Software Architecture Document

Version 1.2

for

Skynet Room Reservation System

Prepared by

GABRIELE BAVARO	27399103	GABRIELE.BAVARO@BELL.NET
EL-MEHDI BEGHDADI	26781276	ELMEHDI.BEGHDADI@GOOGLE.COM

INSTRUCTOR:	*DR. C CONSTANTINOS*
COURSE:	SOEN 343
DATE:	NOVEMBER 25, 2016

<Project name>	Version: 1.0
Software Architecture Document	Date: November 10, 2016

Document history

Date	Version	Description	Author
November 10, 2016	1.0	Started work on SAD document	Gabriele Bavaro Tian Li Medhi
November 12, 2016	1.1	Made Sequence Diagram for makeReservation and addToWaitlist	Gabriele Bavaro Tian Li Medhi
November 19, 2016	1.2	Updated SDs	Gabriele Bavaro Tian Li Medhi

TABLE OF CONTENTS

1. Introduction

Purpose

This Software Architecture Document (SAD) is used to provide a comprehensive architectural overview of the system. The audience of this document is mainly for designers to design the software and for the developers that are implementing the application. It is intended to capture the significant architectural decisions which have been made on the system. It will first show architectural requirements including the goals and constraints (functional and non-functional). This document will also use two different types of architectural view to show different aspects of the system. First, it will show the Use case view and then the Logical View of the application.

Scope

This SAD provides an architectural overview of the Room reservation application created by the team Skynet. The product is developed for Concordia University, a university based in Montreal, Quebec. This document helps designers to generate ideas and create different software architectural designs. It also help for developers to implement the application. This document is highly influenced by the Software Requirement Specification Document (SRS). The design is based from the requirements of the stakeholders.

Definitions, acronyms, and abbreviations

RUP: Rational Unified Process

UML: Unified Modeling Language

SAD: Software Architecture Document

SRS: Software Requirements Specification

2. ARCHITECTURAL REPRESENTATION

1. **Logical view** : Audience: Designers. The logical view is concerned with the functionality that the system provides to end-users. UML Diagrams used to represent the logical view include **Class diagram**, and **interaction diagrams** (**communication diagrams**, or **sequence diagrams**).

2. **Use case view** (also known as Scenarios) : Audience: all the stakeholders of the system, including the end-users. The description of an architecture is illustrated using a small set of use cases, or scenarios. The scenarios describe sequences of interactions between objects, and between processes. They are used to identify architectural elements and to illustrate and validate the architecture design. They also serve as a starting point for tests of an architecture prototype. Related Artifacts : **Use-Case Model**.

3. Architectural requirements: goals and constrains

Requirements are already described in SRS. In this section we describe *key* requirements and constraints that have a significant impact on the architecture.

FUNCTIONAL REQUIREMENTS (USE CASE VIEW)

The key requirements of the system that affect its architecture are:

Create reservation: A user shall be able to create a reservation.

Cancel reservation: A user shall be able to cancel his/her reservation.

Data persistence for these two key requirements will be handled by the use of a relational database.

These key requirements affect the design of the architecture of the system as a pattern will be used to promote reuse of classes through a separation of concerns and a layered architecture to access the data services layer.

The main constraint is:

- Rooms can only be accessible for write activities by one user at a time but are not restricted for read operations.

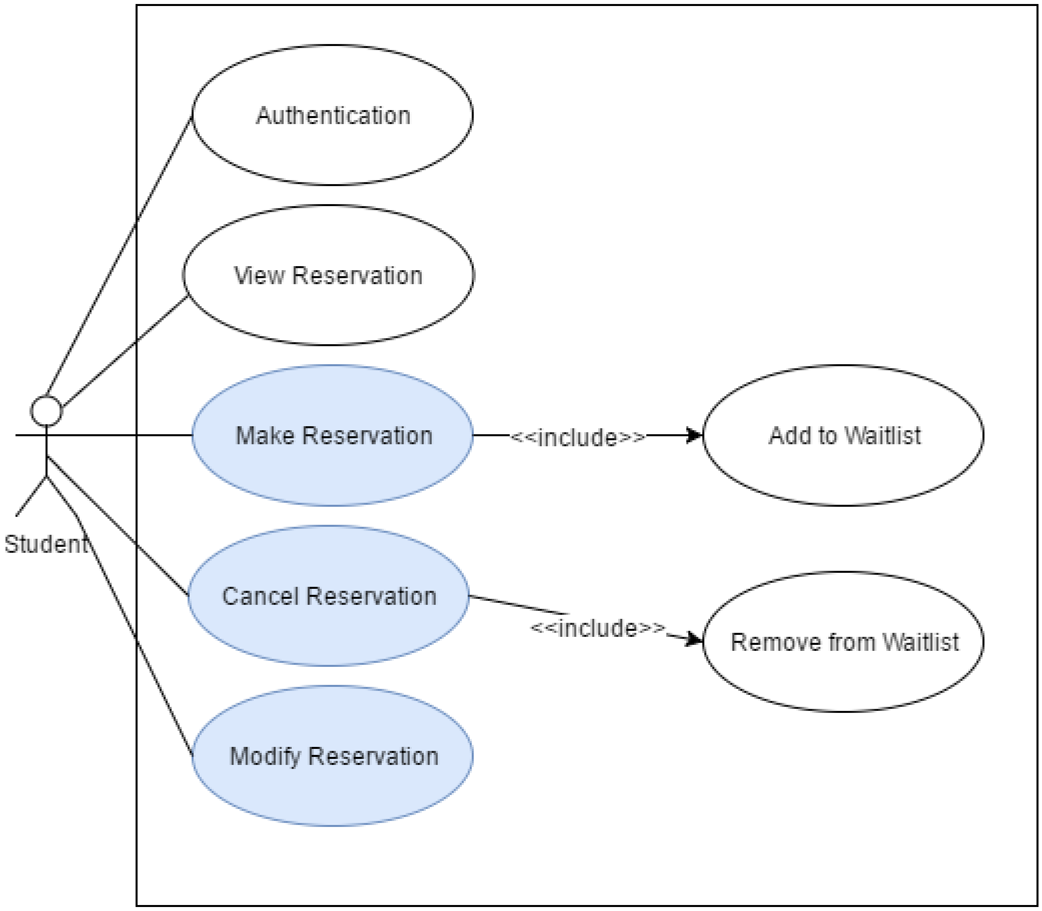
It affects the architecture of the system as it has to be fair to users who start a transaction first.

Functional requirements (Use case view)

The overview below refers to architecturally relevant Use Cases from the Use Case Model (see references).

Source	**Name**	**Architectural relevance**	**Addressed in:**

Use Case 1	Make Reservation	This requires persistence of data and thus the use of a relational database and the use of an architectural pattern.	Section 6 of this document.
Use Case 2	Cancel Reservation	Same reason as above.	Section 6 of this document.



NON-FUNCTIONAL REQUIREMENTS

The architecturally relevant non-functional requirements, i.e. those which are important for developing the software architecture.

Think of security, privacy, third-party products, system dependencies, distribution and reuse. Also environmental factors such as context, design, implementation strategy, team composition, development tools, time to market, use of legacy code may be addressed.

Source	**Name**	**Architectural relevance**	**Addressed n:**
Section 3 of SRS	Performance efficiency: Resource utilization	The system back-end: Wamp Server(Apache web server, MySQL database) will need to connect to the Spring framework (Java language)	Section 5 of this document.
Section 3 of SRS	Compatibility: Co-existence	Android front-end mobile application shall co-exist with the Spring backend framework	Section 5 of this document.

Section 3 of SRS	Maintainability: Modularity	Separate responsibilities and lower coupling.	Section 5 of this document.
Section 3 of SRS	Maintainability: Reusability	Multi-layer architecture to allow main domain classes to be reused	Section 5 of this document.
Section 3 of SRS	Security: Integrity	The system shall be safe and fair live to every user.	Section 5 of this document.
Section 3 of SRS	Security: Confidentiality	System shall not disclose the identity of the room holders to other users	Section 5 of this document.

4. Logical view

The logical view captures the functionality provided by the system; it illustrates the collaborations between system components in order to realize the system's use cases. Describe the architecturally significant logical structure of the system. Think of decomposition in tiers and subsystem. Also describe the way in which, in view of the decomposition, Use Cases are technically translated into Use Case Realizations.

