SOFTWARE ARCHITECTURE DOCUMENT

for

The Force Awakens

PREPARED BY:

Name	Student ID	Email
Erin Benderoff	27768478	erin.benderoff@gmail.com
Syrine Krim	29773118	syrine_krim@hotmail.com
Ryan Lee	27752504	rswllee@hotmail.com
Jeremy Melnyk	26374603	jeremyjm91@gmail.com
Dimitri Topaloglou	29358269	dtopaloglou@gmail.com
Kevin Yasmine	27195346	kevinyasmine@gmail.com

Instructor: Dr. Constantinides

Course: SOEN 344

Date: April 4, 2017

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 PURPOSE AND SCOPE OF THE SAD

This document provides an overview of the architectural design of The Force Awakens room reservation system. The back-end architecture of the system was refactored extensively in order to provide a solid framework for implementing the new requirements for SOEN 344. This document provides a detailed summary of the refactoring that was done.

The SAD describes three views of the system: use case view, logical view and data view. Use case view describes the different interactions between the user and the system and is represented by the use case model previously shown in the SRS document. Logical view describes the functionality provided by the system to its users in terms of use case realizations. It is represented by UML class diagrams and sequence diagrams. Class diagrams describe the class structure for each subsystem, including their attributes and methods, as well as the relationships between classes and between different subsystems. Sequence diagrams are a type of interaction diagram that show the sequence of messages sent between objects in the system. Finally, a data view is provided to show the relationships between data entities in the system; it is represented by an entity-relationship model.

An additional architectural requirement for SOEN 344 was to migrate one pattern from an object-oriented to an aspect-oriented context. This migration is described later in this document.

The SAD is intended to be used by any stakeholder of the system, though, unlike the SRS, it requires a certain level of expertise in software development to understand the precise architectural design details.

1.2 DEFINITIONS, ACRONYMS, AND ABBREVIATIONS

SAD: Software Architecture Document

TDG: Table Data Gateway

UoW: Unit of Work

UML: Unified Modeling Language

ER: Entity Relationship

OOP: Object-Oriented Programming

AOP: Aspect-Oriented Programming

2 OVERALL DESCRIPTION

The architecture that this project follows is the Client-Server Architecture, where the server is broken into 3 layers: Presentation, Application and Data Source layers. This was all done in PHP on the server end, with HTML, CSS and Javascript on the client side. The client side uses a single page architecture to focus functionalities on a sole page. For reusability purposes, the system was implemented using a public API that is compatible with any third party client. Data Mappers and Table Data Gateways also facilitate the reusing of code, as they decrease coupling and make a clear separation between the domain business objects and the data service objects. Within the relational database, the unit of work is used to reduce the amount of calls to it.

The lock in this system when creating a reservation is done with the Pessimist approach. A user's request to change, cancel or create a reservation is not processed until they have been allocated the necessary resources. The lock lasts for 120 seconds, after which the user will be timed out and other users are permitted access to make changes once more. For security reasons, the system hashes user passwords in a database and it uses recent native PHP query builders to combat SQL injection attacks.

3 REFACTORING SUMMARY

- Got rid of most functional code found in most pages since it was very difficult to follow with no comments on logic. PHP, JavaScript, and HTML was mixed together in pages that handle several concerns (user input, validation, SQL statements, etc.)
- Refactored logic from functional code into cohesive classes that deal with certain aspects of the business logic; e.g. checking for conflicts on a new reservation was moved into the ReservationManager class. This allows the logic to be re-used between create, modify, and delete reservations.
- Refactored TDG and Mapper classes so that they conform to specific interfaces that follow basic CRUD functions. We used inheritance and abstract classes to increase reusability between mappers and TDG objects. Mappers now build Domain Objects based on requested queries from the TDGs and the data returned by the TDGs.
- Replaced all SQL queries. These queries were vulnerable to SQL injections as their parameters were not sanitized. Incorporated the DBAL framework that wraps SQL queries into objects. The DBAL framework wraps PHP's native PDO library which insures clean and safe queries.
- Added a settings file that can be parsed by a singleton object in order to quickly retrieve app-wide settings. This made it easier to reuse settings.
- Removed redundant php sessions that might have caused session clashes would could potentially log the user off for no reason.
- Cleaned up javascript by using the already built-in jQuery library. Most javascript was converted to jQuery in order facilitate Ajax calls between the UI and the backend. This allows for future modularization of jQuery code.
- Added PHP autoloaders that allow classes to be instantiated and found at runtime rather than calling file imports across the application. This allowed for any object to communicated with other objects with having to include the class' file destination.
- Cleaned up much of the UI using jQuery modals rather than the included Bootsrap modals. This allowed to us to use jQuery functionality mixed with the modal windows as they were all based on the same jQuery library.
- Rebuilt the database and ER schema to use date time instead of time slots e.g reservations can be made at the minute level (10:11 to 12:49) rather than in blocks. As long as there is no time overlap for the same room on the same date, than a reservation can be made active.
- Revamped wait list logic to allow for wait-listed reservations to be automatically moved to active when the requested date, start, and end is available due to a deletion or modification.
- Added function comments to all new classes and utilities that were created by separating the coupled functionality into reusable components.

