Concordia University
Department of Computer Science
and Software Engineering

Software Architecture and Design II
SOEN 344 S --- 2017

Chronos
Software Architecture Documentation

Team members	
Name	**Student ID**
An Ran Chen	27277385
Christiano Bianchet	27039573
Philippe Kuret	27392680
Alexander Rosser	27543069
Saif Mahabub	27392974
Amr Mourad	26795331
Youness Tawil	27013353
Adriel Fabella	27466005

Table of Contents

List of Figures

List of Tables

1. Introduction

In software development, an ordered process is required in engineering a software system. The main stages of this process include requirements engineering, design, implementation, testing, maintenance and version control. The Software Architecture Document (SAD) is dedicated to the second stage of the software development process: design.

In this SAD document, we will provide the various UML diagrams needed to structure the implementation of the software project.

1.1 Purpose

The SAD is a documentation artifact that describes the high-level architecture and low-level design of a software system. It is used to illustrate the system with UML diagrams providing information on its architecture and specific scenarios. In this SAD, it contains architectural views, use cases, class diagram and entity relation diagram.

This document is intended to be used by the team responsible for implementing the system. This development team will produce the code repository respecting the architecture provided in the SAD.

1.2 Scope

The SAD for Chronos, an online university conference room reservation system, is applied to the implementation of its code. It is affected by the Software Requirements Specification (SRS) document made at the previous stage of development: requirements engineering. The architecture for Chronos' code is constructed following the requirements stated in its SRS.

1.3 Glossary of Terms

Table 1: Glossary of Terms

Term	Definition
Active Reservation	The earliest reservation stored for a room at a specific time slot. This reservation currently holds the room and is not part of the waiting list
Available	Room that can be reserved by a user
Reservation	Time slot which has been booked by a user
Room	The entity the user wants to reserve
SAD	Document containing artifacts related to the architecture of the software
Time Slot	One-hour long time interval starting on the hour
User	Student or staff member who is registered with the engineering faculty
Waiting list	A series of reservations that represents the order in which users will take over a canceled reservation for a specific room and time
Equipment	Computers (3) and Projectors (3) that are made available to a user

2. Logical View

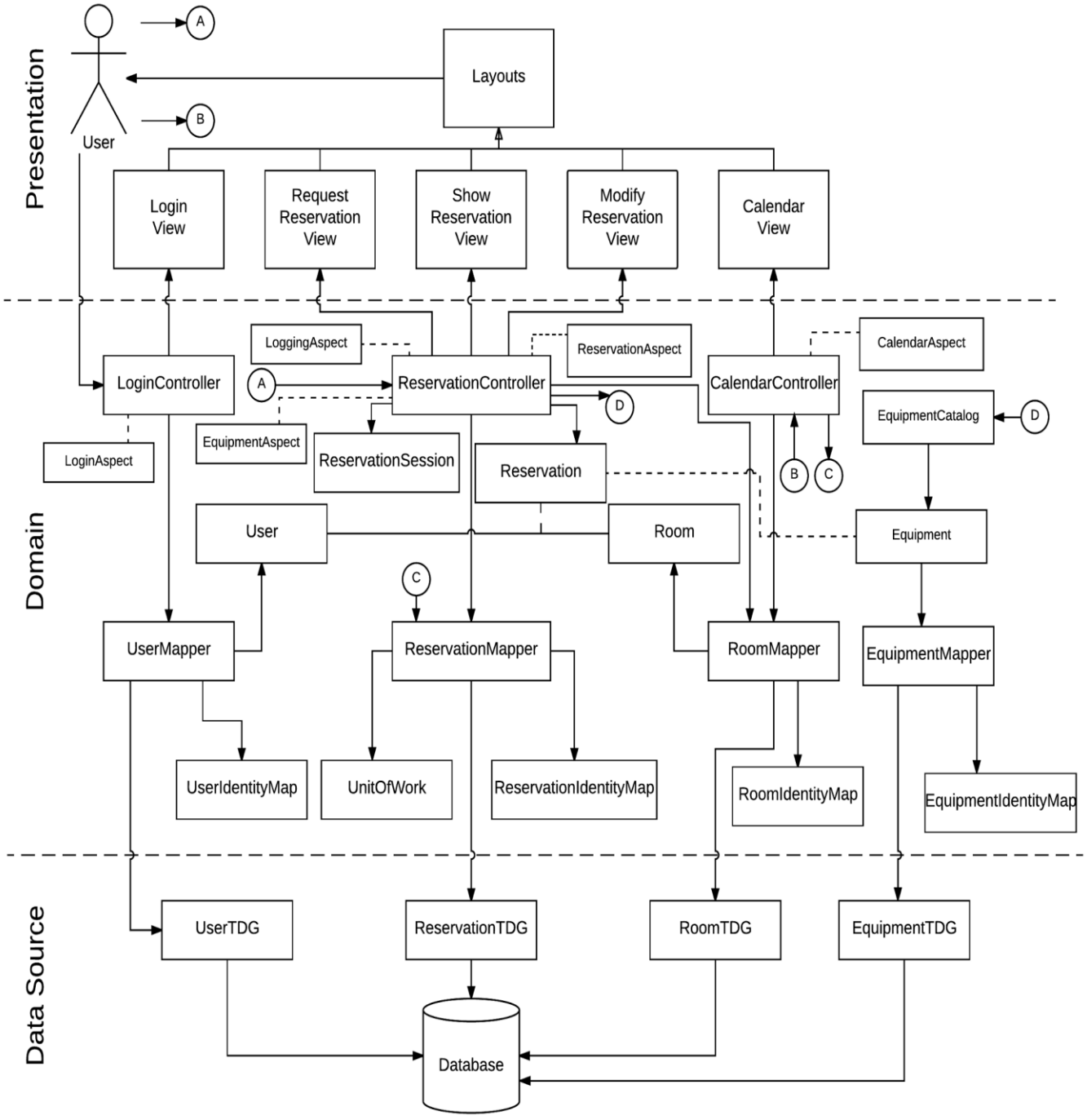


Figure 1: Layered Architecture Diagram

2.1 Layers

2.1.1 Overview

As shown in *Figure 1*, the architecture of the system is best described as following a layered architectural style consisting of three layers: Data, Domain and Presentation. This follows the standard three-layer style of a system to best separate concerns and modularize these components of the system. Also, as seen in the same figure, layers can only communicate with adjacent layers which makes this a closed (opaque) design. The system may also be considered a Client-Server architecture wherein each layer acts as both a client and server to adjacent layers. The Presentation layer is the topmost layer and therefore acts solely as a client to the Domain layer. The latter acts as a server to respond to the Presentation's requests and as a client which makes its own requests to the Data layer. This layer, in turn, acts solely as a server to serve those requests. Lastly, the system can be described as 2-tier. The first-tier is the browser which the client uses to connect to the website. The second tier is the machine which hosts the website and contains the database. There does not exist a pure tier separation as each tier holds at least some logic of all three layers (ex. front-end validation logic, back-end presentation creation).

2.1.2 Data Layer

The data layer is the lowest layer of the system; it contains the Table Data Gateways (TDG) that interact with the database to persist objects of their related classes. The data layer can only be accessed by a mapper interacting with its respective TDG. The TDGs of concern are the UserTDG, ReservationTDG, RoomTDG, EquipmentTDG and ReservationSessionTDG .

2.1.3 Domain Layer

The domain layer is the middle layer of the system. It contains the main objects of the system, their controllers, and their respective mappers. The domain layer also contains the Identity Maps used to synchronize the use of objects throughout the system. An example of this is the UserMapper, which maps the functions in the UserTDG that manipulate columns in the database with their respective attributes in the User class. The UserIdentityMap then tracks when changes are made to the User object ensuring that those changes are limited to only a few database calls instead of one for each change. The process is similar for both the ReservationMapper , EquipmentMapper and RoomMapper.

2.1.4 Presentation Layer

The presentation layer is the outermost layer of the system. This outermost layer is the layer the user interacts with. It contains all the views generated by controllers in the domain level and based on the user's input. These views are controlled by three main controllers which exist in the Domain layer: LoginController, ReservationController and CalendarController. They are the link between the two layers and make/display changes to the user.