In this section, the architecturally significant logical structure of the system is described by mappers, units of work (UoW), TDG and databases. The mappers handle communication between the classes that handle the logic of the system and the classes that interact directly with the database and its tables. The UoW keeps track of the changes made to the database by each class inside the system. This ensures that there is no accidental data corruption when two or more classes access the same information from a table inside the database. The TDG handles interactions between the domain objects and the database. Finally the databases store all the necessary and relevant information that needs to persist for the system to run properly such as login information and room information.

The above is accomplished through several different layers which handle different aspects of the overall system. More of this will be explained below in the LAYERS, TIERS, ETC section of the SAD.

Use cases are transformed into use case realizations through two cases; critical and non-critical cases. For non-critical cases the use case simply contributes to the Domain Model, which can be found in the accompanying SRS document. As for the SAD, the use case information is taken and used to help construct an interaction diagram such as a communication diagram which is then used to help build the final class diagram. This is because the architectural information that the use case is describing is considered non-critical to the overall system and can be easily and safely implemented without any more design.

The same is not true for critical use cases. For these the use case is used to build first a state diagram, then a system sequence diagram (SSD) which then lends itself into building a systems operation and then an operations contract. This is in addition to using the use case to help build the domain model. All of the above are presented inside the SRS document. As for the SAD document, using the above artifacts, an interaction diagram is built which is once again used to help build the class diagram. The reason there are so many artifacts for a critical use case is because the use case represents a vital part of the overall system and its architecture that cannot afford to be compromised. Therefore the multitude of artifacts are created to ensure that the final implementation of the use case is sound and has little to no known flaws.

LAYERS, TIERS ETC.

The system follows a layered architecture to promote low coupling, reuse of core classes and a separation of concerns. This is accomplished through the use of TDG, UoW and mappers which help the domain objects communicate with the databases and ensure that there is no corruption of information happening to information inside the database when two or more domain objects access the same information from the database.

The overall system is divided into many different layers which can be viewed below. First is the Presentation layer which contains the GUI folder which handles all of the front end implementation of the system. For example, the Presentation layer handles the design and visual aspects of the login page, the reservation page etc. Next comes the Domain/Application layer of the overall system which contains the Core folder. This folder contains all the necessary classes needed to handle the internal logic of the system and contains the mappers and UoW necessary for the above internal logic to interact appropriately with the database to retrieve and modify the necessary information. Lastly comes the Data Source/Technical Services layer which contains the database and all classes that handle communication with the database which are the TDG classes that correspond with their respective classes inside the Domain/Application layer.

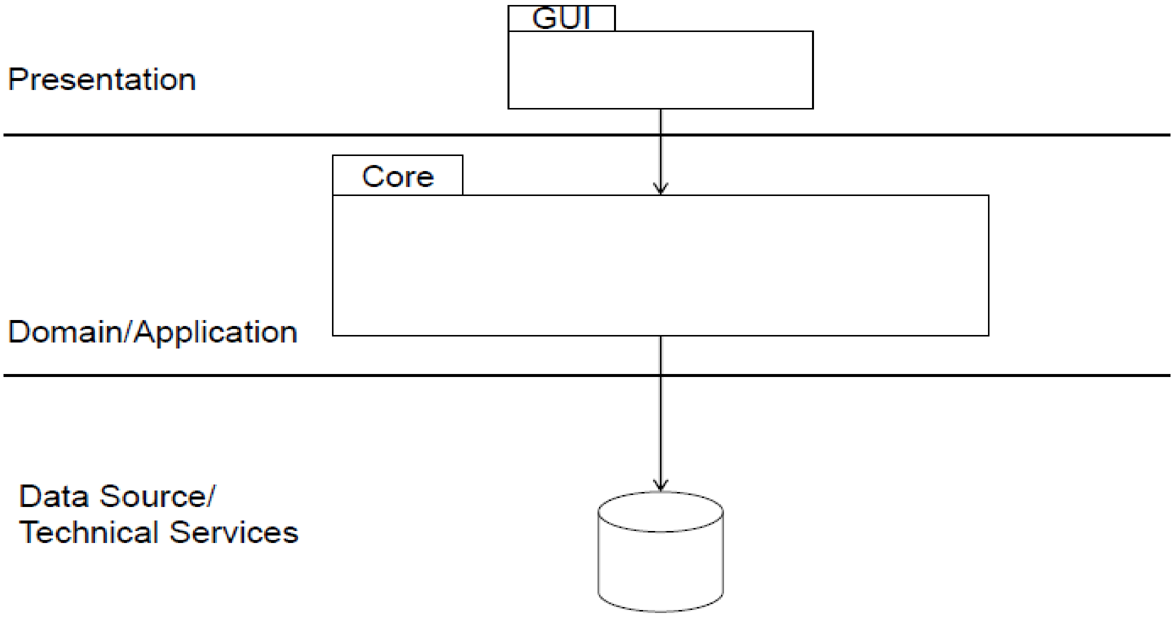


Figure 1: Simplified version of layers and folders that compose final system.

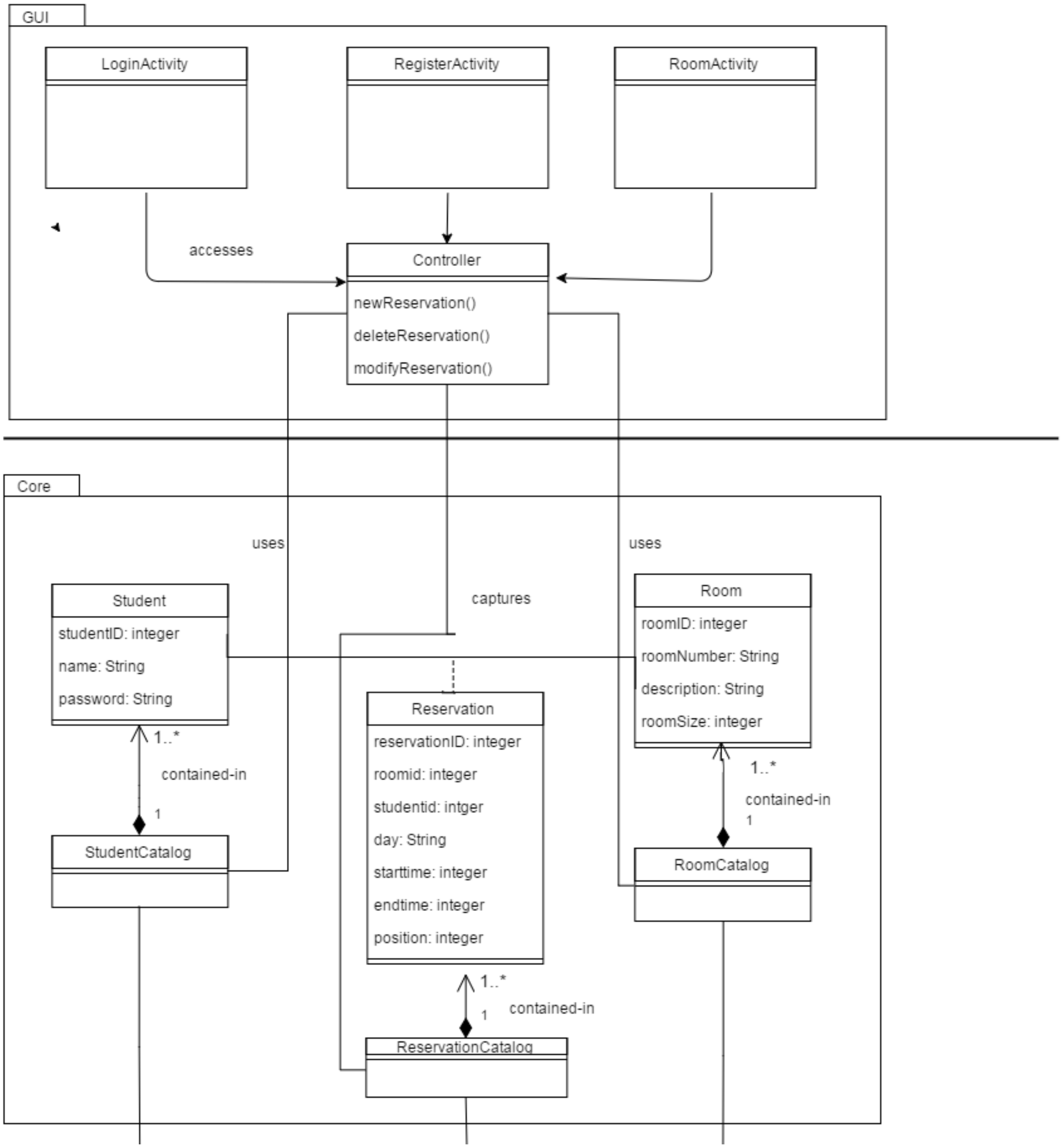

GUI
LoginActivity
RegisterActivity
RoomActivity
accesses
Controller
newReservation()
deleteReservation()
modifyReservation()
Core
uses
uses
captures
Student
studentID: integer
name: String
password: String
1..*
contained-in
1
StudentCatalog
Room
roomID: integer
roomNumber: String
description: String
roomSize: integer
1..*
contained-in
1
RoomCatalog
Reservation
reservationID: integer
roomid: integer
studentid: intger
day: String
starttime: integer
endtime: integer
position: integer
1..*
1
contained-in
ReservationCatalog

Figure 2: Class diagram of the system representing the core folder and the GUI folder.

