Task 1: Software Architecture

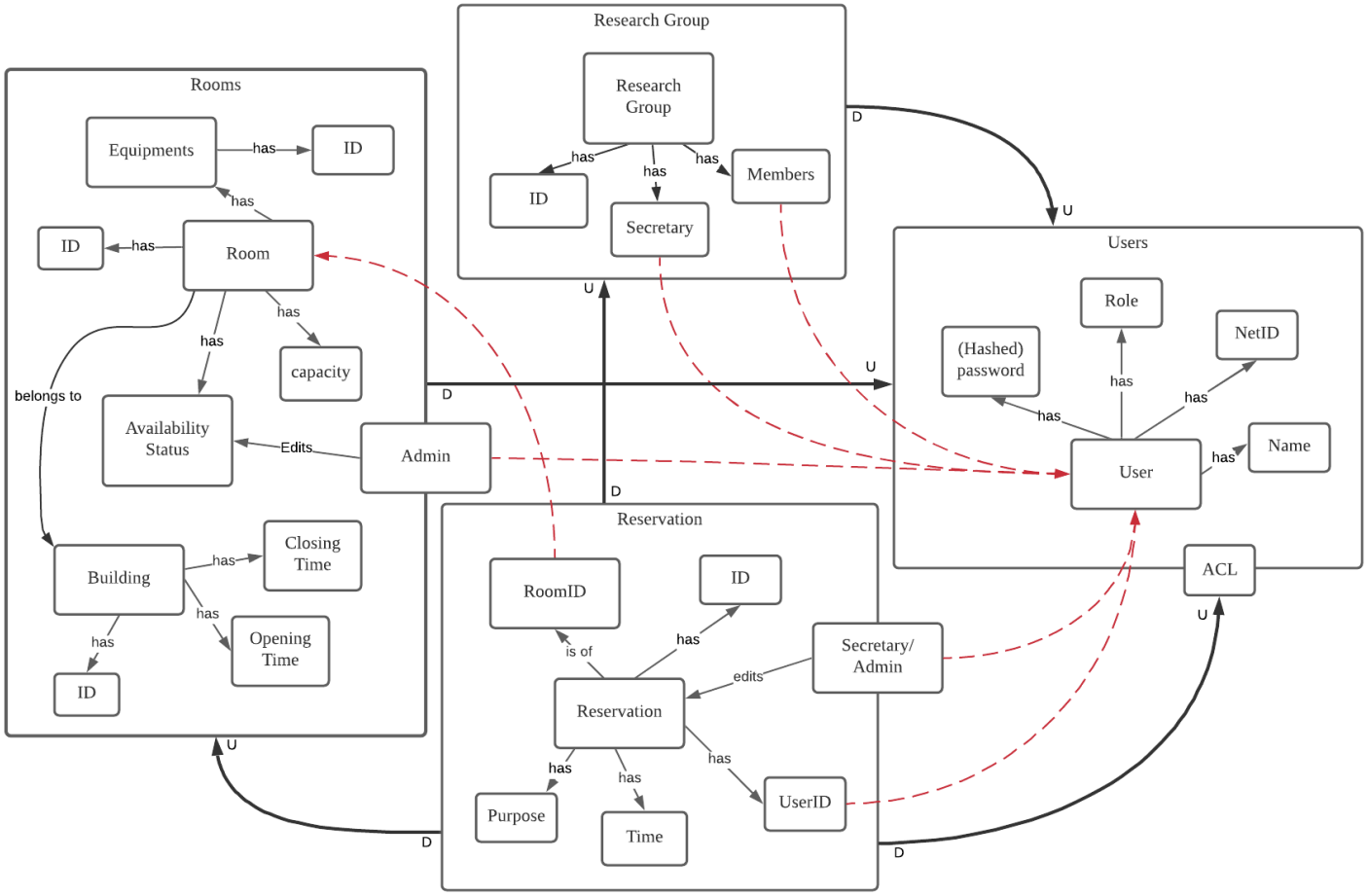
Domain-driven design

We started by identifying the domains for the bounded contexts using Domain-Driven Design (DDD):

- User (core domain)
- Authentication (generic domain)
- Reservation (core domain)
- Room (core domain)
- Research groups (core domain)

These domains are each responsible for a sizable portion of the program, and they are about suitably distinct parts.