2.2 Class Diagram

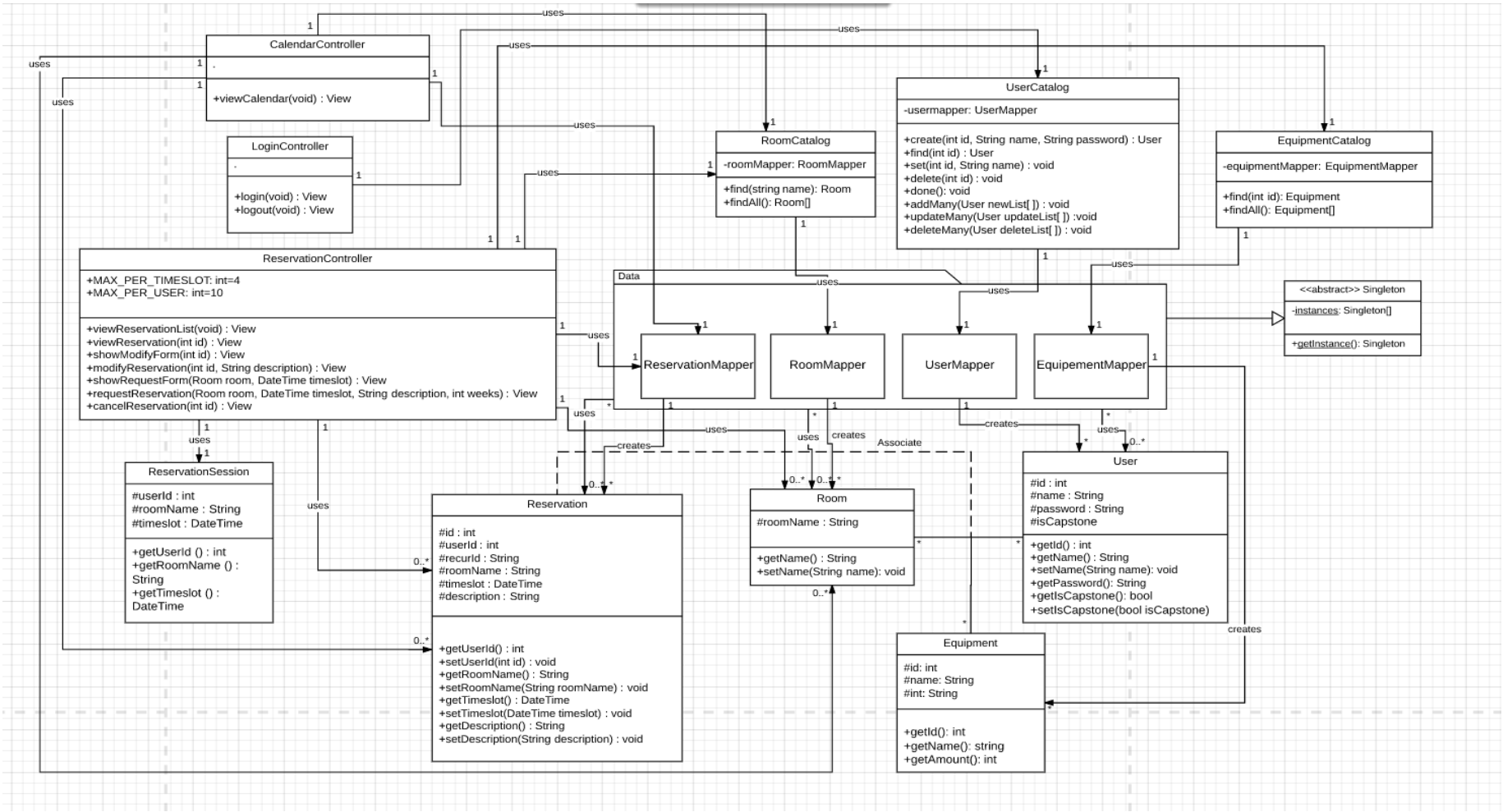


Figure 2: Main Class Diagram

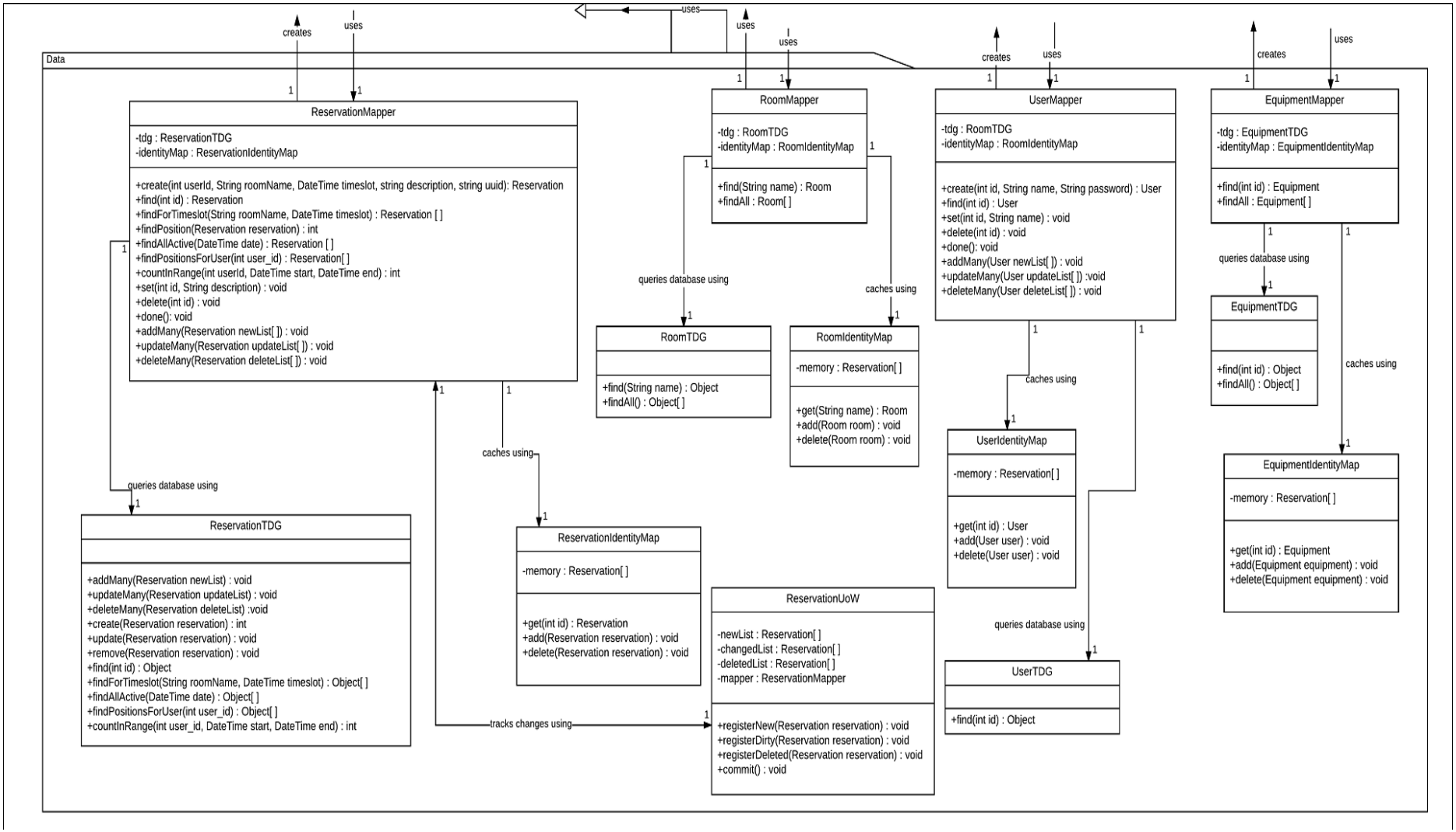


Figure 3: Full Implementation of Data Package

There are two main design patterns on display in this class diagram: Singleton and Controller. Every class within the data package extends the Singleton abstract class. The Singleton design pattern simply ensures that only one instance of that class exists using a private static variable which holds that instance. The only way to acces it is through the public static method getInstance(). This means that every Mapper, IdentityMap, TDG, and UoW class only exists as one instance throughout the entire system.

The Controller design pattern consists of one class having many associations and delegating tasks accordingly. In the case of this system, there are three main controllers: LoginController, ReservationController, and CalendarController. The LoginController handles all tasks regarding retrieving user information, through delegation to the UserMapper, and associating that user with this current session when logging in. The ReservationController handles the creation, through the ReservationMapper, and manipulation of reservations. Lastly, the CalendarController handles both Reservations and Rooms, their scheduling, and how they should be displayed to the user. This gives a general sense of the flow of the system, starting from the LoginController and moving to the creation, scheduling and display of Reservations using the ReservationController and CalendarController.

Further information on interceptor pattern will be discussed in section *5 - AOP Migration of Interceptor Pattern*.

2.3 Interaction Diagrams

section 2.3.7 for all referenced sequence diagrams

2.3.1 View Calendar

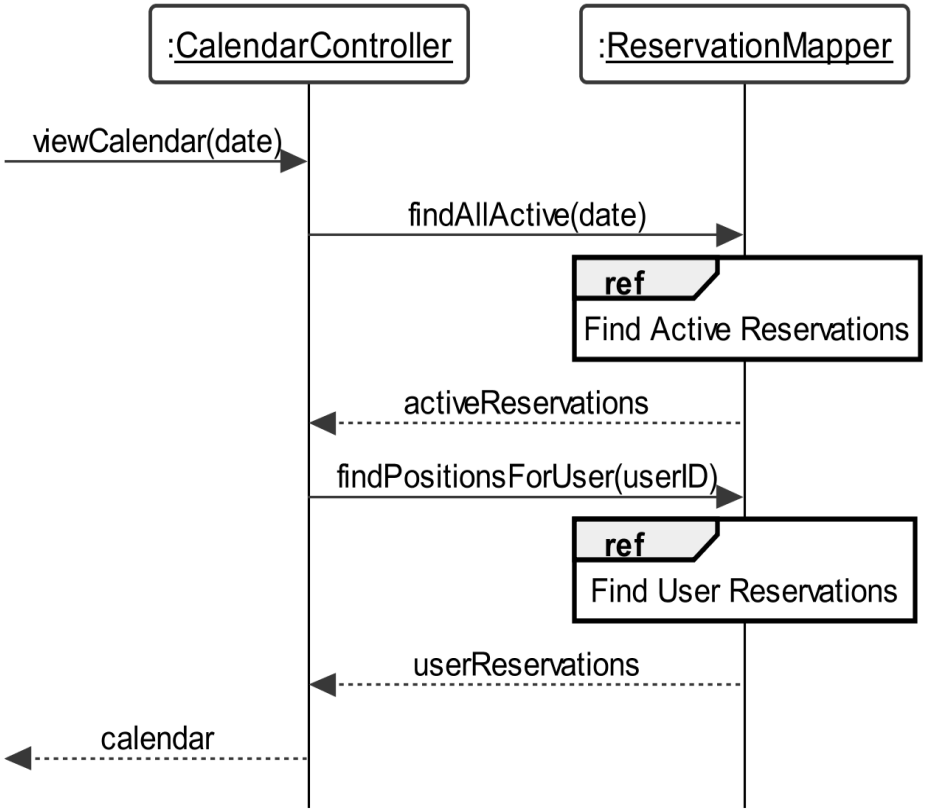
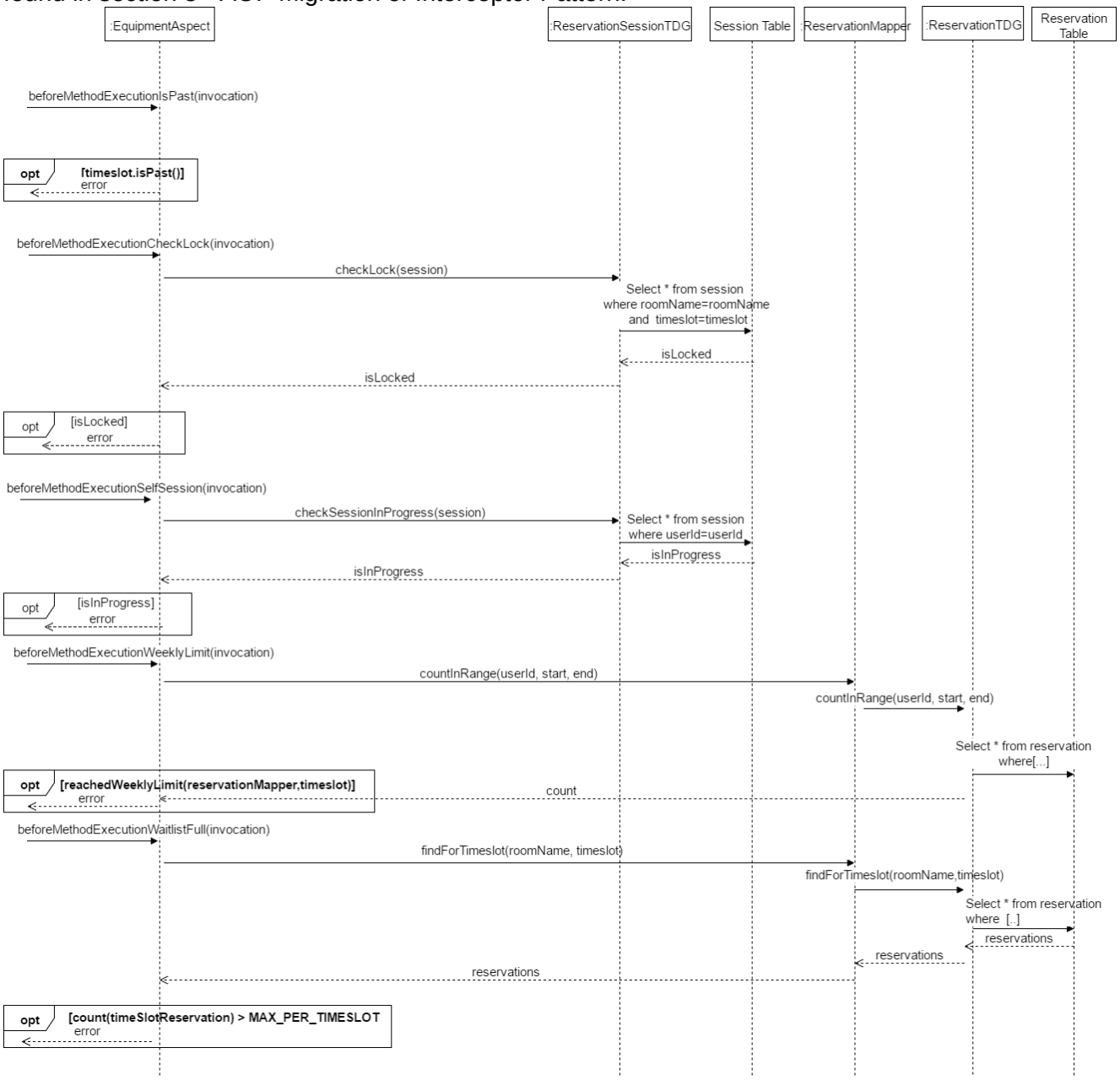


Figure 4: Sequence diagram for View Calendar's viewCalendar() system operation

2.3.2 Request Reservation

2.3.2.1 showRequestForm

Before the method *showRequestForm* is called, a few condition checking is performed in *EquipmentAspect* using the *Interceptor* pattern. Further information on *EquipmentAspect* can be found in section 5 - AOP Migration of Interceptor Pattern.