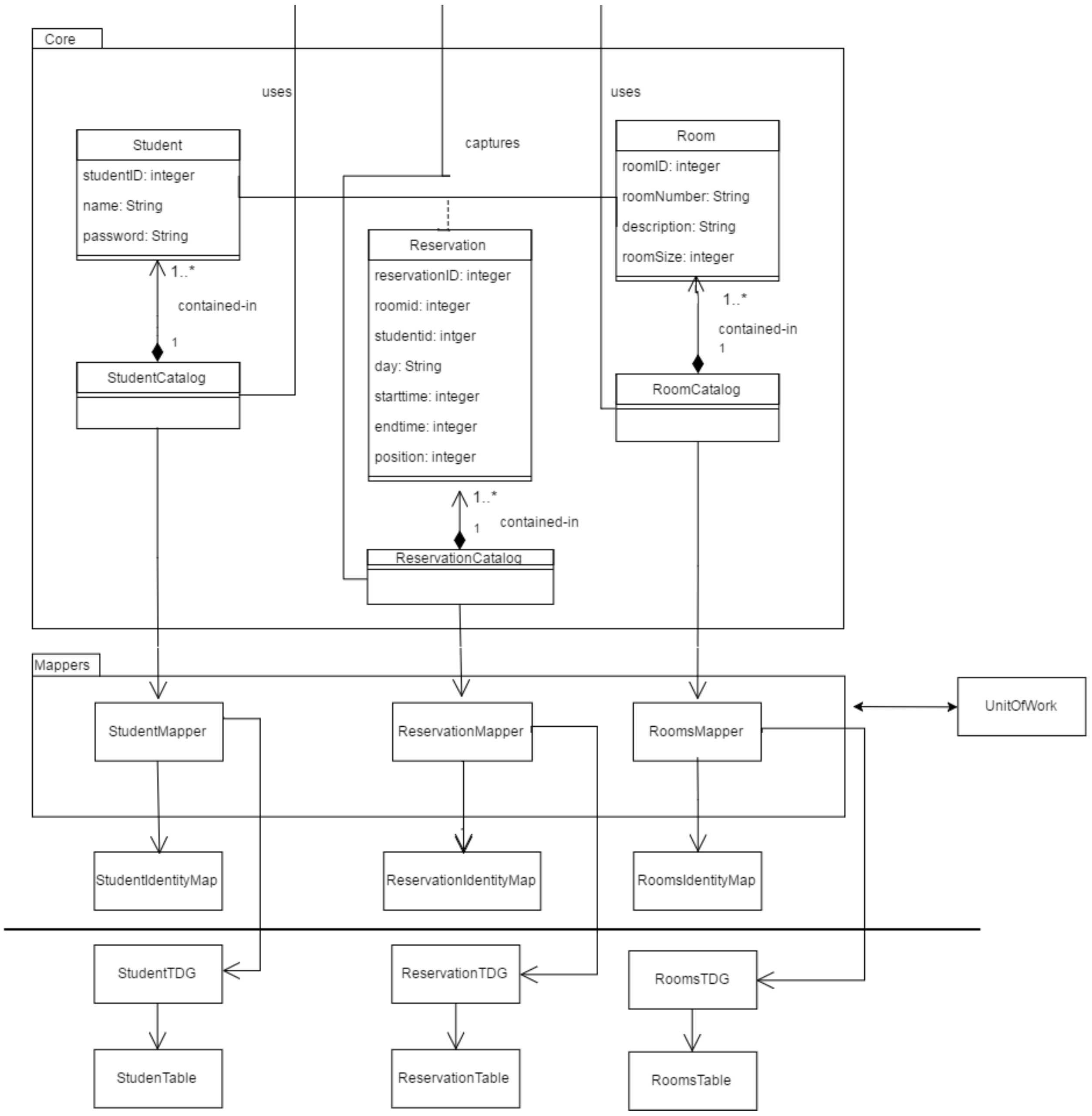

Core
uses
captures
uses
Student
studentID: integer
name: String
password: String
Reservation
reservationID: integer
roomid: integer
studentid: intger
day: String
starttime: integer
endtime: integer
position: integer
Room
roomID: integer
roomNumber: String
description: String
roomSize: integer
1..*
contained-in
1
StudentCatalog
1..*
contained-in
1
RoomCatalog
1..*
1
contained-in
ReservationCatalog
Mappers
StudentMapper
ReservationMapper
RoomsMapper
UnitOfWork
StudentIdentityMap
ReservationIdentityMap
RoomsIdentityMap
StudentTDG
ReservationTDG
RoomsTDG
StudenTable
ReservationTable
RoomsTable

Figure 3: Class diagram of the system representing the core folder and the Mapper folder alongside the connected TDG and UoW classes.

SUBSYSTEMS

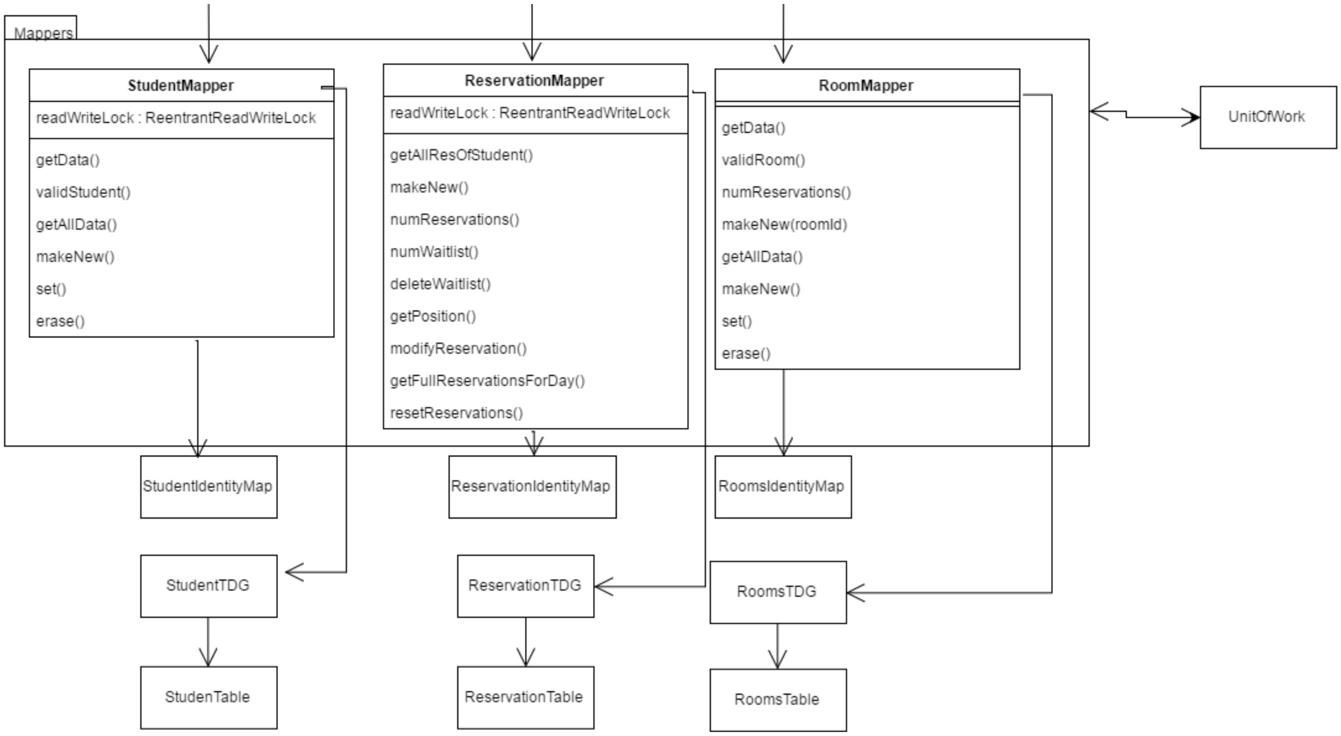


Figure 4: Class diagram of the blown up Mapper folder with the connected TDG classes and UoW.