Bounded Contexts Map



The map doesn't contain all the functionalities and attributes, but only the core functionalities of the application. Each domain has objects within the domain with some of the core attributes and the domains can interact with each other. The red arrows link the foreign keys in the domain. The boxes and arrows at the domain boundary signifies an action that can take place by communication. The arrows have a label in natural language to describe the relation between them which make the whole context map easier to interpret. Lastly, it has arrows between the domains to show the upstream and downstream between contexts. The map can also be viewed on Lucidchart.

Domain-Driven Design to Microservices

Having already identified the core domains, we thought it was logical to map each core domain to one microservice and combine the Authentication domain with Users as authentication mainly deals with users and their credentials.

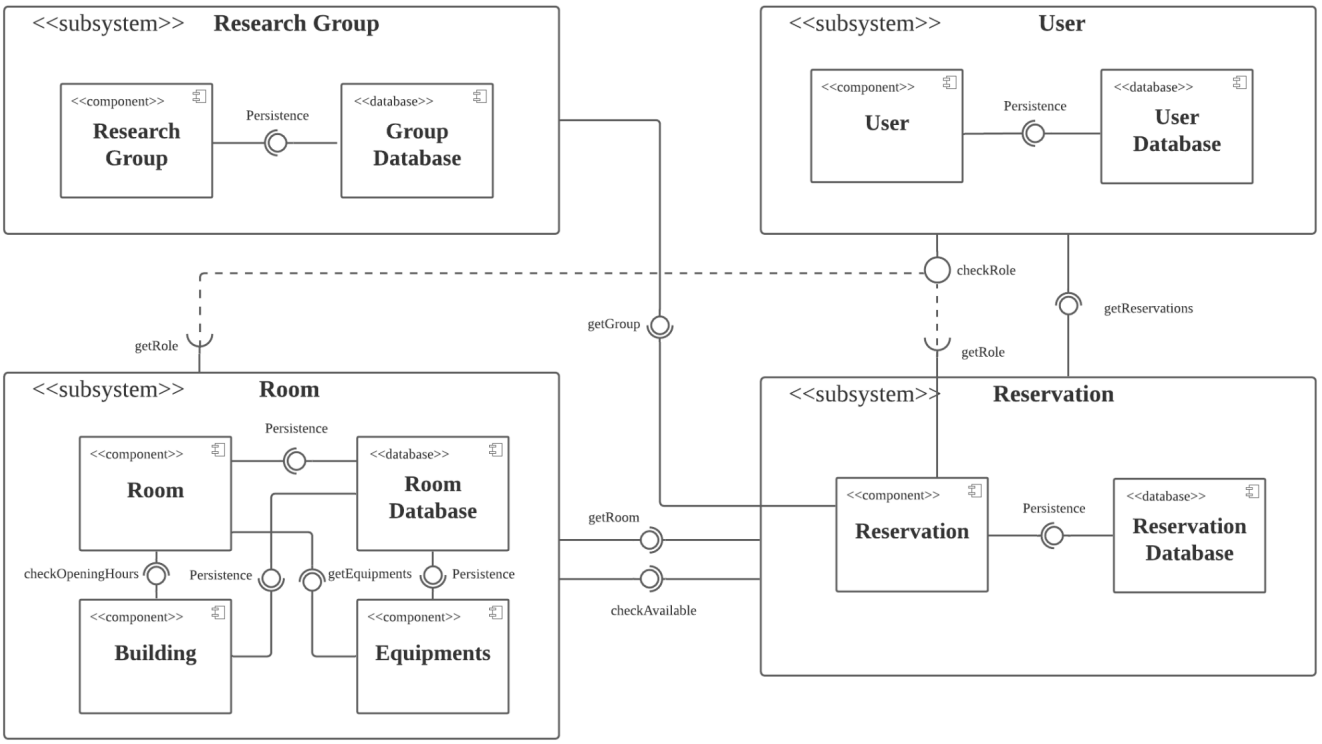
These are the core tasks of each of the microservices:

- User: Keep track of users with their netID, password, and other information including their roles
- Room: Deals with rooms and their capacity, equipment, building, opening time, etc.
- Reservation: Create and store reservations of rooms
- Research Group: Keep track of each microservice by storing the secretaries and the members of each group

The first three microservices were easily identified: User, Room, and Reservation. They are big domains on their own and they can be isolated easily. Then we decided we should use a fourth microservice with Research groups because a secretary can have multiple research groups and an employee can be in many research groups. It seemed too complicated to add more tables in the user database, since users already had many functions, so we separated the logic into another microservice.

Architecture Diagram (UML Component Diagram)

This is the UML component diagram of the application. We focused on illustrating the communication between the services and the databases. The diagram can be also viewed on Lucidchart.



User microservice

An upstream service which makes sure each user only has access to information and functionality intended for that user by confirming their identity and role, using their NetID and password. It makes sure they are stored safely by hashing the password and using Spring security.

It stores the user's NetID, (hashed) password, role, and name in the Users class. The user microservice is responsible for making sure the users have a correct access level according to their roles. The service also confirms a user's identity, because if a user tries to edit their own reservation, the system should only allow it when it is a reservation by the same user or the secretary of the research group. Most importantly, it handles login attempts by checking the correct combination of their netID and the password that is stored in the database.

Some of the core methods in the service:

- boolean isRole(netID, role): Checks if a user has a certain role (or higher)
- String getRole(netID): Returns the role of the specified user

Reservation microservice

A downstream service that handles the creation and management of the reservations. It is responsible for storing the ID, user, location (roomID), time and purpose. The service must also be able to display the time and location of all scheduled meetings for a given user. Furthermore, it enforces that a user cannot make a reservation more than two weeks in advance, that they cannot reserve two or more rooms in the same time slot, that they cannot cancel or edit reservations made by other users or make a reservation for other users, or for a room that is under maintenance or not within the opening hours of the building. However, it does provide some exceptional rights to the secretary and the system admin, more specifically a secretary can book multiple rooms and modify existing reservations made by their research group members, while the admin can make, edit and cancel other users' reservations at any moment. However, it does require secretaries to specify the employee for which they are making a reservation and provide a purpose. It also requires that the admin provides an explanation for the cancellation of a reservation.

Some of the core methods in the service:

- boolean makeReservation(int netID, int roomID, Time time, String purpose): Creates a new reservation according to the given data
- List<Reservation> getReservations(int netID): Returns all reservations of the given user

Room microservice

An upstream service that stores all the information about rooms and provides them when needed. It has a Room class that stores the room's id, name, building, equipment, capacity, and status. It also has a Building class, Equipment class, and a RoomNotice class. With these classes and the information, the service is responsible for determining if a room is available at a certain time. For the convenience of the user, it also allows searching for a particular room with their id to show details, as well as filtering with specific characteristics which are time-slot availability, capacity, equipment, or building. The service also allows admins to mark rooms as unavailable due to maintenance. Furthermore, a user can leave a message about the room (e.g. when there is a malfunction in the equipment).

Some of the core methods in the service:

- boolean getAvailable(roomID, startTime, endTime): Checks if the room is available at a given time by checking if the time is within the building's open hours and if the room's status is set to available, not under maintenance.
- Room getRoom(roomID): returns the room's information with the given roomId
- List<Room> getRooms(conditions): Returns all the rooms that satisfy the given conditions and is not under maintenance
- void changeStatus(roomID, boolean available): change the rooms' status to 'available' or 'under maintenance' (only available for admins)

Research Group Microservice

Encapsulates the n to n relationships between secretaries, research groups and their members. It is responsible for keeping track of the groupID, name and secretary for each research group. Then the members will be linked to the research group in a n to n relationship using a separate table in the database.