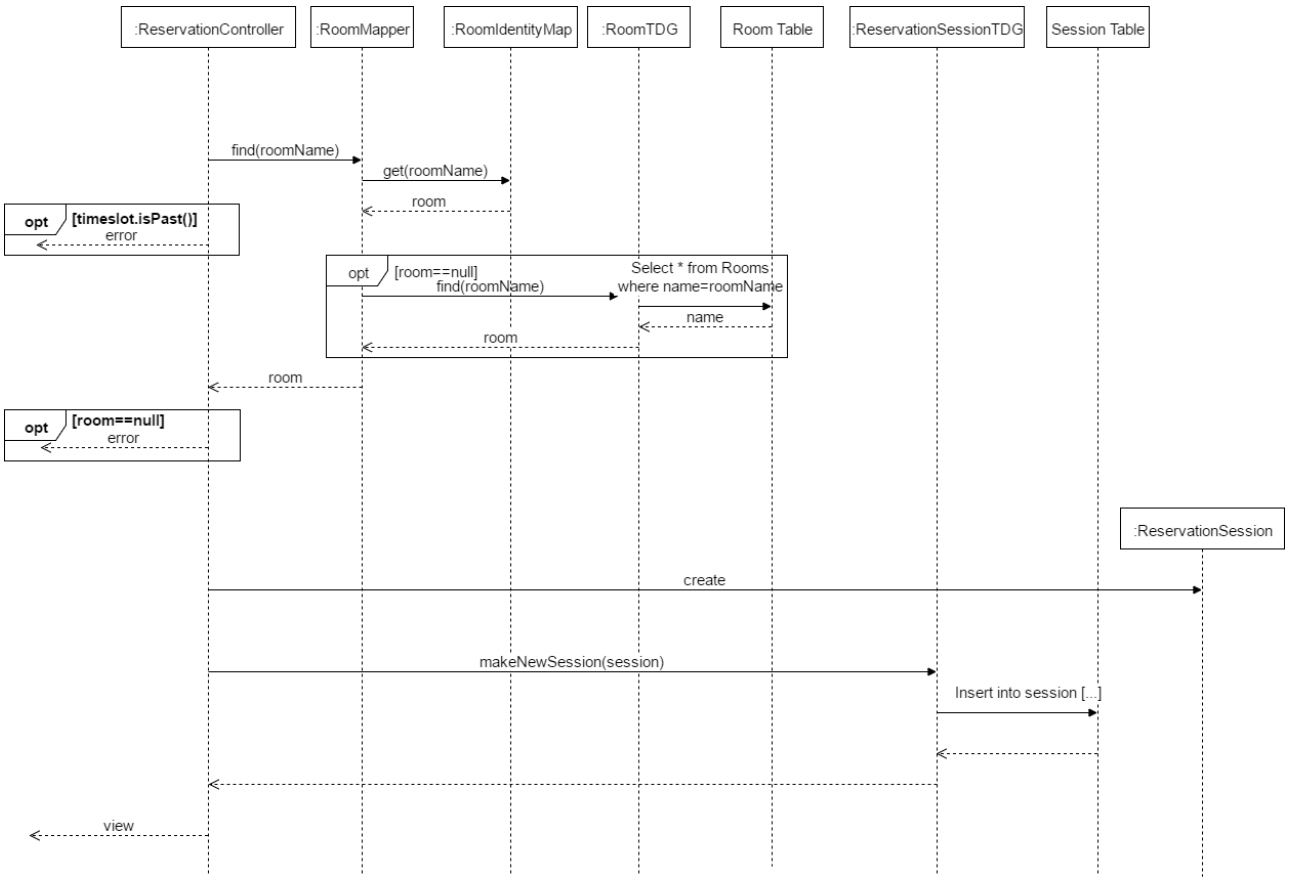


Figure 5: Sequence diagram for showRequestForm() system operation

2.3.2.2 requestReservation

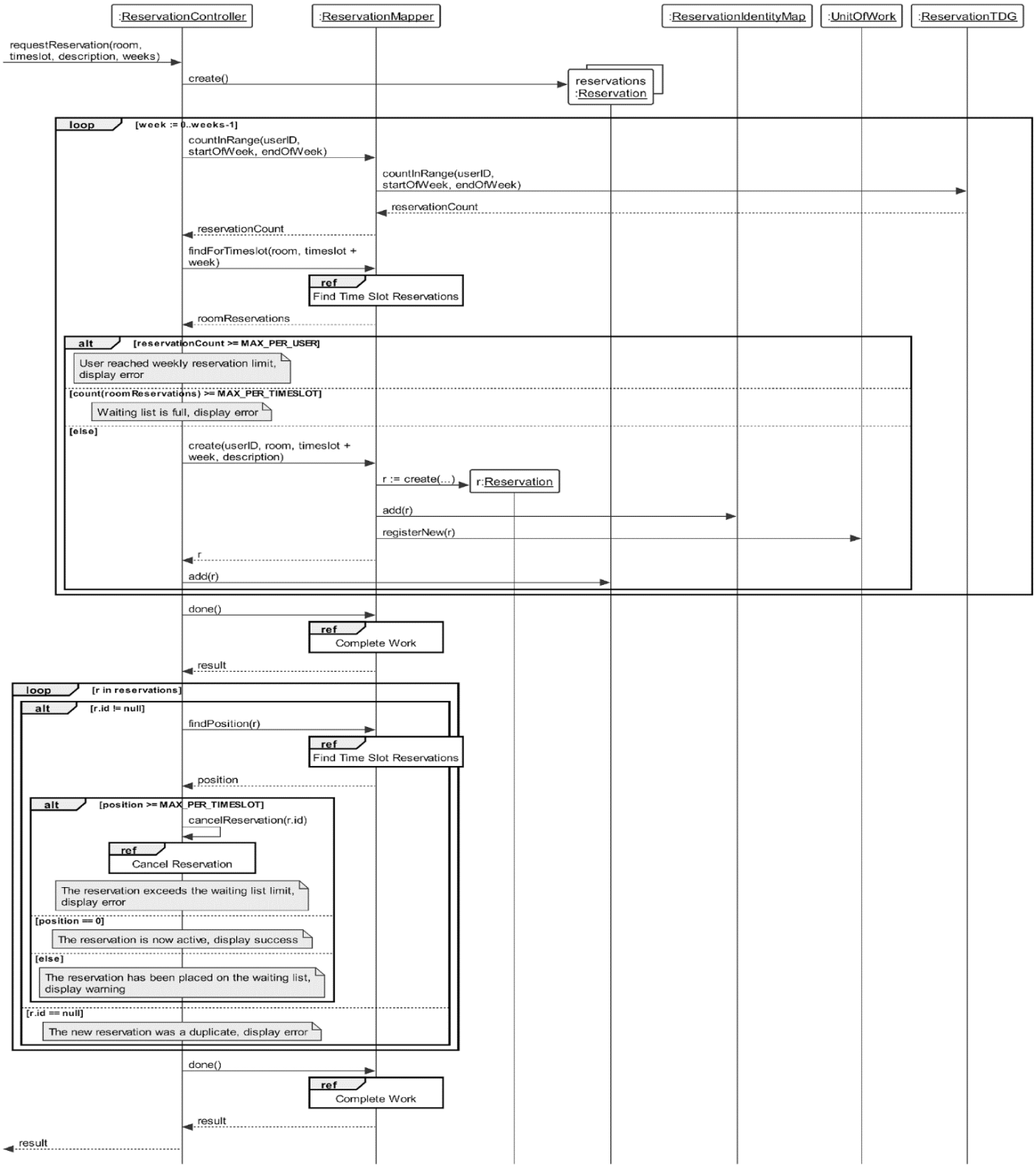


Figure 6: Sequence diagram for requestReservation() system operation

2.3.3 Modify Reservation

2.3.3.1 showModifyForm

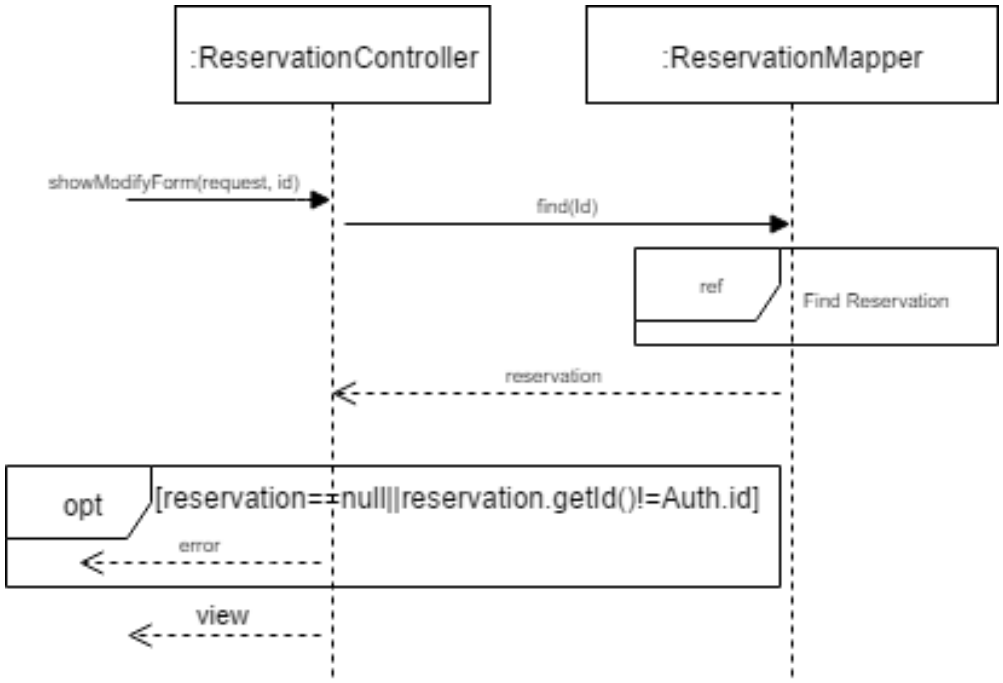


Figure 7: Sequence diagram for getModificationForm() system operation

2.3.3.2 modifyReservation

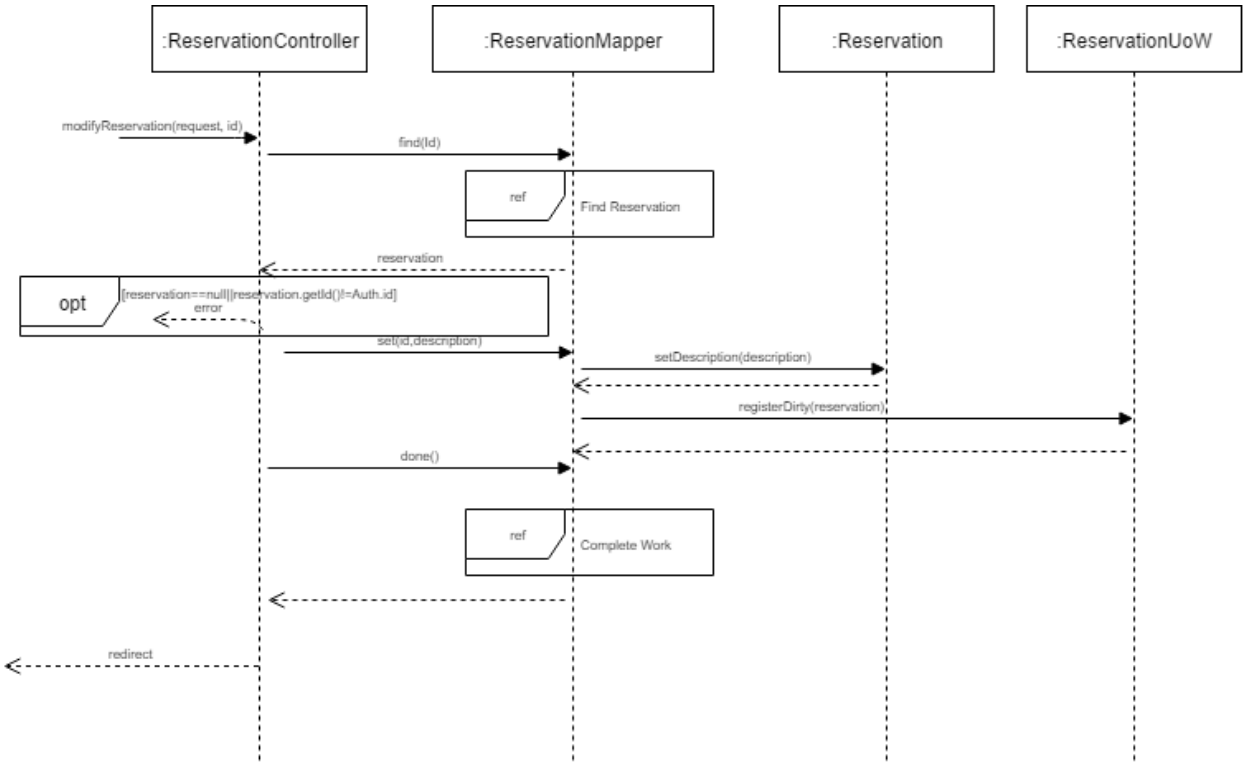