Architecturally significant design packages

Describe packages of individual subsystems that are architecturally significant. For each package include a subsection with its name, its brief description, and a diagram with all significant classes and packages contained within the package.

USE CASE REALIZATIONS

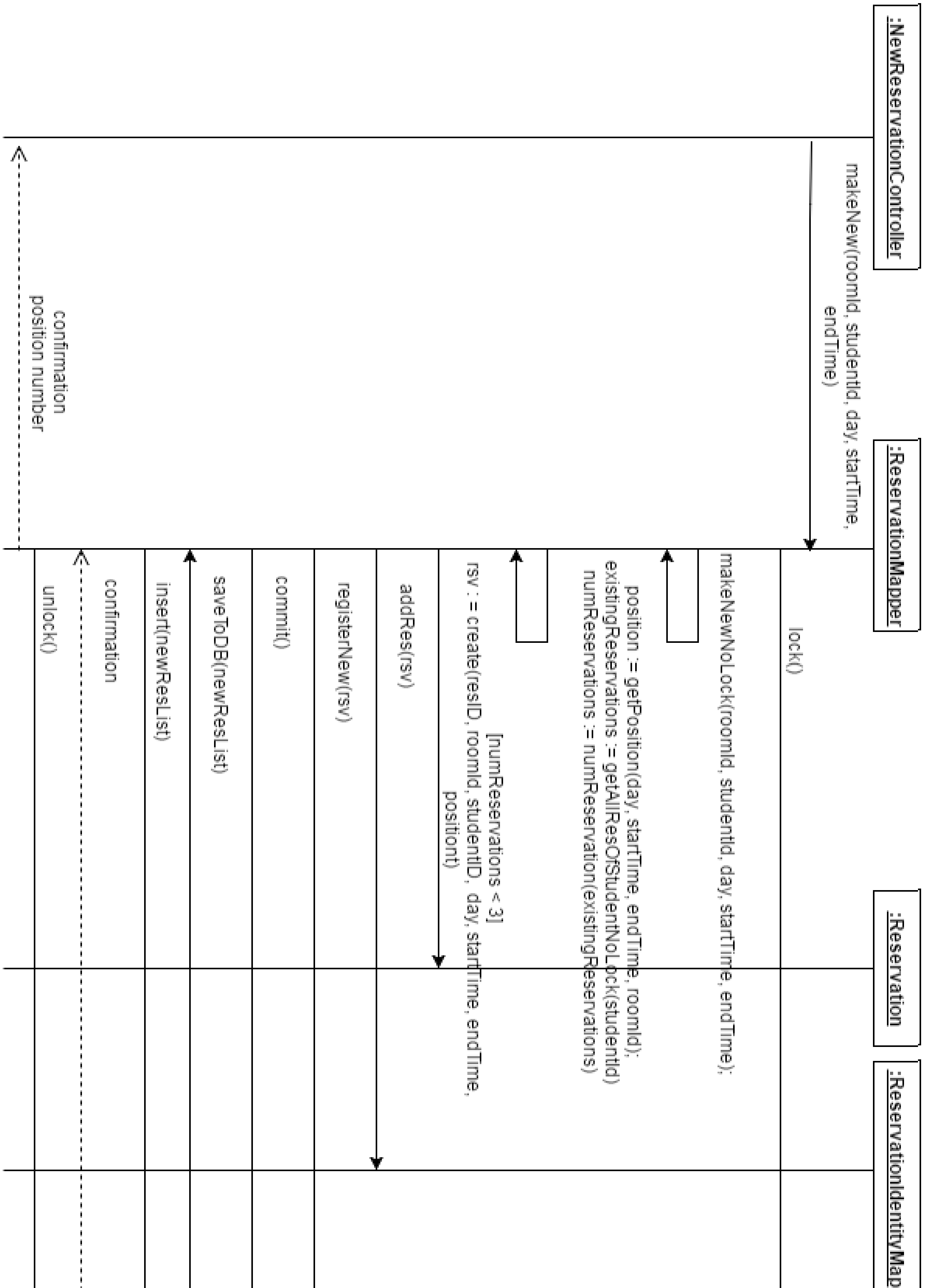
In this section you have to illustrate how use cases are translated into *UML interaction diagrams*. Give examples of the way in which the Use Case Specifications are technically translated into Use Case Realizations, for example, by providing a sequence-diagram. Explain how the tiers communicate and clarify how the components or objects used realize the functionality.

1- Make Reservation/ Add to waitlist:

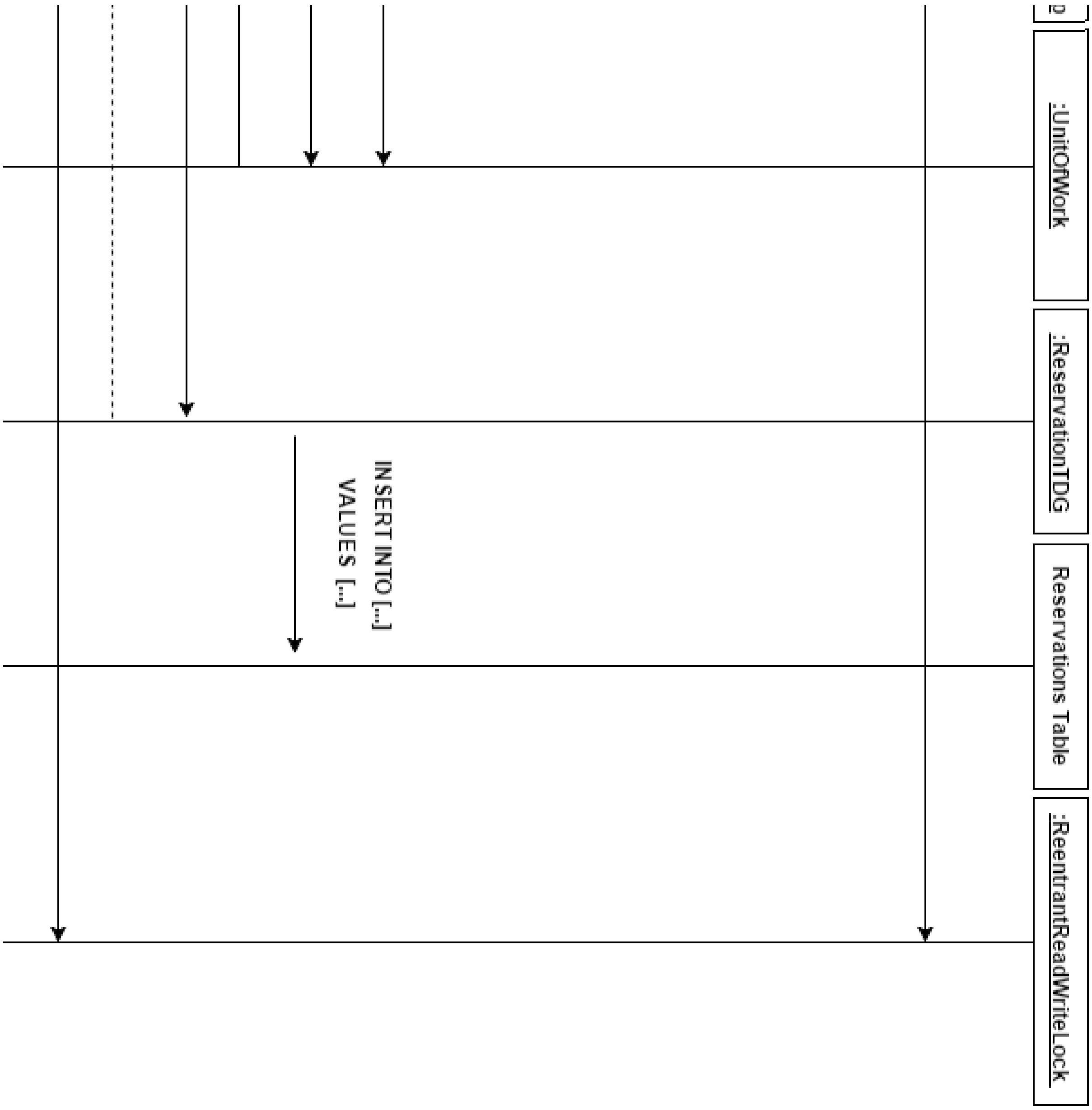
For this use case, a user requests a specific room at a particular time, and then receives a confirmation from the system. A similar flow happens when a user requests to be added to a wait list if a room is not available.