Some of the core methods in the service:

- boolean inTeam(secretary, team member): Checks if the team member is in a research group that the secretary is a secretary of.
- List<UserID> getMembers(secretary, team member): Returns all of the team members in all of the research groups of the secretary

Communication

Most of the communications taking place should be synchronous because it needs the information to further continue the process and we need to control the access level per role carefully. It is also synchronous between the service and the database.
Searching for a room:

- The reservation service provides a list of criteria to the room service. It will then return a list of rooms that match these criteria.
- The reservation service sends a roomId the room service. Then the room service will return the information of the room in detail.

Making/Editing a reservation:

- The reservation service sends a request to the room service with the desired time to check if the room is available at that time. Then the room service will respond with a boolean.
- If the user is trying to make a reservation for another user, the reservation service can query the research group service to check if the user is from the same research group.

Editing a reservation:

- If the user is not the user that made the reservation, the reservation service will check the role of the user by asking the User service.
- If the user service replies saying the user is a secretary, it will then query the research group service to check if the reservation the secretary is trying to edit is from a user in the same research group.

Changing a room status:

- If a user is trying to change the status of the room, the room service will make a request to the user service to check if the user is an admin.

Viewing the schedule:

- The user service can ask the reservation service to view all reservations for the user asking. The reservation service will reply with a list of reservations

Security

In order to secure our application, we have a main gateway where all requests go to, before then being redirected to the corresponding microservice. This gateway is where Security is implemented, and allows for some endpoints (such as /auth and /user/signup) to be accessed without authentication, whereas all other endpoints will need an authenticated user. After a user sends a POST request to /auth with a netId and password, if the information is valid, a JWT token will be returned. Attaching this JWT token to the header of requests as ‘Bearer <jwt token>’ will suffice. As for the hashing of the passwords, the passwords stored in the database will be hashed using the BCryptPasswordEncoder. With this encryption, it will be easy to tell if a provided text matches one of these hashed passwords, but the

original password would be almost impossible to obtain. When the request for authentication is sent, the e provided password will be hashed to check for a match on the database given the netID.

Task 2: Design patterns

Since we thought the reservation class is very complex and the process of making it should be secure, we decided to apply both design patterns in the Reservation microservice. So we will be creating a Reservation using the Builder, and then call the chain of responsibility validators to check that the reservation is valid, then save it in the repository.