4 USE CASE VIEW

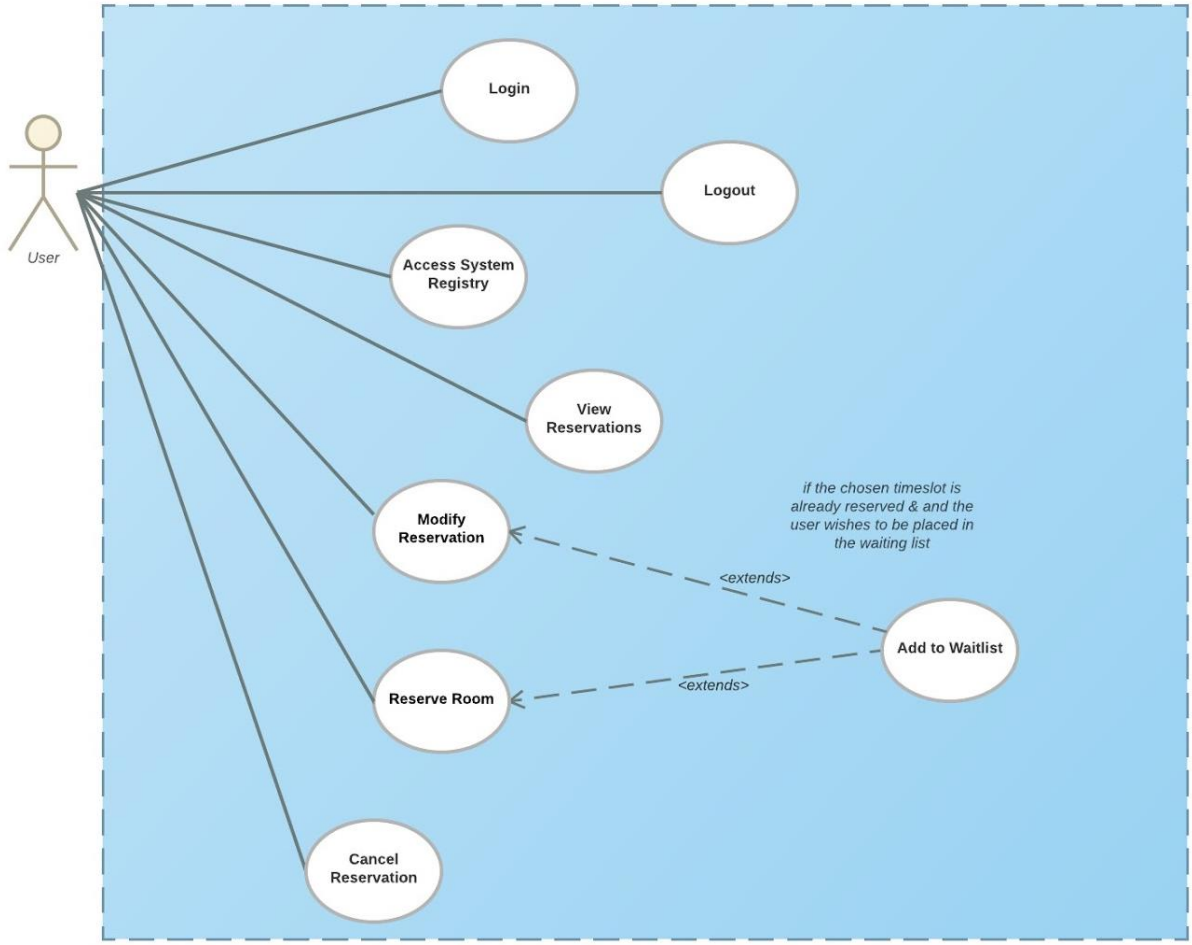

Figure 1: Use Case View

5 LOGICAL VIEW: CLASS DIAGRAMS

5.1 OVERALL ARCHITECTURE VIEW

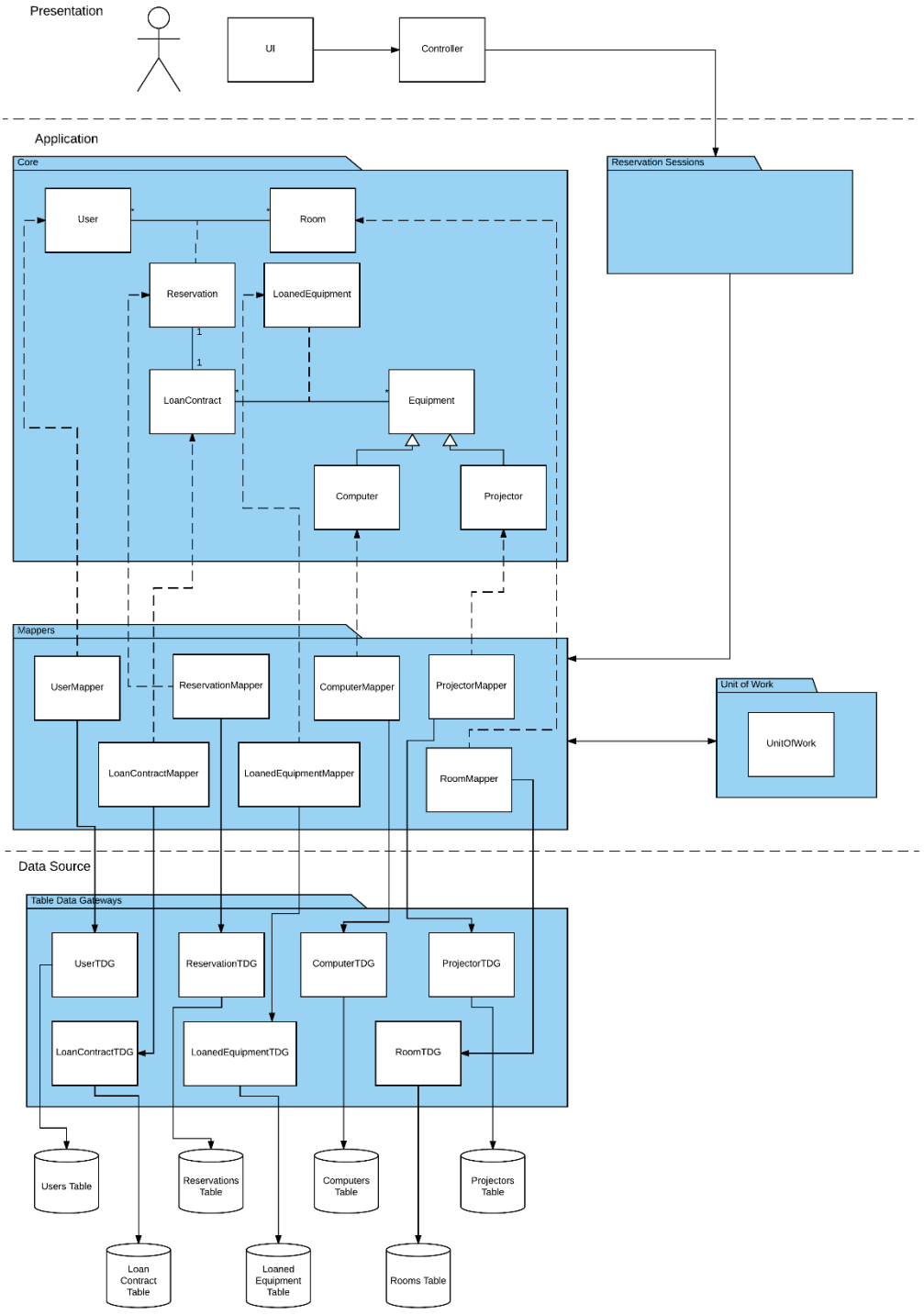

Figure 2: Class Diagram - Overall Architecture View

4.2 ARCHITECTURALLY SIGNIFICANT DESIGN PACKAGES

4.2.1 CORE

The core package describes the domain-level objects, derived from the domain model of the SRS.