The makeReservation sequence diagram as well as the addToWaitlist sequence diagram are identical in our system because of the use of a position variable . This variable helps us differentiate between a confirmed reservation (the position variable would be 1 for the student associated to that reservation) and a waitlisted reservation. In a waitlisted reservation, the position variable would be greater than one and reflect the position of the student in that waitlist for that particular room at a particular time.

.



:NewReservationController
:ReservationMapper
:Reservation
:ReservationIdentityMap
makeNew(roomId, studentId, day, startTime, endTime)
confirmation
position number
lock()
makeNewNoLock(roomId, studentId, day, startTime, endTime);
position := getPosition(day, startTime, endTime, roomId);
existingReservations := getAllResOfStudentNoLock(studentId)
numReservations := numReservation(existingReservations)
[numReservations < 3]
rsv : = create(resID, roomId, studentID, day, startTime, endTime, position)
addRes(rsv)
registerNew(rsv)
commit()
saveToDB(newResList)
insert(newResList)
confirmation
unlock()



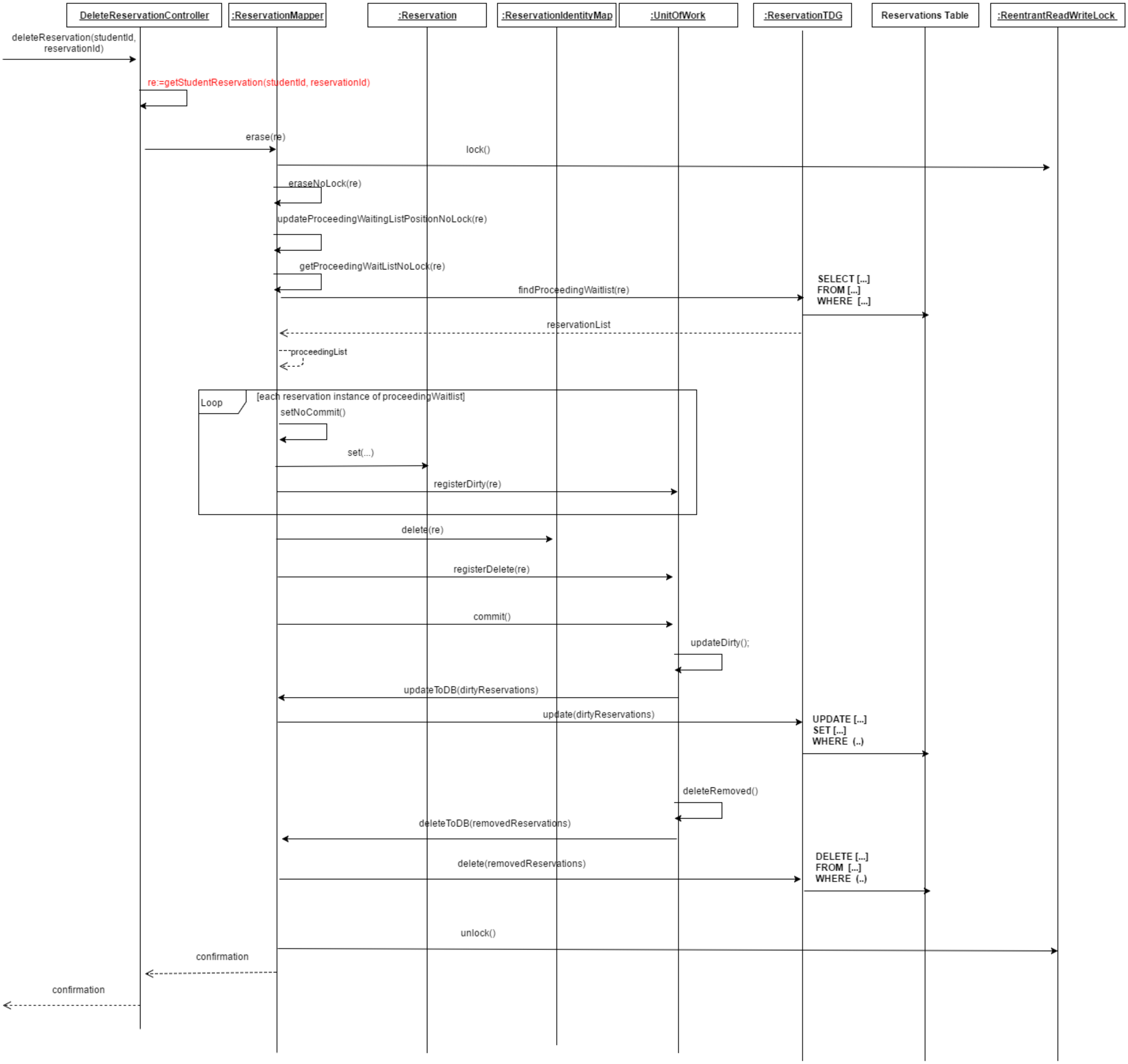
:UnitOfWork
:ReservationTDG
INSERT INTO [...]
VALUES [...]
Reservations Table
:ReentrantReadWriteLock

2- Cancel Reservation/ Remove from waitlist:

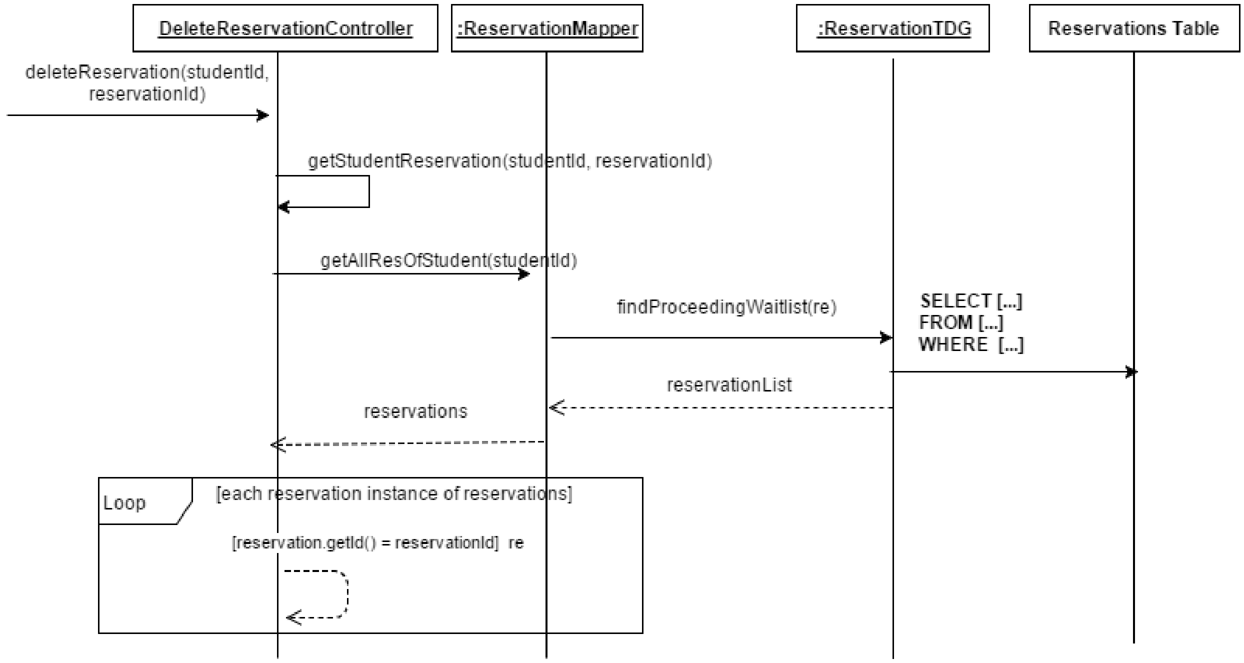
Cancel reservation and Remove from a wait list is the same implementation.