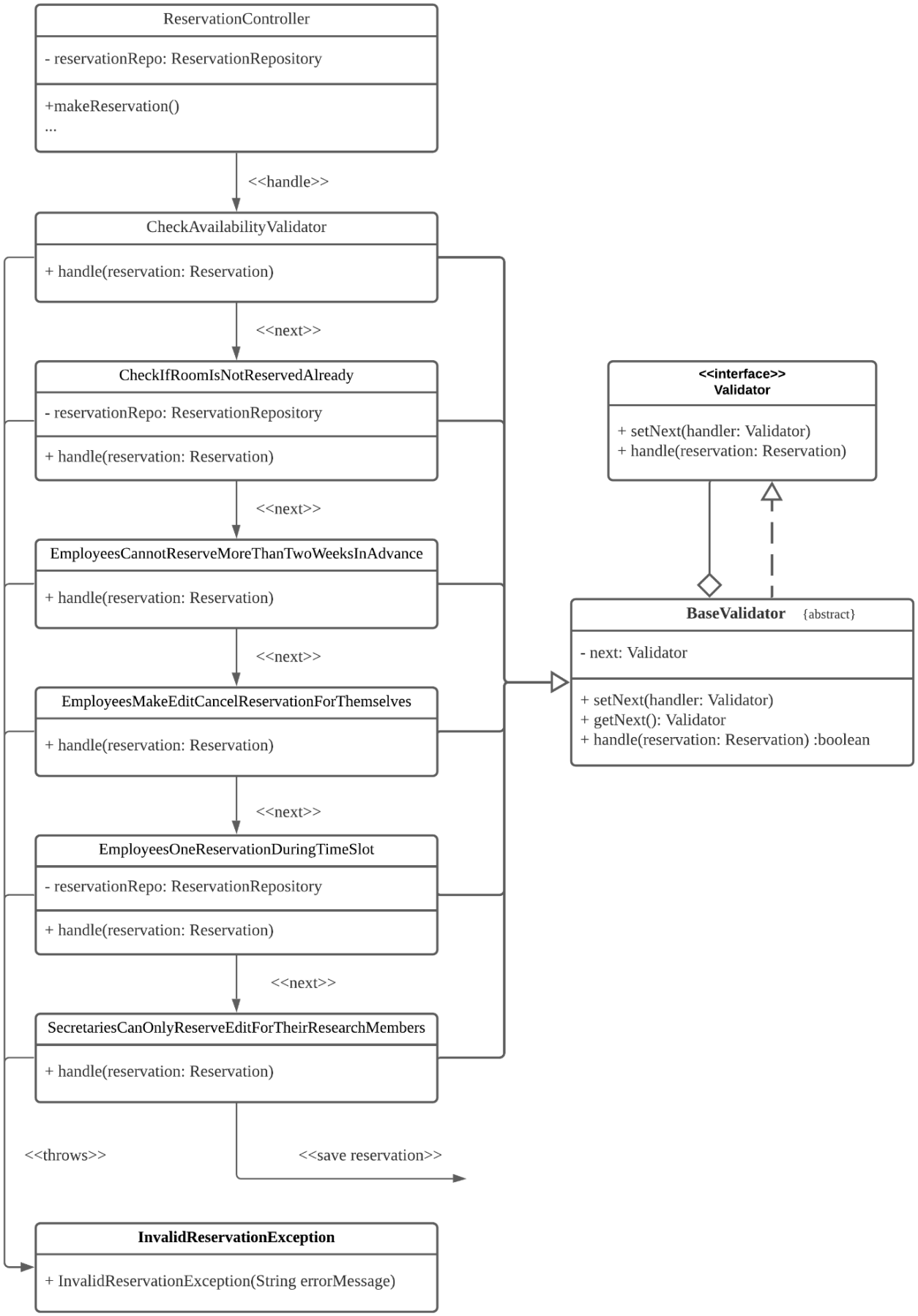
Design pattern 1 : Chain of Responsibility

First it was easy to choose the Chain of responsibility pattern as our first pattern because making a reservation requires many checks within the service and from other services. We thought using the validators would make the process more transparent and easy to manage. In the validators folder in the reservations service, we have many validators that check conditions in order like the room's availability, user's schedule etc. and they return exceptions accordingly.

We have an interface for the Validator which has the setNext() method and the handle() method. Then we have the Base Validator that implements the Validator interface. The Base Validator is an abstract class for the actual Validators, and it has the attribute next which is the next Validator in the chain, and the setNext() to set it. It also has the checkNext() method to pass it on to the next validator if there was no exception thrown. We have 6 actual Validators and all of them override the method handle() in the Validator interface. The handle() method is the most important part as it has the actual logic for checking the conditions.

UML Class Diagram for Chain of Responsibility

The diagram can also be viewed on LucidChart.



Design pattern 2: Builder pattern

For the second design pattern, we chose the Builder pattern. Reservation is a complex object because it has many many fields, some are optional, and also has types such as the reservations that the user makes for themself or the reservations that the secretary makes. By using the Builder pattern we can create the different types through one construction process.

We have an interface for the Builder which has several methods that are each used to set one of the attributes of the reservation that is to be created. It also has a method that takes all the attributes the Builder has been given and returns a Reservation with those attributes. The ReservationBuilder class implements the Builder interface, and it has the attributes that can be set for a reservation at creation. The other attributes are only used if the Reservation is edited in the future. The ReservationBuilder has to be initialized with basic attributes, to make sure reservations cannot be created with missing values. The Director class uses the Builder to create a Reservation with preset values, although some types of Reservation require additional attributes.

UML Class Diagram for Builder
The diagram can also be viewed on LucidChart.