Figure 8: Sequence diagram for modifyReservation() system operation

2.3.4 View Reservation

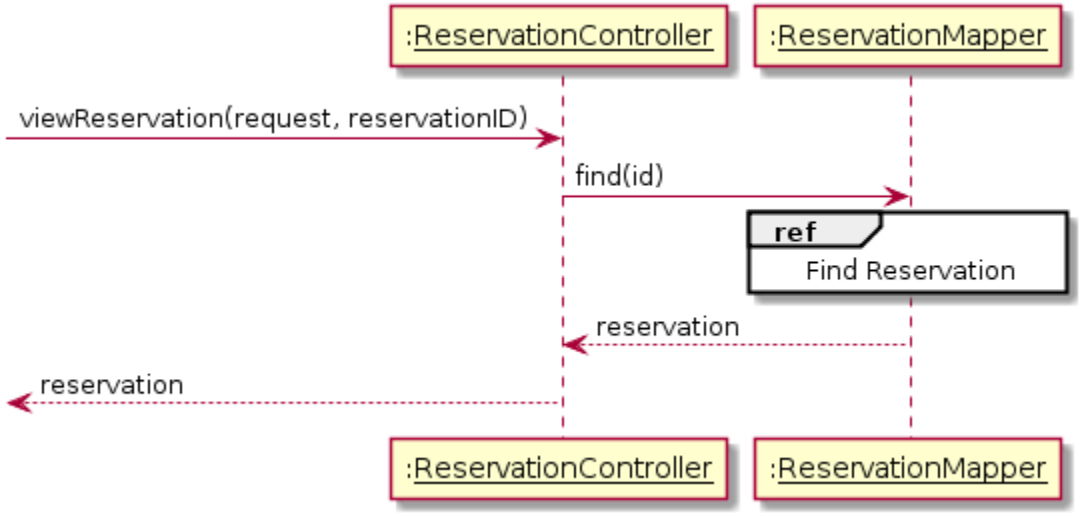


Figure 9: Sequence diagram for View Reservation's viewReservation() system operation

2.3.5 View Reservation List

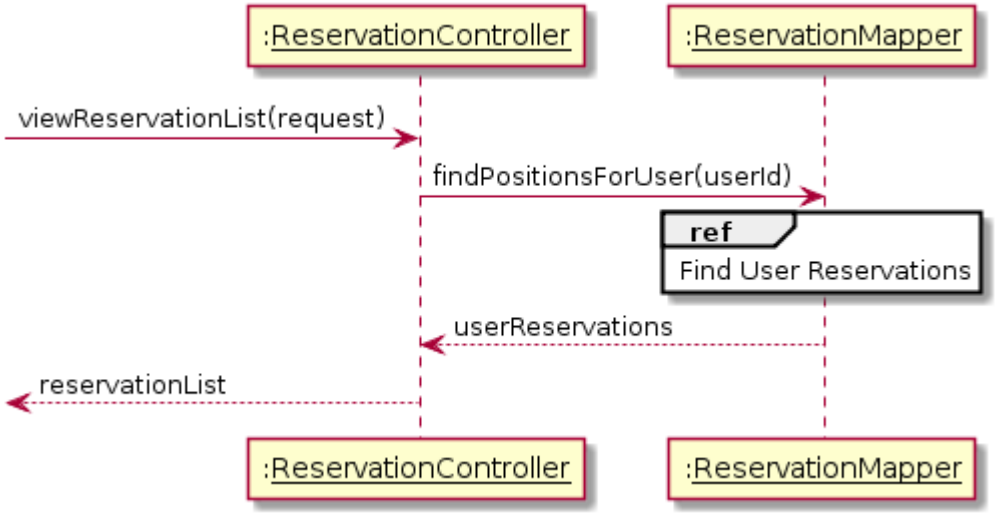


Figure 10: Sequence diagram for viewReservationList() system operation

2.3.6 Cancel Reservation

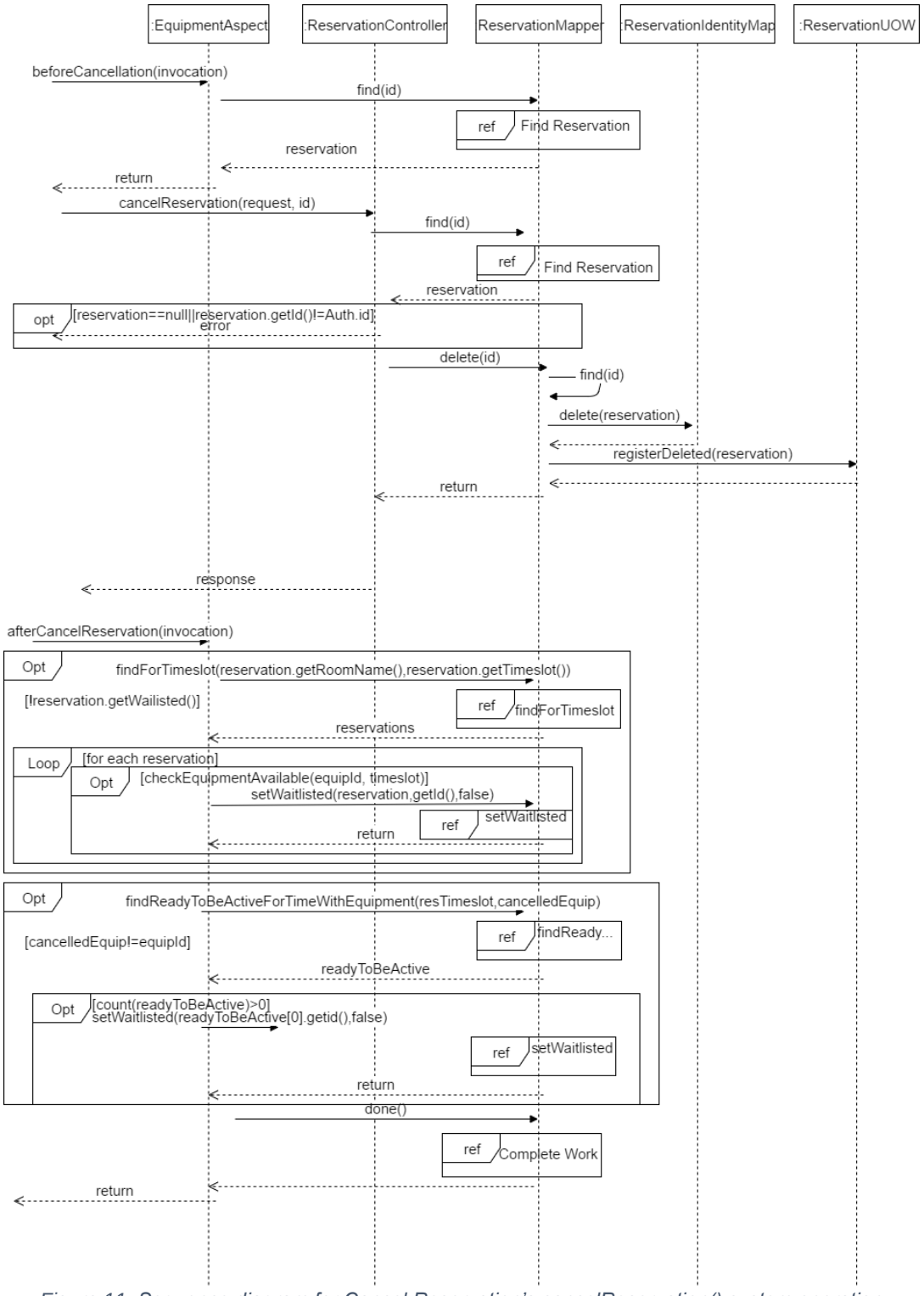


Figure 11: Sequence diagram for Cancel Reservation's cancelReservation() system operation

