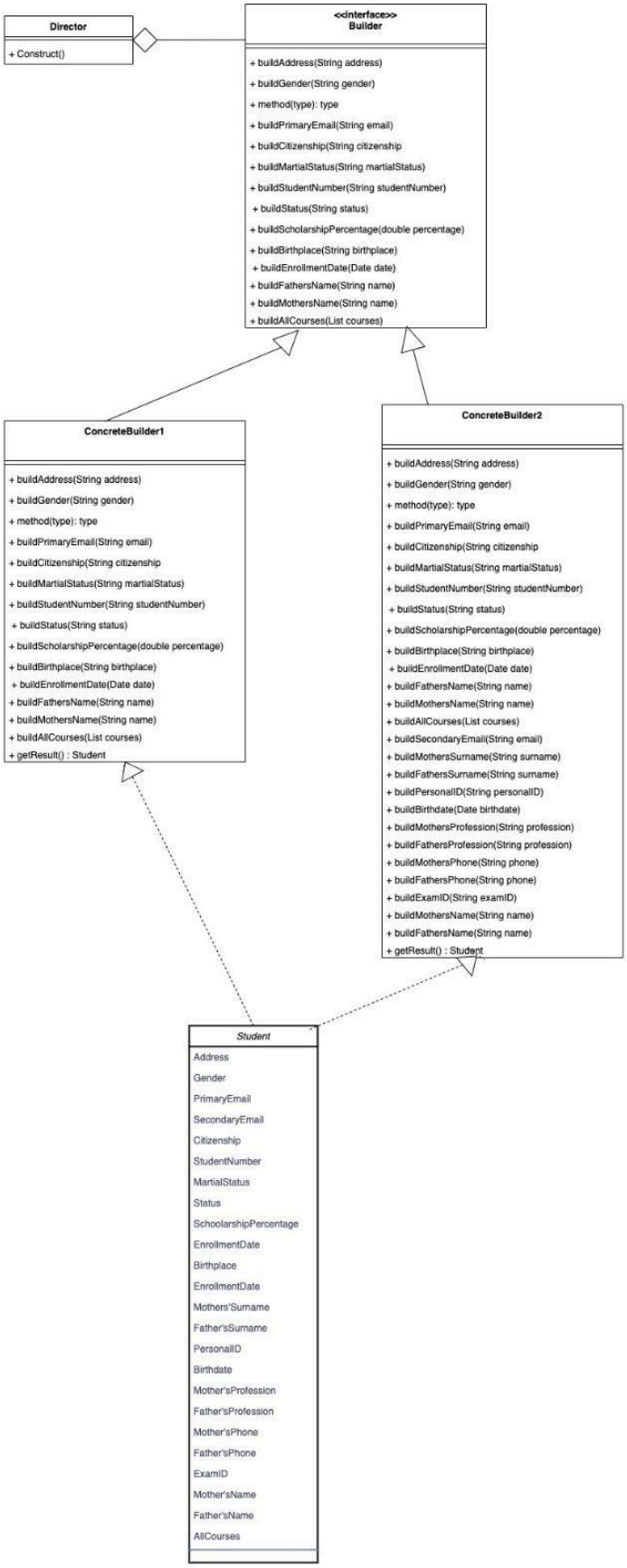


Builder

We have chosen to use the Builder design pattern for the Student class due to its extensive set of attributes. This design pattern is particularly advantageous in this context for several reasons:

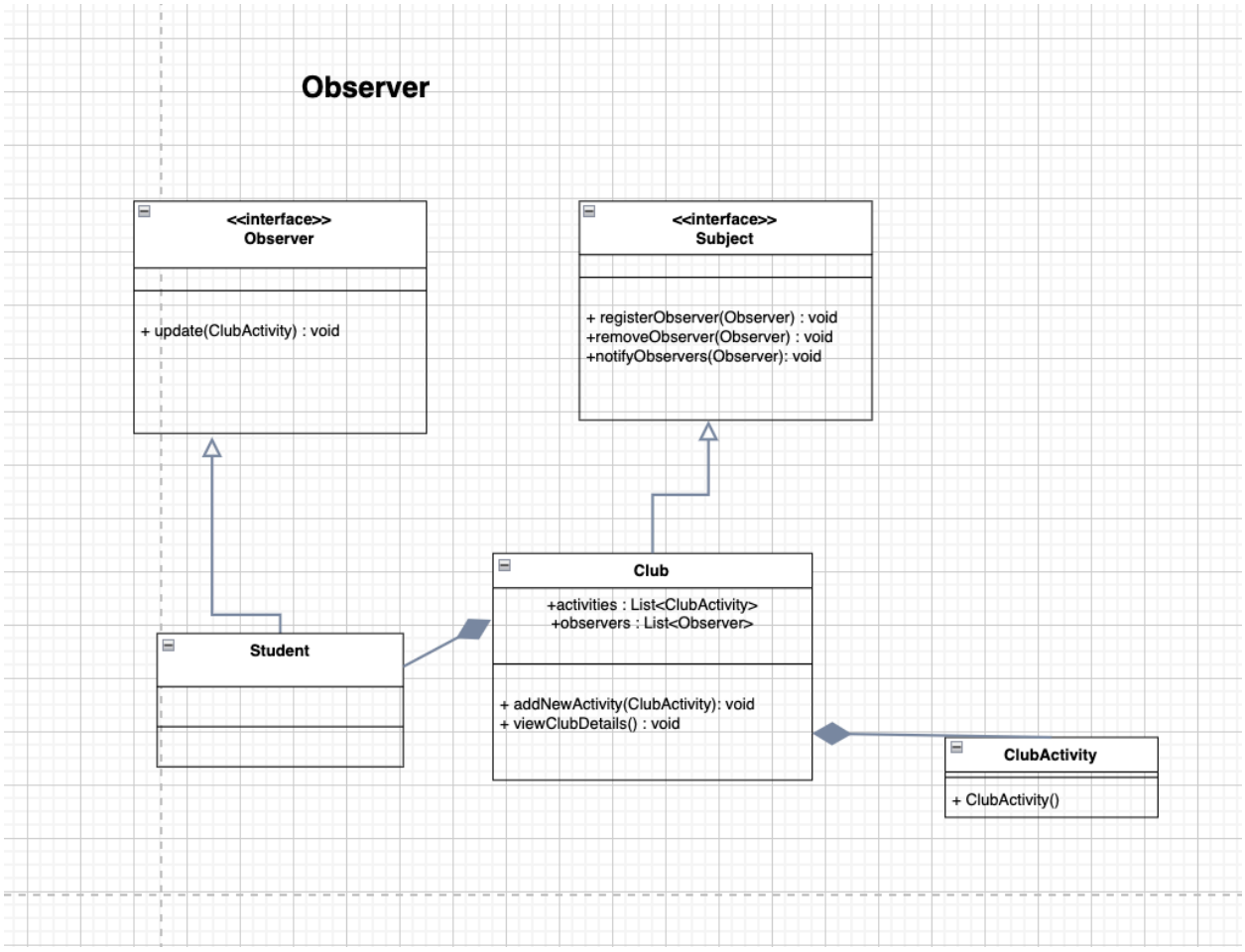
1. Readability and Maintainability: The Student class has a large number of attributes, which can make its constructor cumbersome and difficult to read. By using the Builder pattern, we can construct Student objects in a more readable manner, where each attribute can be set in a step-by-step fashion. This enhances the code's readability and maintainability.
2. Scalability: As the Student class evolves, additional attributes may be added. The Builder pattern provides a scalable approach to object construction, as new attributes can be incorporated into the builder without modifying the existing constructor, thus adhering to the open/closed principle.

Builder


Director
+ Construct()
<<interface>>
Builder
+ buildAddress(String address)
+ buildGender(String gender)
+ method(type): type
+ buildPrimaryEmail(String email)
+ buildCitizenship(String citizenship
+ buildMartialStatus(String martialStatus)
+ buildStudentNumber(String studentNumber)
+ buildStatus(String status)
+ buildScholarshipPercentage(double percentage)
+ buildBirthplace(String birthplace)
+ buildEnrollmentDate(Date date)
+ buildFathersName(String name)
+ buildMothersName(String name)
+ buildAllCourses(List courses)
ConcreteBuilder1
+ buildAddress(String address)
+ buildGender(String gender)
+ method(type): type
+ buildPrimaryEmail(String email)
+ buildCitizenship(String citizenship
+ buildMartialStatus(String martialStatus)
+ buildStudentNumber(String studentNumber)
+ buildStatus(String status)
+ buildScholarshipPercentage(double percentage)
+ buildBirthplace(String birthplace)
+ buildEnrollmentDate(Date date)
+ buildFathersName(String name)
+ buildMothersName(String name)
+ buildAllCourses(List courses)
+ getResult() : Student
ConcreteBuilder2
+ buildAddress(String address)
+ buildGender(String gender)
+ method(type): type
+ buildPrimaryEmail(String email)
+ buildCitizenship(String citizenship
+ buildMartialStatus(String martialStatus)
+ buildStudentNumber(String studentNumber)
+ buildStatus(String status)
+ buildScholarshipPercentage(double percentage)
+ buildBirthplace(String birthplace)
+ buildEnrollmentDate(Date date)
+ buildFathersName(String name)
+ buildMothersName(String name)
+ buildAllCourses(List courses)
+ buildSecondaryEmail(String email)
+ buildMothersSurname(String surname)
+ buildFathersSurname(String surname)
+ buildPersonalID(String personalID)
+ buildBirthdate(Date birthdate)
+ buildMothersProfession(String profession)
+ buildFathersProfession(String profession)
+ buildMothersPhone(String phone)
+ buildFathersPhone(String phone)
+ buildExamID(String examID)
+ buildMothersName(String name)
+ buildFathersName(String name)
+ getResult() : Student
Student
Address
Gender
PrimaryEmail
SecondaryEmail
Citizenship
StudentNumber
MartialStatus
Status
SchoolarshipPercentage
EnrollmentDate
Birthplace
EnrollmentDate
Mothers'Surname
Father'sSurname
PersonalID
Birthdate
Mother'sProfession
Father'sProfession
Mother'sPhone
Father'sPhone
ExamID
Mother'sName
Father'sName
AllCourses