When deleteReservation has been called. The system will retrieve the reservation of the student. When erase() method is being called, it will first go to the Database to retrieve all reservations that are lower than the chosen reservation to delete in the waitlist (higher position number). Then for each of these reservations will be updated (in order to change the positions) and they will be set as "dirty". Then the reservation will be deleted in the Reservation Identity Map. At commit, all the dirty reservations will be updated and the reservation will be deleted in the Database.

NOTE: -for detail version of method getStudentReservation(), go to the diagram after the diagram below


DeleteReservationController
:ReservationMapper
:Reservation
:ReservationIdentityMap
:UnitOfWork
:ReservationTDG
Reservations Table
:ReentrantReadWriteLock
deleteReservation(studentId, reservationId)
re:=getStudentReservation(studentId, reservationId)
erase(re)
lock()
eraseNoLock(re)
updateProceedingWaitingListPositionNoLock(re)
getProceedingWaitListNoLock(re)
findProceedingWaitlist(re)
SELECT [...]
FROM [...]
WHERE [...]
reservationList
proceedingList
Loop
[each reservation instance of proceedingWaitlist]
setNoCommit()
set(...)
registerDirty(re)
delete(re)
registerDelete(re)
commit()
updateDirty();
updateToDB(dirtyReservations)
update(dirtyReservations)
UPDATE [...]
SET [...]
WHERE (..)
deleteRemoved()
deleteToDB(removedReservations)
delete(removedReservations)
DELETE [...]
FROM [...]
WHERE (..)
unlock()
confirmation
confirmation

Detailed version for getStudentReservation() method:



DeleteReservationController
:ReservationMapper
:ReservationTDG
Reservations Table
deleteReservation(studentId, reservationId)
getStudentReservation(studentId, reservationId)
getAllResOfStudent(studentId)
findProceedingWaitlist(re)
SELECT [...]
FROM [...]
WHERE [...]
reservationList
reservations
Loop
[each reservation instance of reservations]
[reservation.getId() = reservationId] re

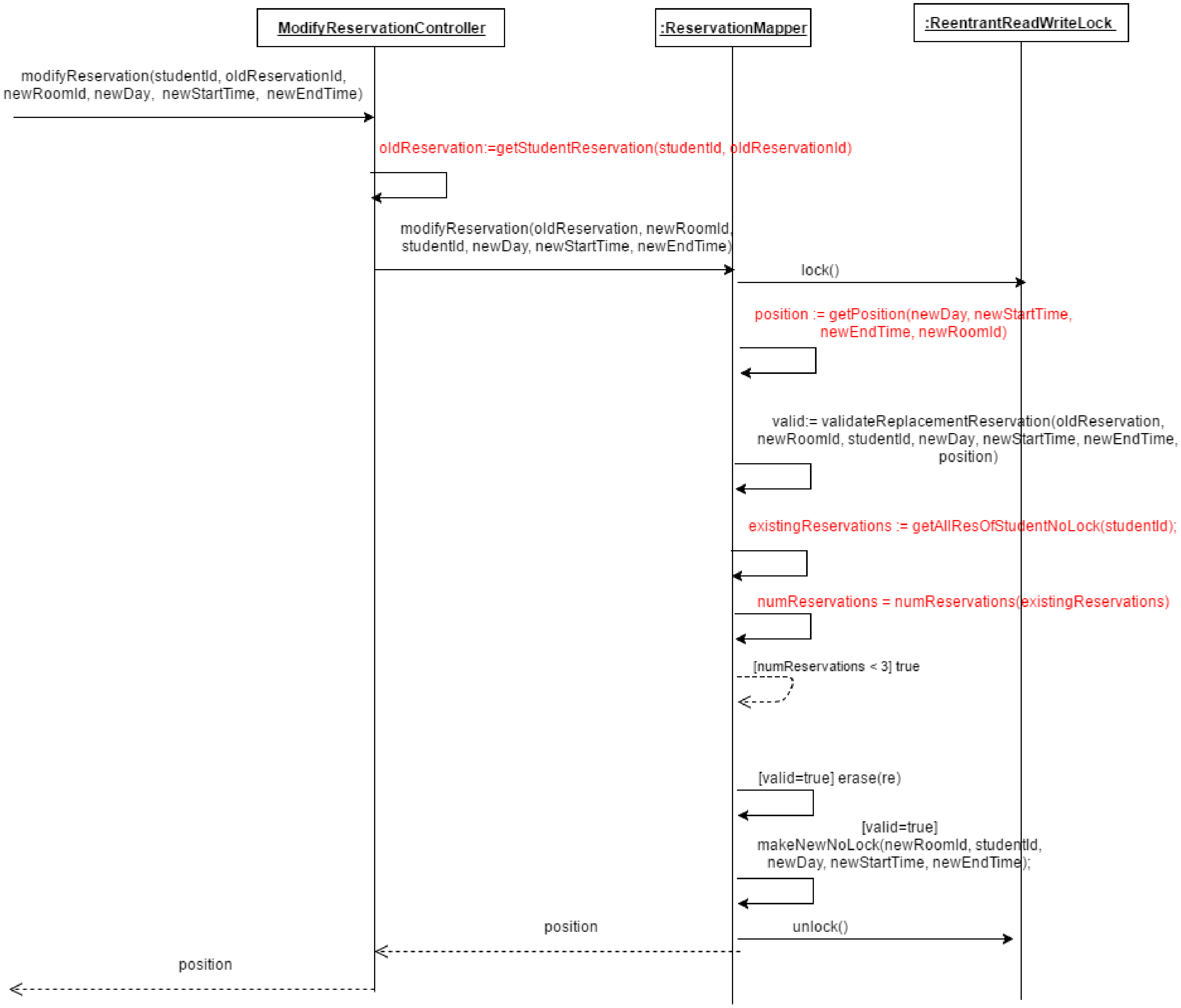
3- Modify Reservation:

When modifyReservation method has been called, the system will retrieve the old reservation of the student. Then the second modifyReservation method will be called. The system will first get the position, either first place or in the waitlist. The System will then validate the changes that need to be done. When it is validated that the number of reservation of the student did not exceed its maximum (3 max), it will return true. Then the method erase() and makeNew() method will be call.

NOTE: -please see diagram for method getStudentReservation() in Cancel reservation section

- see previous diagrams above for method erase() and makeNew()