2.3.7 Referenced Sequence Diagrams

2.3.7.1 Find Active Reservations

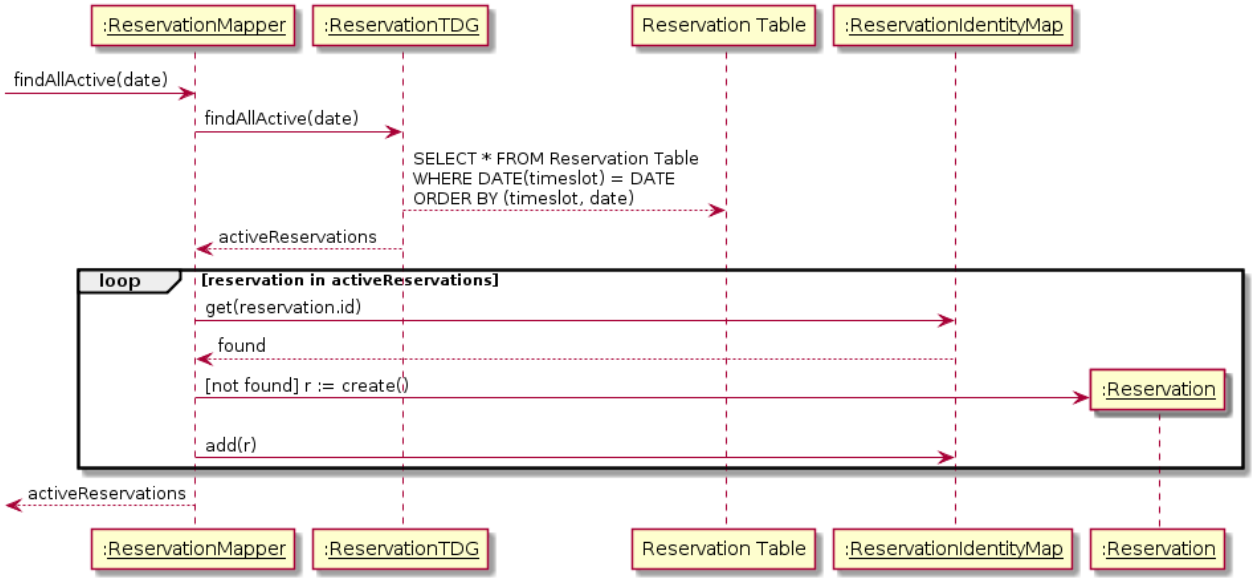


Figure 12: Referenced Sequence Diagram Find Active Reservations

2.3.7.2 Find Reservation

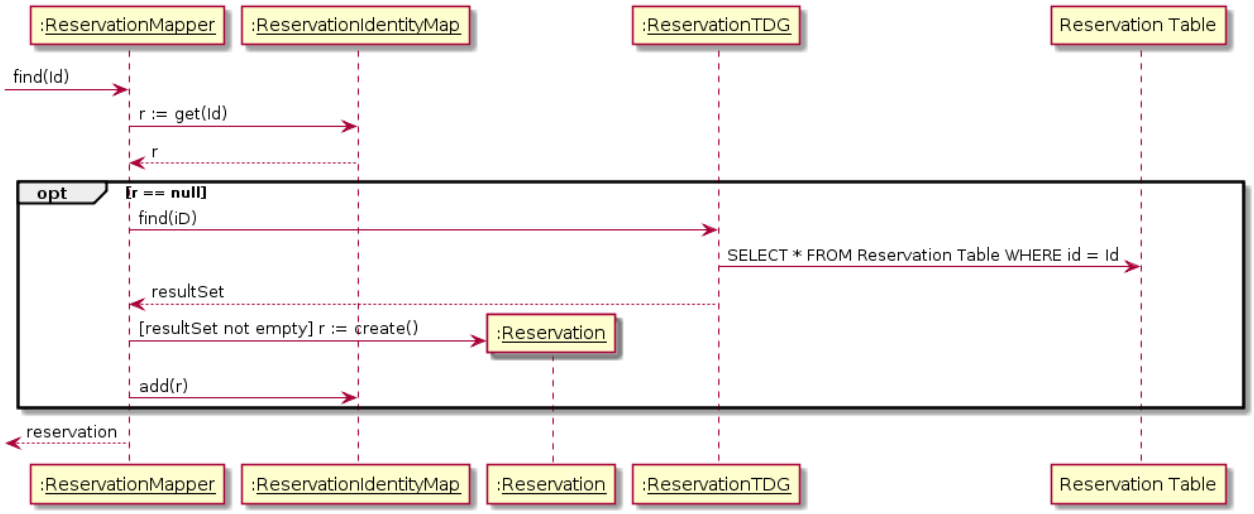


Figure 13: Referenced Sequence Diagram Find Reservation

2.3.7.3 Find Time Slot Reservations

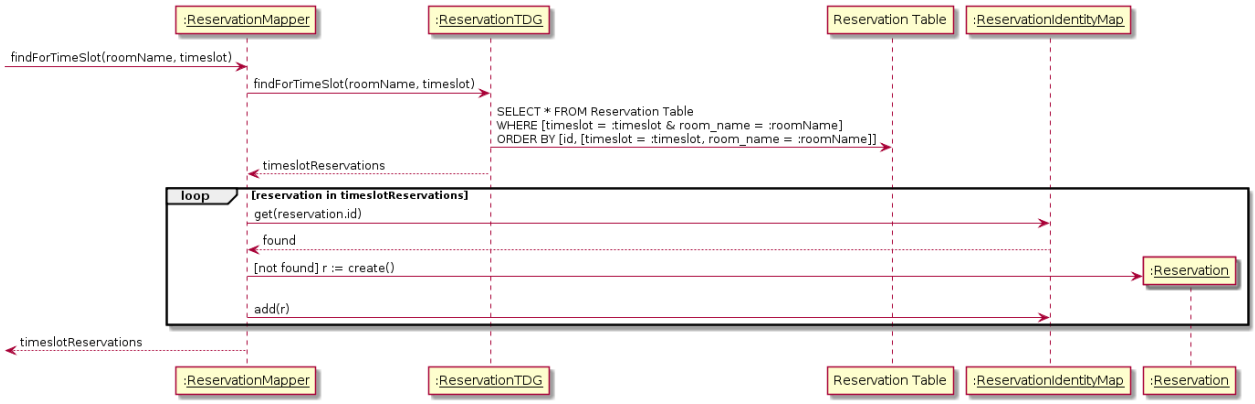


Figure 14: Referenced Sequence Diagram Find Time Slot Reservations

2.3.7.4 Find User Reservations

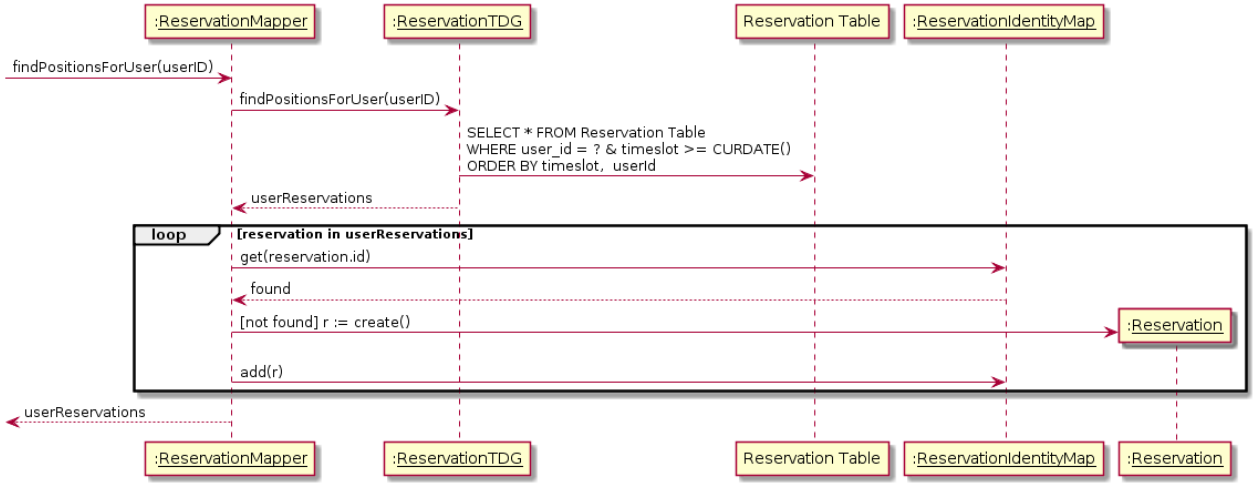


Figure 15: Referenced Sequence Diagram Find User Reservations

2.3.7.5 Complete Work

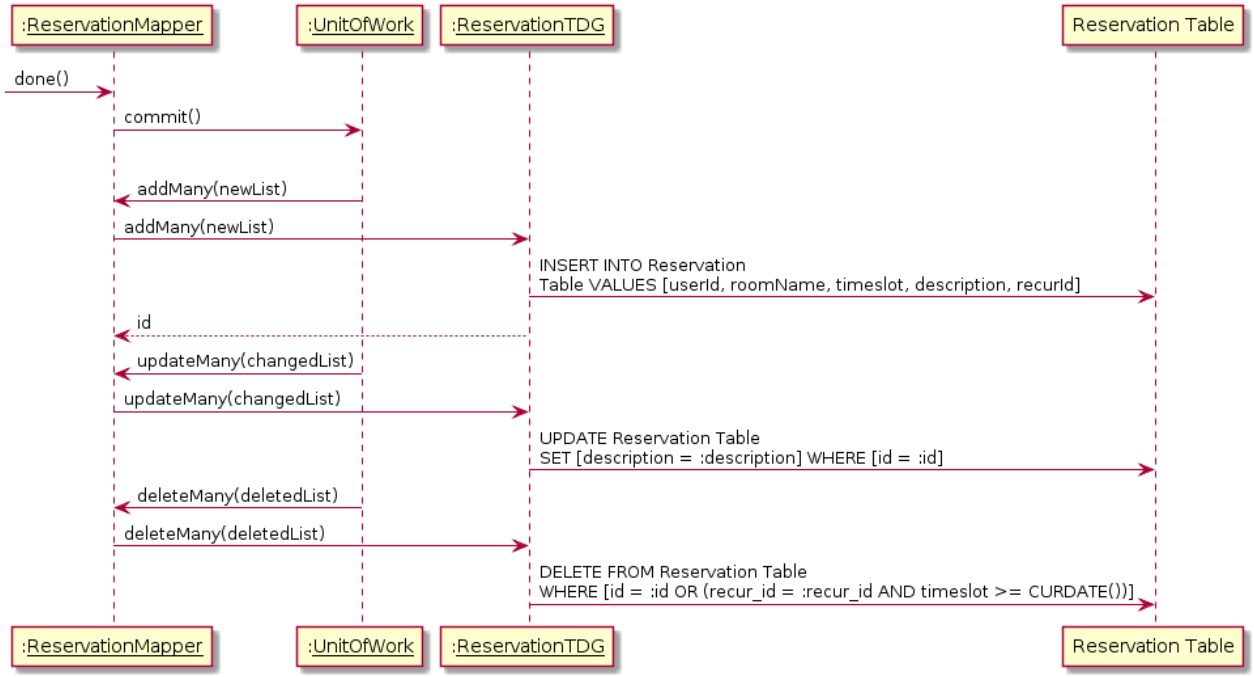


Figure 16: Referenced Sequence Diagram Complete Work

3. Use Case View

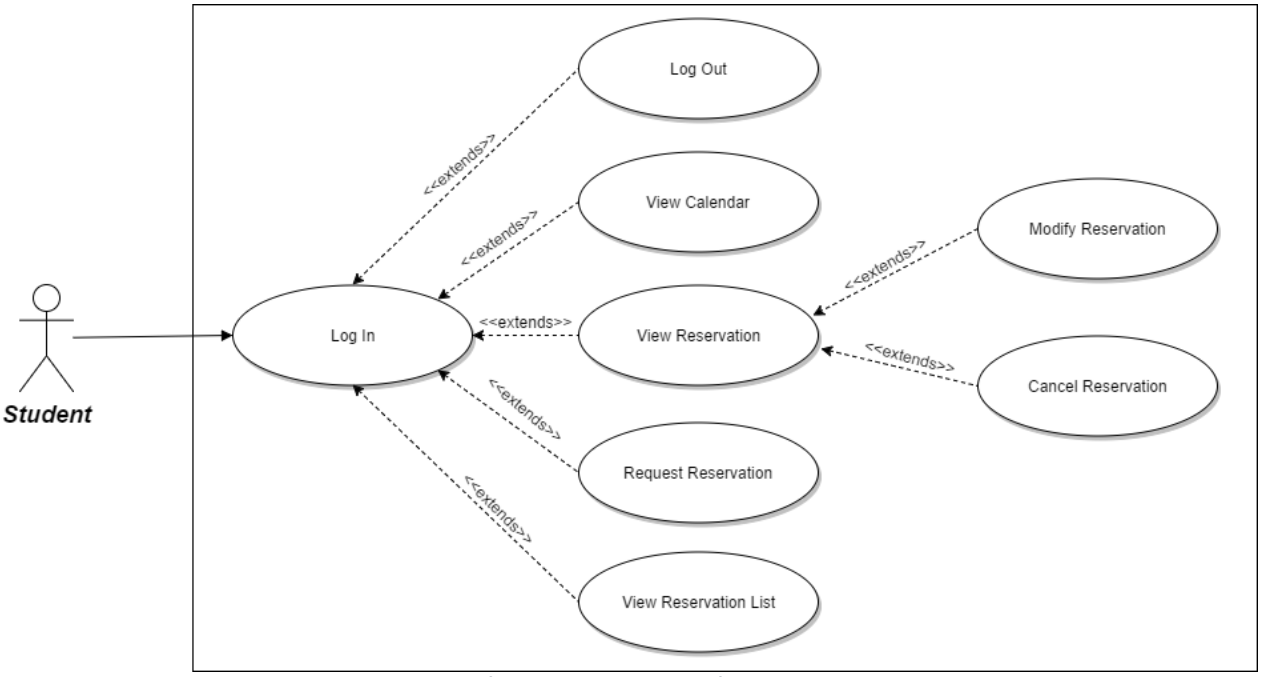


Figure 17: Use Case Diagram

4. Data Model

For persisting data across system requests, a data model is used inside of a relational database management system. This database is accessed using the Table Data Gateway classes within the system. There are five tables in the database, corresponding to the five domain classes that require persistence in the system: Room, User, Reservation, Equipment and ReservationSession.