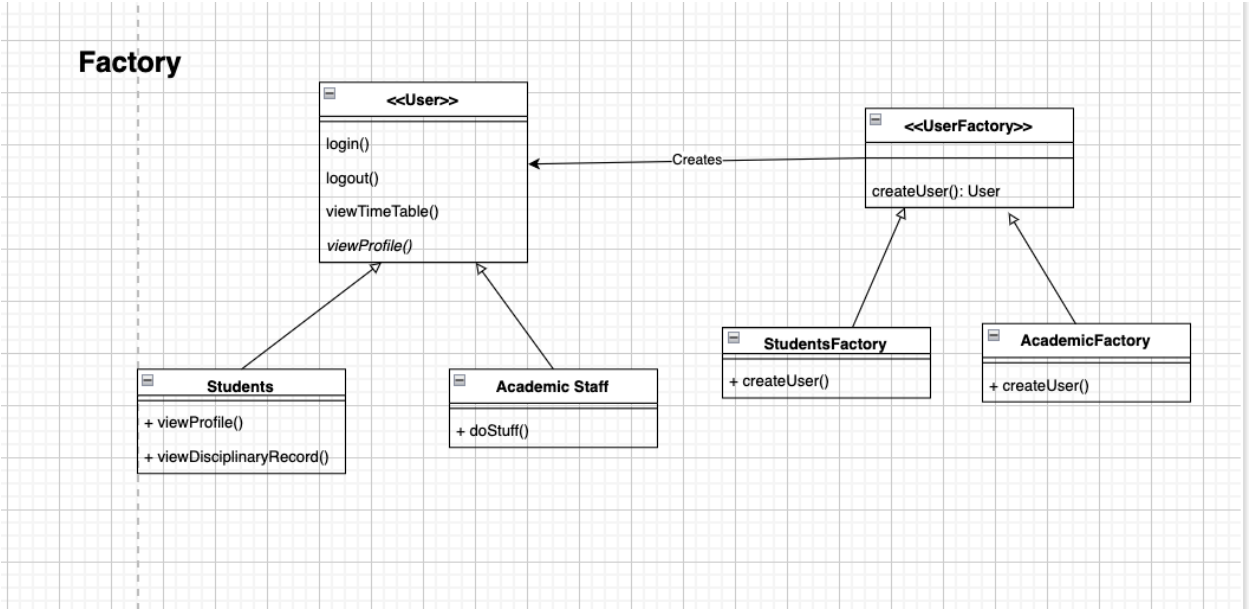
Observer

Students could get notified about new club activities from the clubs they are members of. We have the Observer interface and the Subject. In our case the concrete observer is Student and concrete Subject is Club. The club will hold a list of observers (student members of that club) and notify them when new activities are posted. We could also make AcademicStaff an Observer, making the club hold a list of student members and also staff members related to the club (club advisors for example)



Factory

The factory design pattern is used to provide an interface for creating objects in a superclass, allowing subclasses to alter the type of objects that will be created. This is useful in a school management system where different types of users (e.g., students, academic staff) may need to be created, each with its specific attributes and behaviors.



Proxy

We have implemented the Proxy design pattern because multiple users need to access and perform various operations on the same database. The Proxy pattern offers several key benefits in this context:

1. Controlled Access: By using a proxy, we can control and manage access to the database. This is essential when multiple users are involved, as it ensures that only authorized users can perform specific operations. The proxy can handle authentication and authorization checks before allowing any actions on the database.
2. Resource Management: The proxy can manage database connections efficiently. Instead of each user opening a new connection, the proxy can pool connections and reuse them, which improves performance and reduces the overhead associated with opening and closing database connections frequently