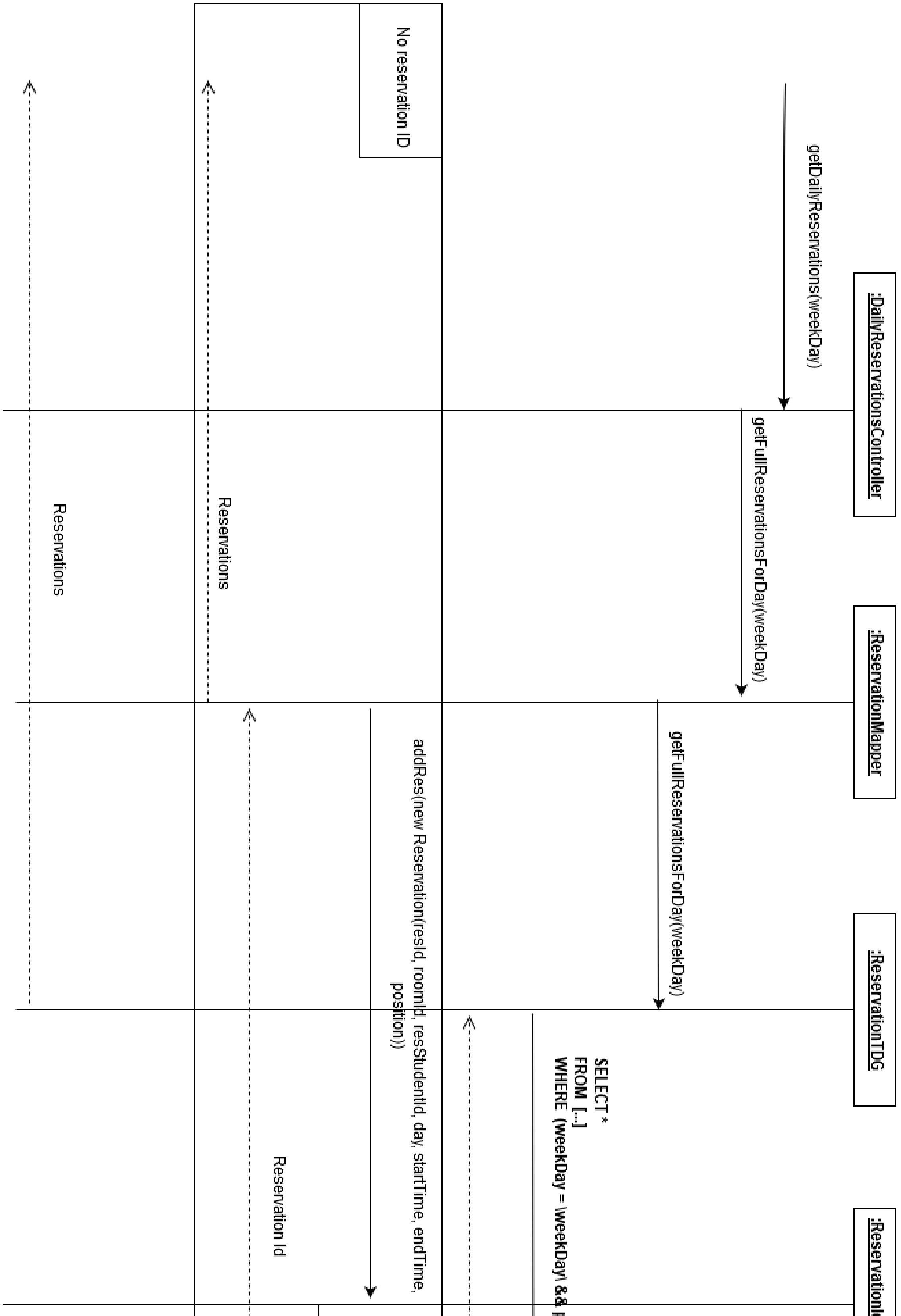
-methods in red are simplified (no detail)

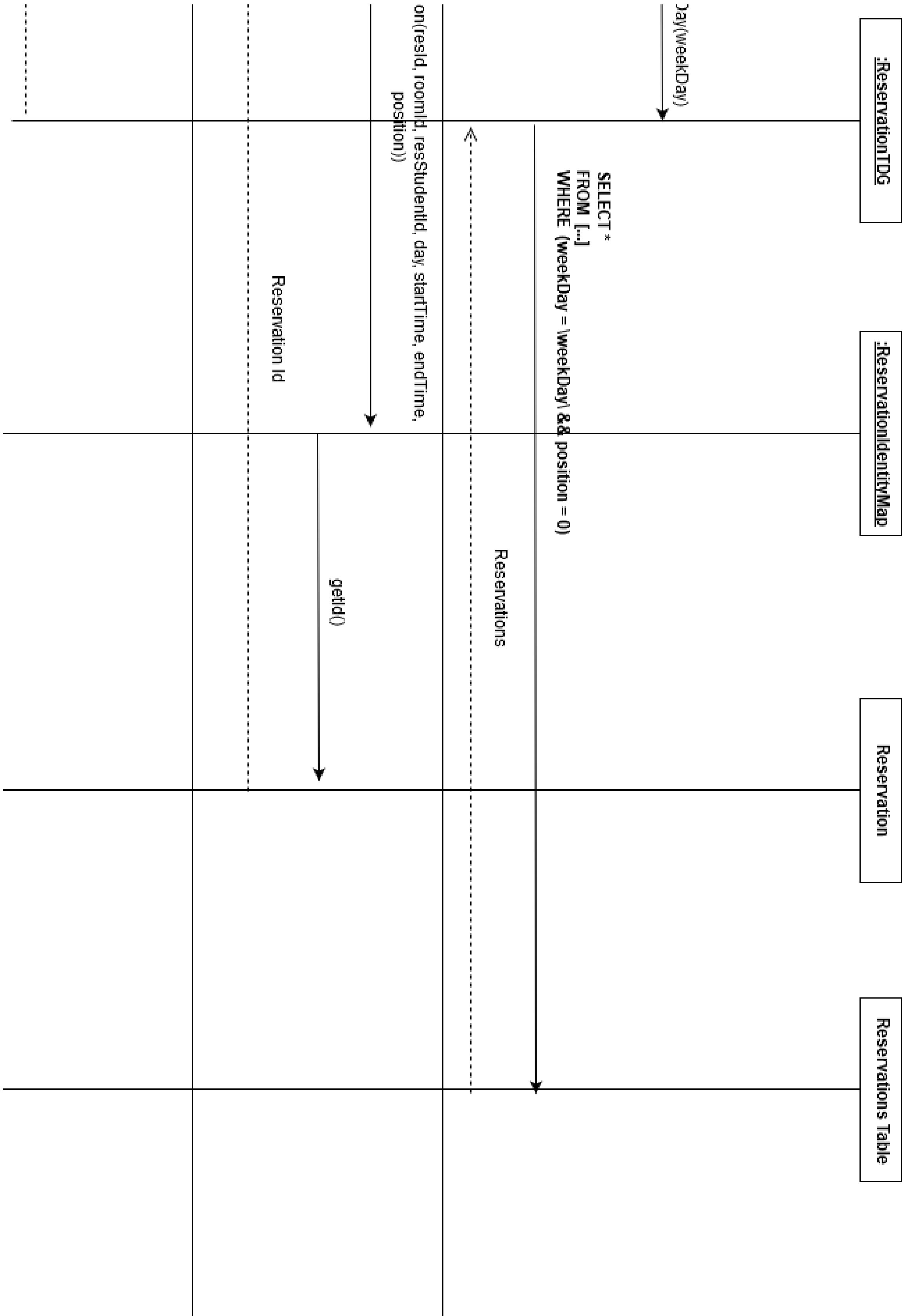


ModifyReservationController
:ReservationMapper
:ReentrantReadWriteLock
modifyReservation(studentId, oldReservationId, newRoomId, newDay, newStartTime, newEndTime)
oldReservation:=getStudentReservation(studentId, oldReservationId)
modifyReservation(oldReservation, newRoomId, studentId, newDay, newStartTime, newEndTime)
lock()
position := getPosition(newDay, newStartTime, newEndTime, newRoomId)
valid:= validateReplacementReservation(oldReservation, newRoomId, studentId, newDay, newStartTime, newEndTime, position)
existingReservations := getAllResOfStudentNoLock(studentId);
numReservations = numReservations(existingReservations)
[numReservations < 3] true
[valid=true] erase(re)
[valid=true] makeNewNoLock(newRoomId, studentId, newDay, newStartTime, newEndTime);
position
unlock()
position

4-View Reservation:

When getDailyReservationsWeekDay method has been called, the system will retrieve all the reservations made for a particular day of the week from the database and return them. If nothing can be found then the reservation ID is retrieved and then inputted into the database to retrieve the proper reservations for a particular day.