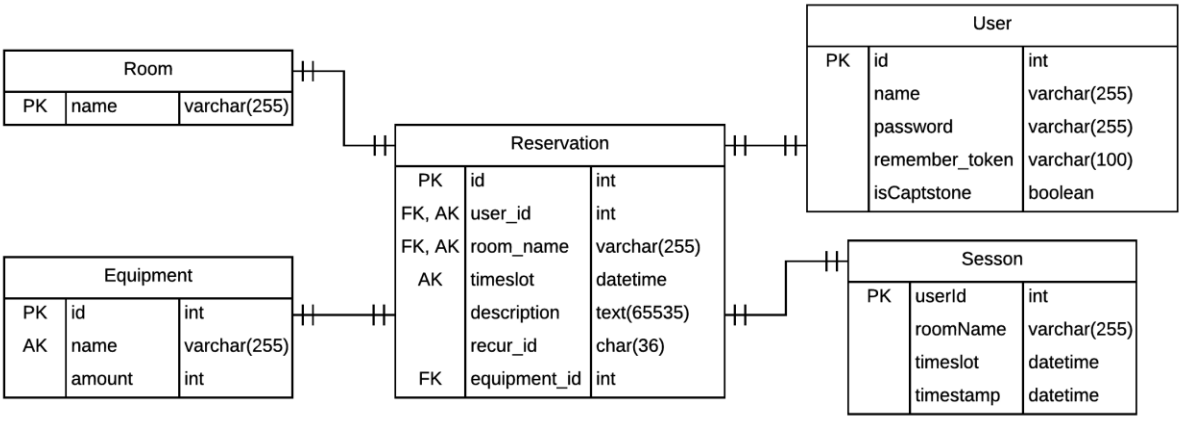


Figure 18: Entity-relationship diagram for the system database

The Room object maps to the *rooms* table, using the room name as its primary key. The User object maps to the *users* table, where the user id is used as the primary key. The table also holds a name and a password field, both strings, for storing the respective attributes of the User domain class, as well as an isCapstone attribute which holds a boolean value.

Then, to represent the relationship between a Room and a User as shown in the class diagram, we use a *reservations* table to map the Reservation class along with foreign key columns to the room's, user's, and equipment primary key columns. That way, we can represent the many-to-many relationship between *rooms*, *users* and *equipment* using a row in the *reservations* table. This table holds additional columns to represent the timeslot of type DATETIME, a reservation's description of type TEXT, and a recur_id to hold the 36-character long UUID string for linking recurring reservations together. A reservation is unique across a specific user_id, room_name and description, however, it is given an additional primary key as an automatically-incrementing integer id to represent insertion order. This id is used to determine waiting list position.

The additions of the Equipment and Session tables did not add much complexity to the system. The Equipment table containing simply the amount of that equipment table, its own id and a unique name. As for the Session table, it is directly associated with a Reservation but its primary key would be the id of a user. It would also have the name of a room as an attribute along with a timeslot and timestamp.

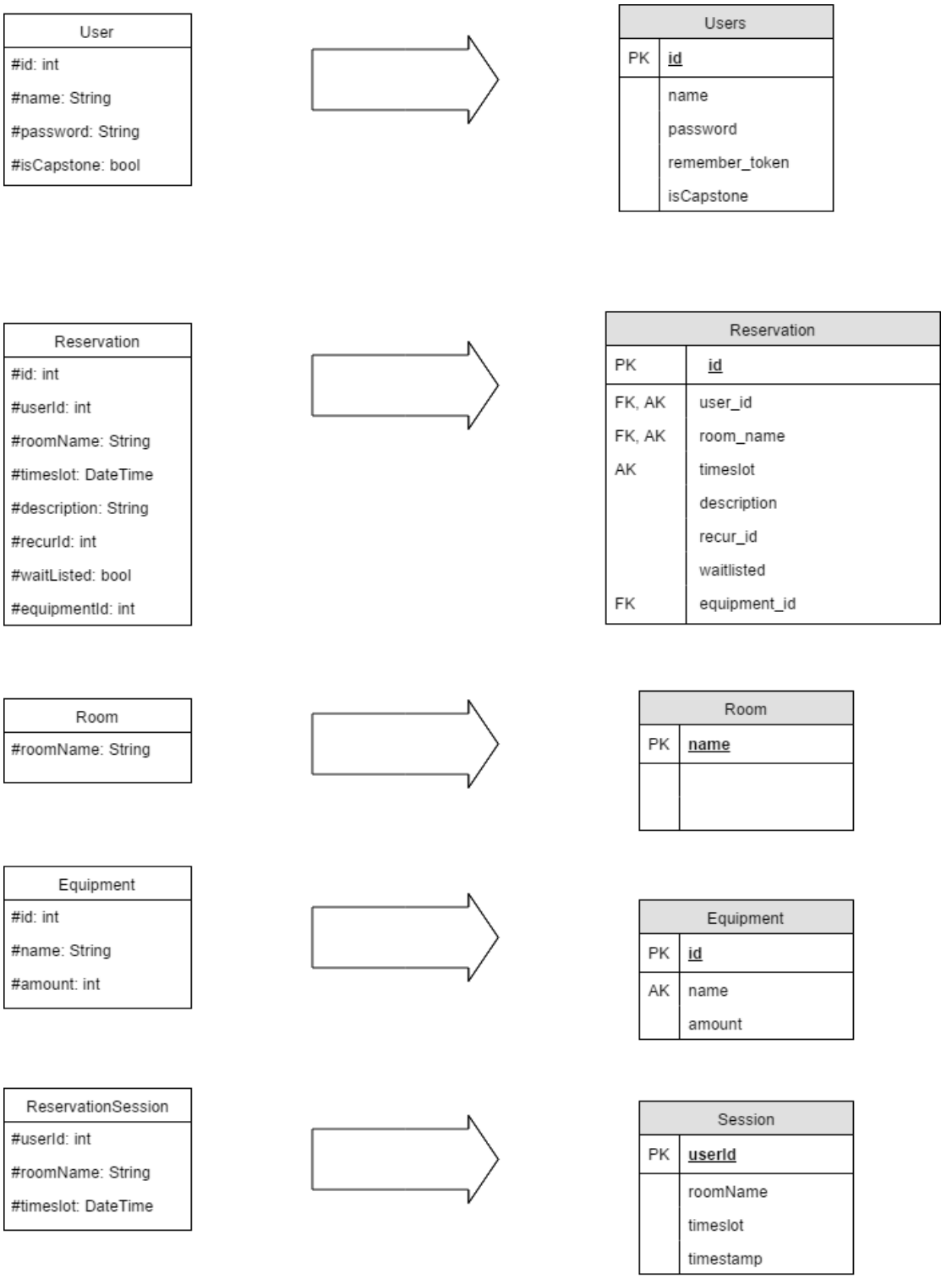


Figure 19: Object Relational Model

The ORM further solidifies the statements made earlier and provides a visual representation of the one-to-one mapping of the domain logic to the data logic.

5. AOP Migration of Interceptor Pattern

Pattern Migration to Aspect Oriented Programming

The new requirements demand that an architectural pattern be used in an aspect-oriented context. This leaves us with the option to implement a completely new pattern or simply take an already existing pattern and migrate it to an aspect-oriented context. Before selecting an option, the pre-existing patterns were analyzed.

Effort analysis of the migration
This whole migration process required a significant amount of reverse engineering as well as re-engineering. By considering what we were given, we had to understand the functionalities and then, examine it carefully to redesign some components so that we may build upon it. More precisely, the effort consisted of first analysing the existing source code to find out the segments that implement the crosscutting functionality. Afterwards, we move on to the transformation of the existing code into an aspect-oriented reformulation. Thus, to fully take advantage of the potential benefits of the AOP style of programming there is definitely a need for migration support of existing systems.

PHP AOP Framework
The selection of frameworks to implement the migration was filtered down to two selections: AOP and Go! Aspect-Oriented. After reviewing both frameworks, it was established that both function in a very similar fashion, the main differences were found in their installation and documentation. Go! Aspect-Oriented offered more documentation and tutorials online which will help improve the learning process for the team. The installation of AOP required pecl, a repository of PHP extensions. The installation of Go! Aspect-Oriented was evaluated to be much simpler as it relied on Composer, which was already installed on all our machines as it was needed to install the original Chronos project.

Installation Process
Once Composer is installed on the machine, downloading the framework simply consists of running the following line in bash:

```
$ composer require goaop/framework
```

The next step is to create and configure an Aspect kernel for our project. Once ready, aspects can be created and registered to the kernel. The full installation guide can be found at : https://github.com/goaop/framework

Migration Plan
The migration plan is divided into three sections:

1) Architecture pattern identification
2) Code segment identification
3) Refactor phase

The first phase consists of identifying the architectural pattern - the interceptor. The second phase identifies the code segments that are most appropriate for migration to aspects. It is where the source code is analyzed carefully to locate suitable aspects. Afterwards, during the refactoring phase the code is transformed for the crosscutting concerns to be comprehended by separate aspects instead of the original classes.

Interceptor Pattern

We perform our OO to AO migration by implementing the interceptor pattern. There are two main benefits of using this pattern in aspect oriented:

1) It allows services to be added transparently on the existing system. This means the integration or removal of the aspects will not influence the core functionality of the system. With little to no code modifications to the existing patterns implemented, there is no added coupling. As well the cohesion of the system stays high because there is separation on of concern among that old and new requirements added.
2) The aspect will be trigger automatically when certain events occur. It is not called explicitly in the call thus keeping the coupling low.

The Interceptor pattern consists of running pre-processing and post-processing functions at given pointcuts. The two main parts of the Interceptor pattern are:

- Dispatcher: Adds and removes specific interceptors. Calls the interceptor method when an event occurs.
- MethodInterceptor: Runs the pre-processing and post-processing functions.

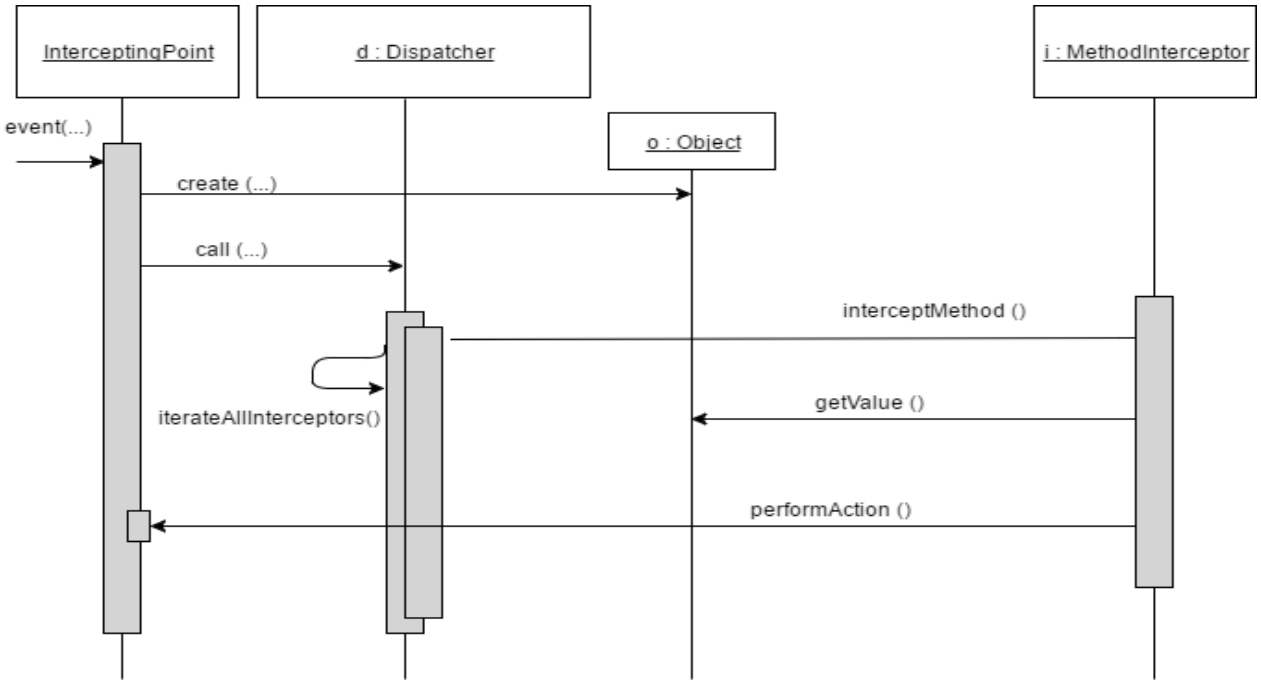


Figure 20: Interaction Diagram for Interceptor pattern

In comparison, it is very similar to the AOP Go! Framework which contains two main actors, ApplicationAspectKernel and Aspect:

- ApplicationAspectKernel: Register and removes specific aspects.
- Aspect: Provides method interception and contains service and functionality.

The chosen framework to implement the Interceptor pattern was AOP Go!. The Go! framework provides all needed functionality to implement new logic transparently on the existing system. The frameworks allow the adding of aspects that would be called at given pointcuts in the system. This keeps the overall architecture of software much easier to grasp and does not add unnecessary coupling between classes. The following diagram demonstrates how the Go! framework implements the Interceptor pattern.

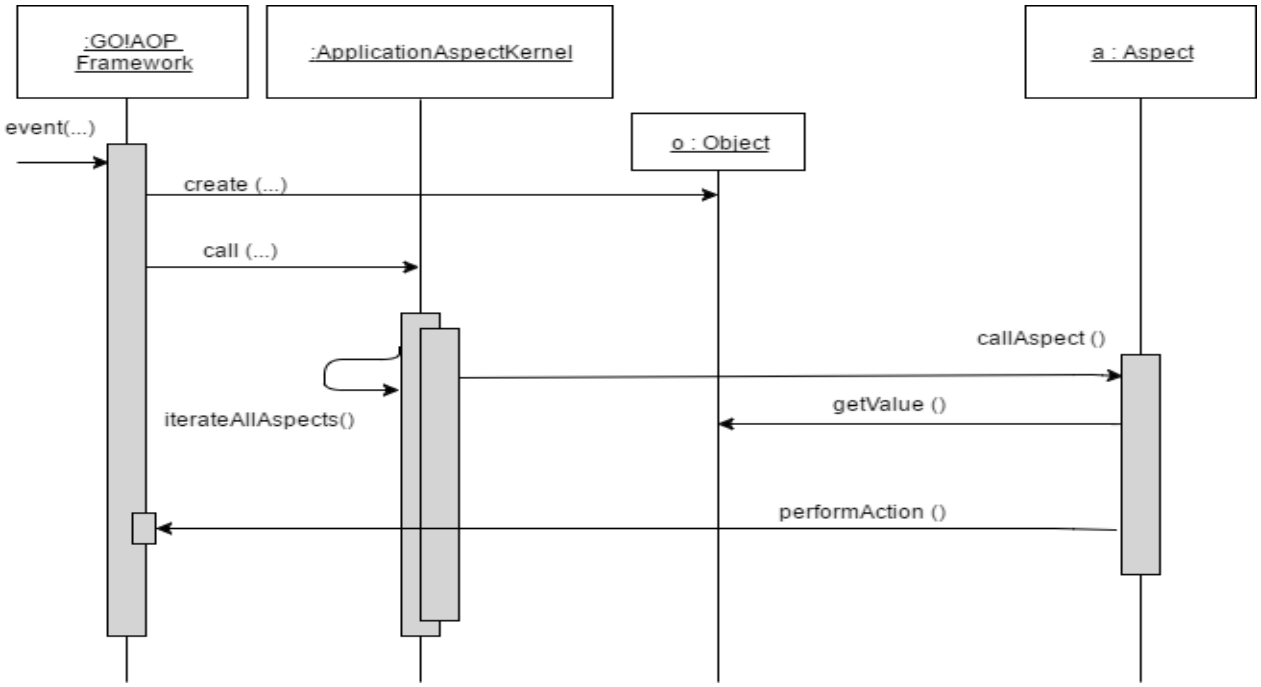


Figure 21: Interaction Diagram for Go! framework

The pattern migration will consist of five different aspects which all have their own role and responsibility in term of offering a better separation of concerns:

1) CalendarAspect: date input validation
2) LoginAspect: login validation on the format of the input
3) ReservationAspect: validate reservation requirement
4) EquipmentAspect: promote a student from the waiting list; promote when an equipment becomes available
5) LoggerAspect: track system event in developer mode

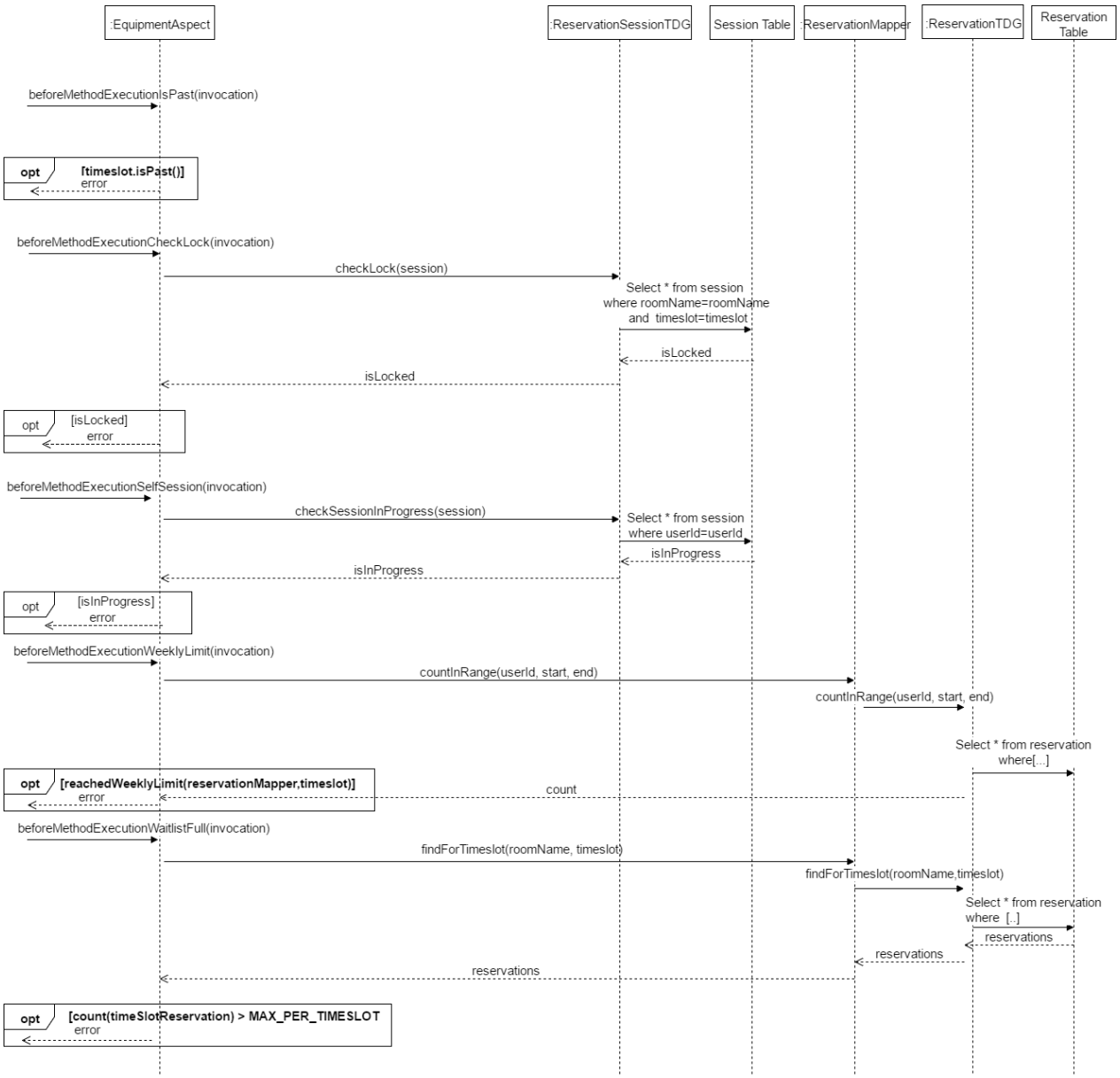
1. Reservation Aspect

The use of AOP will serve to intercept the *'modifyReservation'* method and check few requirements *before* the execution of the method. The requirements include: check whether the requested reservation exists and is owned by a user. This separates the requirement insurance checking process from the actual modification. That way, the crosscutting concern is better

modularized since they are no longer tangled together. The aspect “ReservationAspect” will be created.

In addition, when a user creates a reservation, the system first requests a *reservation making* form which is handled by the method *showRequestForm* in the *ReservationController*. The method was initially designed to open a reservation session for the user in order to handle reservation resource concurrency. However, in addition to this functionality, this method also checks a series of conditions to generate different scenarios that will either allow the reservation form request or deny it. There are several conditions under which the request will be denied:

1) If another student is reserving the same room on the same timeslot
2) If the current user has a reservation session underway
3) If the current user has reached the weekly limit of reservations
4) If the waitlist for the reservation is full
5) If the requested reservation is in the past



With the AOP migration, instead of calling the *showRequestForm* method directly, the *ReservationController* will use the *Interceptor pattern* to check all the constraints around the execution of the method. This change utilizes separation of concerns to provide better code maintainability. This migration allows the base code to remain *oblivious* to this newly introduced aspect, allowing the functionality to be developed independently of it. Thus, in "ReservationAspect", the following conditions will be checked:

Condition 1) Check if another student is reserving the same room on the same timeslot
Condition 2) Check if the current user has a reservation session underway
Condition 3) Check if the current user has reached the weekly limit of reservations
Condition 4) Check if the waitlist for the reservation is full
Condition 5) Check if the requested reservation is in the past

2. Logger Aspect

Logging is not related to a particular class or application component, and involves constantly including the logging module and independently calling it over and over again within method definitions. An example of this is logging system events while in development mode, which are not to be visible in a production environment. By separating logging to an aspect-oriented implementation, the concern is treated completely separately of the application code and can be extracted simply by deactivating the aspect. This also results in cleaner code within the involved classes by not including code that is unrelated to the functioning of the system.

For instance, in development mode it is useful to log the parameters passed to methods of the reservations controller in order to keep track of reservation operations such as cancelling, requesting and modifying reservations. Instead of repetitively calling the logger at the start of each of these methods they can be abstracted into a single aspect.

Creating a *LoggerAspect* class removes the cross cutting of the logger interface in the involved classes/methods. In the case of this example, the pointcut will be all the accessible methods in the *ReservationController* class. The advice is the definition that runs *before* these methods.

This means that the original methods do not need to be aware or explicitly call any logger functions.

If certain methods need different logging structures than others, multiple definitions (advice) within the logging aspect can be added with different pointcuts referring to different, or a more specific, set of methods rather than having a single one as used in this example. So based on different types of requests, we can have different logging aspect methods.

3. Equipment Aspect

One of the advantages of using aspects is to treat the system as a black-box and not modify current implementation. When equipment was added onto the system, it affected the wait listing and created a dependency between reservations across multiple rooms. This means that if a reservation is waiting on another one to be active, it needs to be notified if that other reservation gets cancelled. Basically, the flow is as follows:

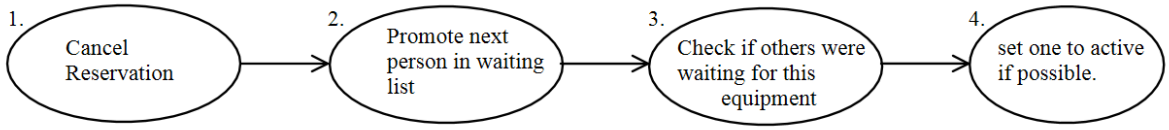


Figure 22: Diagram for Waitlist flow

The only operation that actually involves the current user is the first one, namely the cancellation of the reservation. The rest are computations done for the overall system status. By using AOP, the '*cancelReservation'* method can be intercepted in order to do whatever needs to be done upon a cancellation. In this case, these other functionalities can be done ***after*** the successful completion of the method, meaning after cancellation of the reservation. As previously stated, this allows the underlying implementation to stay the same and separating it from the whole process of promoting other reservations due to equipment dependency. The aspect "*EquipmentAspect*" is created and can be extended to involve other equipment operations. The method *beforeCancellation* will intercept the parameters and store the reservation object **before** the method for future interactions,

and the *afterCancelReservation* will do the necessary work **after** the method completes, namely steps 2, 3, and 4.