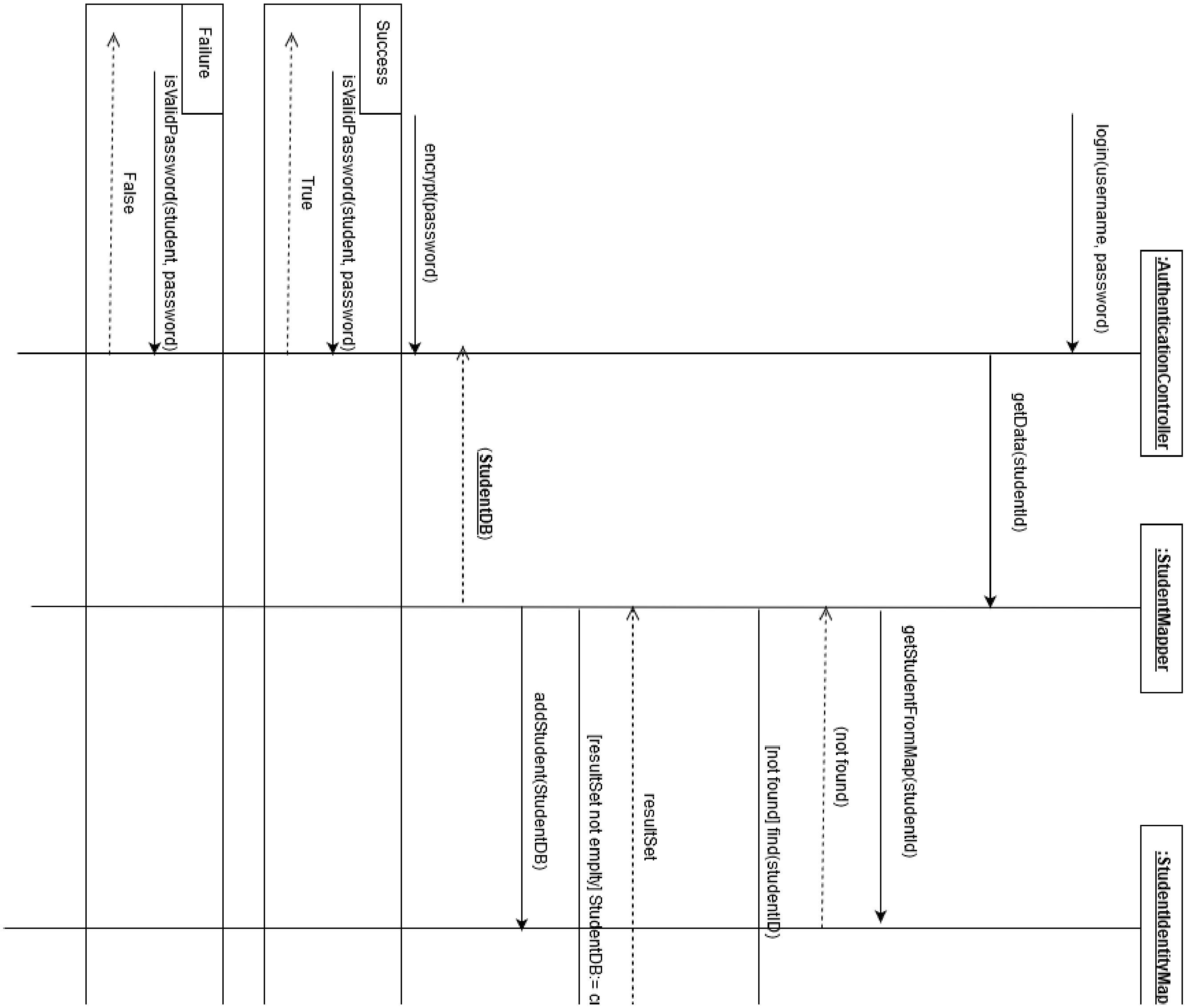

:DailyReservationsController
:ReservationMapper
:ReservationTDG
:Reservation
getDailyReservations(weekDay)
getFullReservationsForDay(weekDay)
getFullReservationsForDay(weekDay)
SELECT *
FROM [...]
WHERE (weekDay = \weekDay\ &&
addRes(new Reservation(resId, roomId, resStudentId, day, startTime, endTime, position))
Reservation Id
No reservation ID
Reservations
Reservations

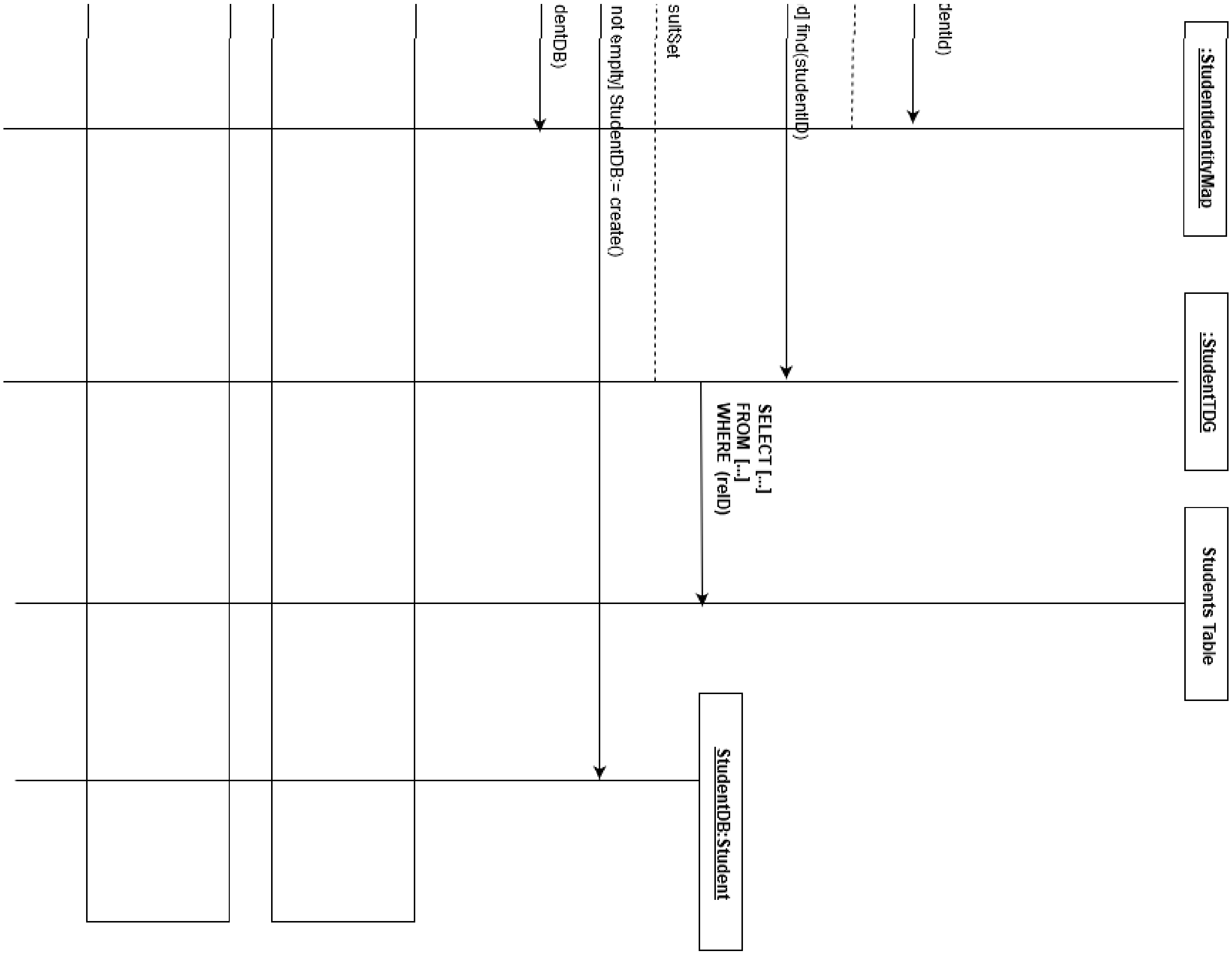


:ReservationTDG
:ReservationIdentityMap
Reservation
Reservations Table
Day(weekDay)
SELECT *
FROM [...]
WHERE (weekDay = \weekDay\ && position = 0)
Reservations
on(resId, roomId, resStudentId, day, startTime, endTime, position))
getId()
Reservation Id

<u>4-Authentication:</u>

When the login method is called it retrieves the student information from the database and compares it that was inputed by the user. If nothing is found the the student information is added into the database. If the inputted information was incorrect then the user is given a failed message. If the inputted information is correct then a confirmation message is given.


:AuthenticationController
login(username, password)
getData(studentId)
:StudentMapper
getStudentFromMap(studentId)
(not found)
[not found] find(studentID)
resultSet
[resultSet not empty] StudentDB:= c
addStudent(StudentDB)
(StudentDB)
encrypt(password)
:StudentIdentityMap
Success
isValidPassword(student, password)
True
Failure
isValidPassword(student, password)
False



:StudentIdentityMap
:StudentTDG
Students Table
StudentDB:Student
dentId)
d] find(studentID)
sultSet
not empty] StudentDB:= create()
dentDB)
SELECT [...]
FROM [...]
WHERE (relD)