Thus, the original *cancelReservation* method behaves as it did before updating the system and is significantly smaller in scope. It only deletes the reservation.

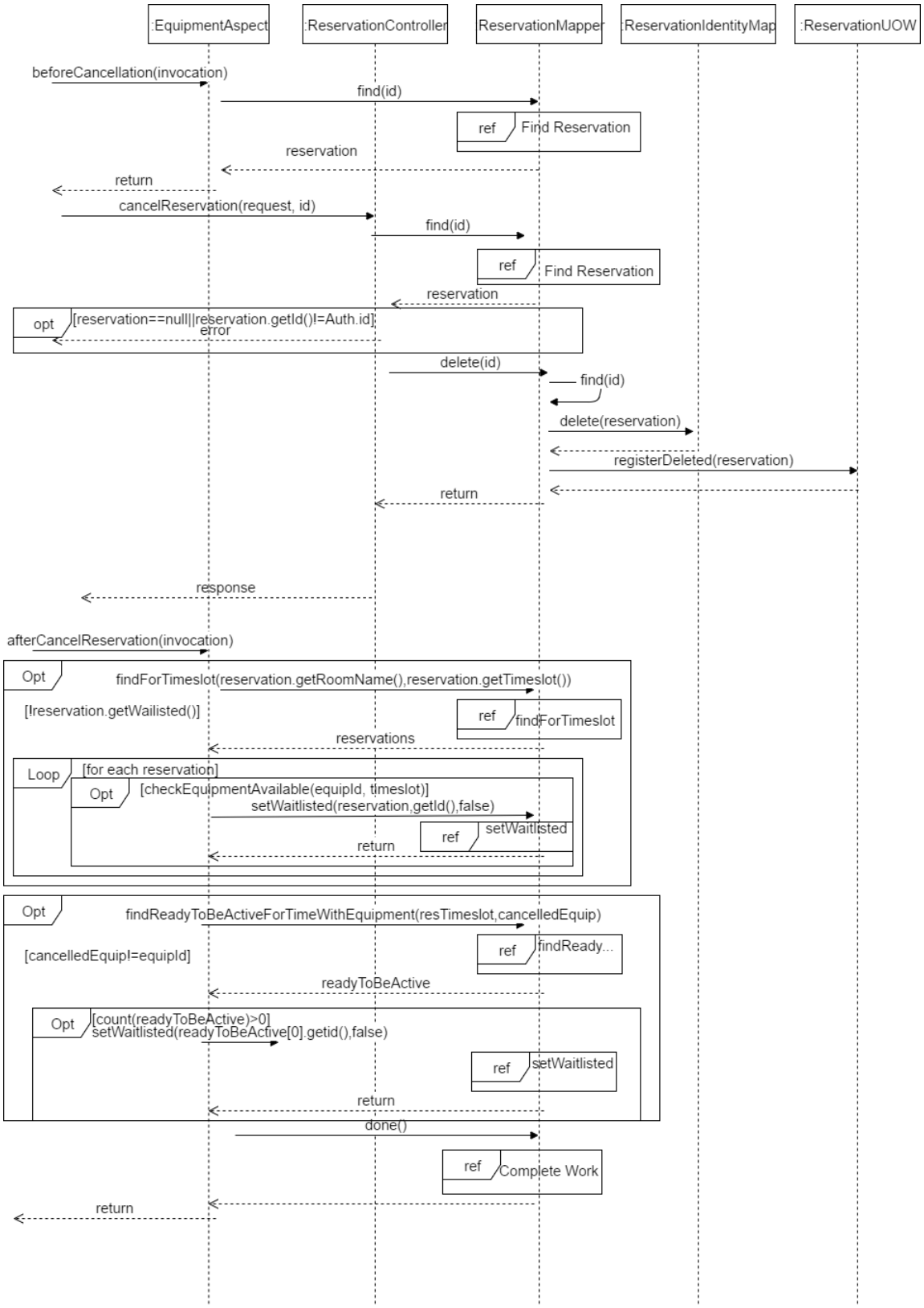


Figure 23: Sequence Diagram for Cancel Reservation

The sequence diagram above shows the system interactions when the cancel Reservation method is called in the Reservation Controller. Before the method even executes, the interaction commences with the Equipment Aspect intercepting the method invocation, finding the reservation using the Reservation Mapper and storing it statically. Once the method terminates, the post-condition in Equipment Aspect uses that statically stored reservation to complete additional logic based on the new equipment available and users on the waitlist.

4. Calendar Aspect

One simple yet effective use of Aspect Oriented Programming in conjunction with this application is to use an aspect to handle input validation on the calendar controller. When the view calendar method is called, the request parameter contains a date input. This input must be validated in multiple ways. Firstly, the input itself is validated on the front-end to ensure that the appropriate format is followed. Secondly, the date parameter must not contain a date which has already past. This check can be provided by an aspect using a pre-condition for the view calendar method. The precondition executes beforehand and checks if the date is in the past and if so, redirects the user back to the page displaying an appropriate error message. This removes the potential cross-cutting concerns in the view calendar method by separating validation from execution logic and placing the former in an aspect.

The arguments of the method invocation of the view Calendar method are analyzed by extracting the date argument and setting the default to the current date if not specified. Otherwise, it converts the date to a year-month-day format and checks whether it is past or not.

5. Login Aspect

Another use for AOP is the separation of validation of user login credentials with the action of logging in. The validation of user credentials is a two-fold action. Firstly, the fields must be not empty and follow the format specified, and secondly, they must correspond to credentials which exist within the database. To ensure that the LoginController does not take care of the trivial matter of ensuring that the fields are entered correctly, the LoginAspect will intercept the method call and ensure the format is respected before proceeding. While there may be some front-end checks to ensure the information is valid, adding this aspect provides further robustness to the system. The check uses a precondition on the login method in LoginController which executes before the method. If the id or password input parameters are null or if the id does not fit the 8-digit format, the login process will fail immediately, redirecting the user back to the login form and displaying the appropriate error. This demonstrates separation of crosscutting concerns by having the aspect handle formatting validation and the controller handle database validation.

The arguments of the method invocation of the login method are analyzed by extracting the user's id and password. A check is completed on the two parameters by checking if they are null and if the id is of length 8 characters. If not, the user is redirected with an error.

6. Installation Manual

Installing Chronos on a server is a straightforward procedure and only requires about 1 to 2 hours of time.

6.1 System Requirements

To install Chronos, you will need an Internet-connected server running a PHP-capable web server and database software. The recommended tested requirements are listed below.

- Apache 2.4 or nginx 1.6
  - Configured to serve files from a "**public**" directory under the installation directory
- PHP 7.0
  - Required extensions: OpenSSL, PDO, Mbstring, Tokenizer
- MySQL 5.6

For this installation manual, an Ubuntu Linux 15.10 installation will be assumed. We will also assume the following paths:

- Installation path is **/var/www/chronos**
- Document root of web server **/var/www/chronos/public**
- URL of website is **http://chronos.chrs.pw**
- Web server user is **www-data**
- MySQL database address is **localhost**

1. Please run the following commands to check your environment:

```
dev:~$ php --version
PHP 7.0.5-2+deb.sury.org~wily+1 (cli) ( NTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
    with Zend OPcache v7.0.6-dev, Copyright (c) 1999-2016, by Zend Technologies

dev:~$ mysql -uroot -p -e "SELECT VERSION();"
Enter password:
+-------------------------+
| VERSION()               |
+-------------------------+
| 5.6.28-0ubuntu0.15.10.1 |
+-------------------------+
```

2. Ensure that your web server and PHP are enabled on your server by creating a sample "index.php" file in the document root and load the website in your web browser.

```
dev:~$ cd /var/www/chronos/public
dev:/var/www/chronos/public$ echo "<?php phpinfo(); ?>" > index.php
```

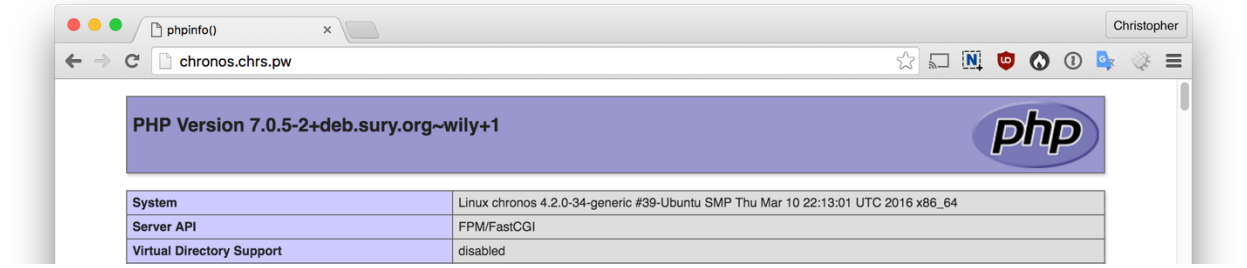


Figure 24: PHP test file output

6.2 Downloading Chronos

Obtain a Chronos release by downloading it from the official repository, and extract it into your installation directory.

1. Change your directory to the installation path:

```
dev:~$ cd /var/www/chronos
```

2. Download the Chronos release:

```
dev:/var/www/chronos$ wget \
  https://github.com/Shmeve/soen343-emu/releases/download/v0.0.1/chronos.tar.gz
```

3. Extract the Chronos system files to the installation path:

```
dev:/var/www/chronos$ tar --strip-components=1 -zxvf chronos.tar.gz
```

6.3 Setting up the Database

After extracting the Chronos system files, it is necessary to create a database and a user for Chronos.

1. Log into your MySQL database as an administrative user, and issue the following commands:

```
dev:~$ mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.6.28-0ubuntu0.15.10.1 (Ubuntu)

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> CREATE DATABASE chronos;
Query OK, 1 row affected (0.00 sec)
```

```
mysql> CREATE USER 'chronos'@'localhost' IDENTIFIED BY 'password';
Query OK, 0 rows affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON chronos.* TO 'chronos'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)

mysql> SET GLOBAL event_scheduler = ON;
Query OK, 0 rows affected (0.05 sec)
```

This has created a database "**chronos**" and a user "**chronos**" with password "**password**" with privileges on that database, ready for use with Chronos. Make sure to set the global event scheduler option to on, so the expired session can be handled by the database automatically.

Now, you must configure Chronos with this database information.

2. First, copy the example ".env.example" configuration file:

```
dev:/var/www/chronos$ cp .env.example .env
```

3. Edit the ".env" file and change the DB variables to match the appropriate values:

```
DB_HOST=localhost
DB_DATABASE=chronos
DB_USERNAME=chronos
DB_PASSWORD=chronos
```

6.4 Install Chronos

Most of the Chronos installation is done via Composer, PHP's package manager. An install script is included to accelerate the process, which will create all necessary database tables and install all dependencies.

1. Execute install.sh:

```
dev@:/var/www/chronos$ ./install.sh
All settings correct for using Composer
Downloading 1.2.0...

Composer successfully installed to: /var/www/chronos/composer.phar
Use it: php composer.phar
Application is now down.
Loading composer repositories with package information
Installing dependencies from lock file
  - Installing symfony/finder (v3.1.7)
    Loading from cache

[…]

Generating autoload files
> php artisan clear-compiled
> php artisan optimize
Generating optimized class loader
```

```
Compiling common classes
Migration table created successfully.
Migrated: 2016_11_16_000000_create_users_table
Migrated: 2016_11_17_045934_create_rooms_table
Migrated: 2016_11_17_051557_create_reservations_table
Generating optimized class loader
Compiling common classes
Application cache cleared!
Application is now live.
```