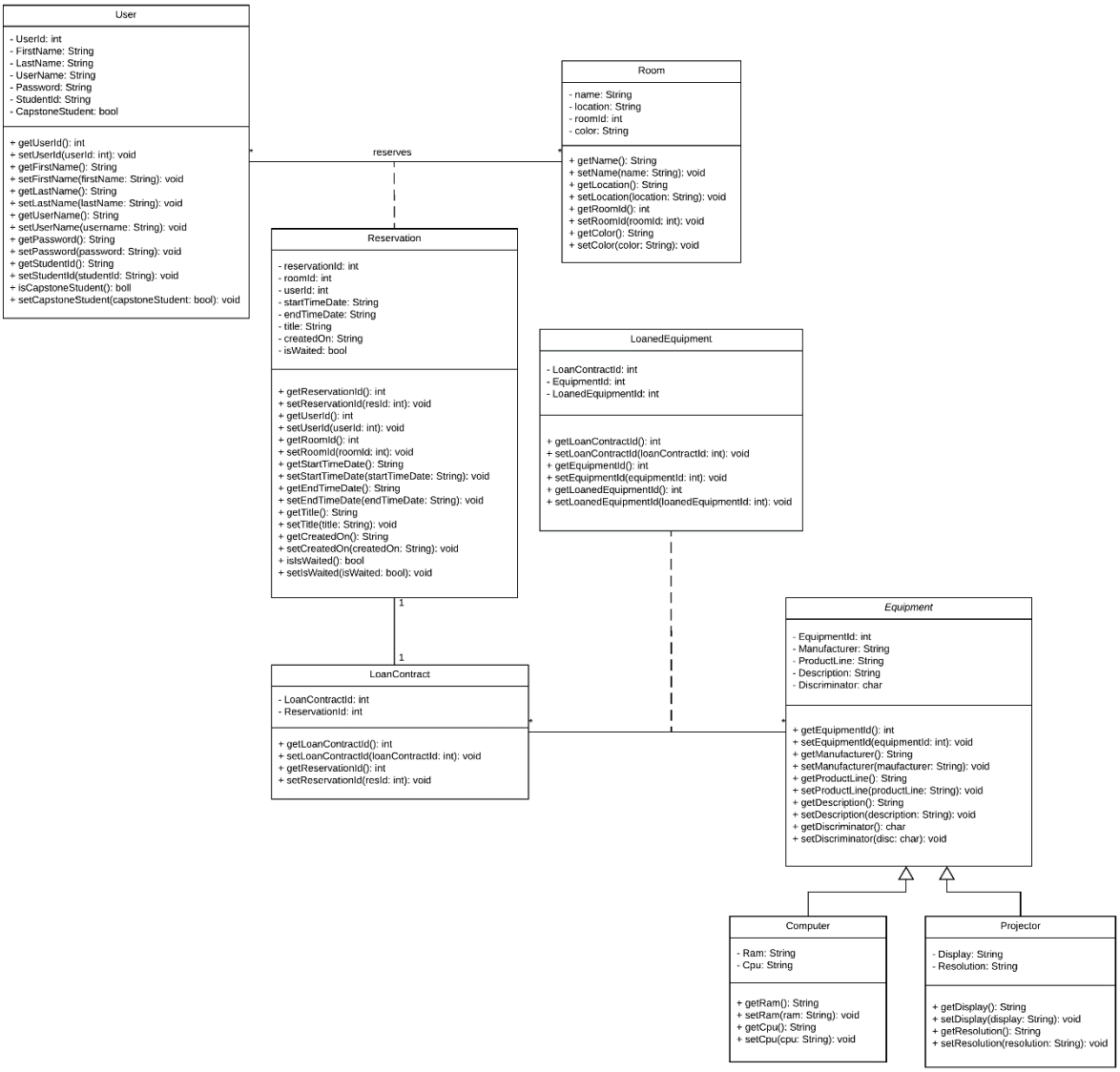

Figure 3: Class Diagram - Core Package

4.2.2 RESERVATION SESSIONS

The classes of the Reservation Sessions package contain the system operations called directly from the Presentation layer.

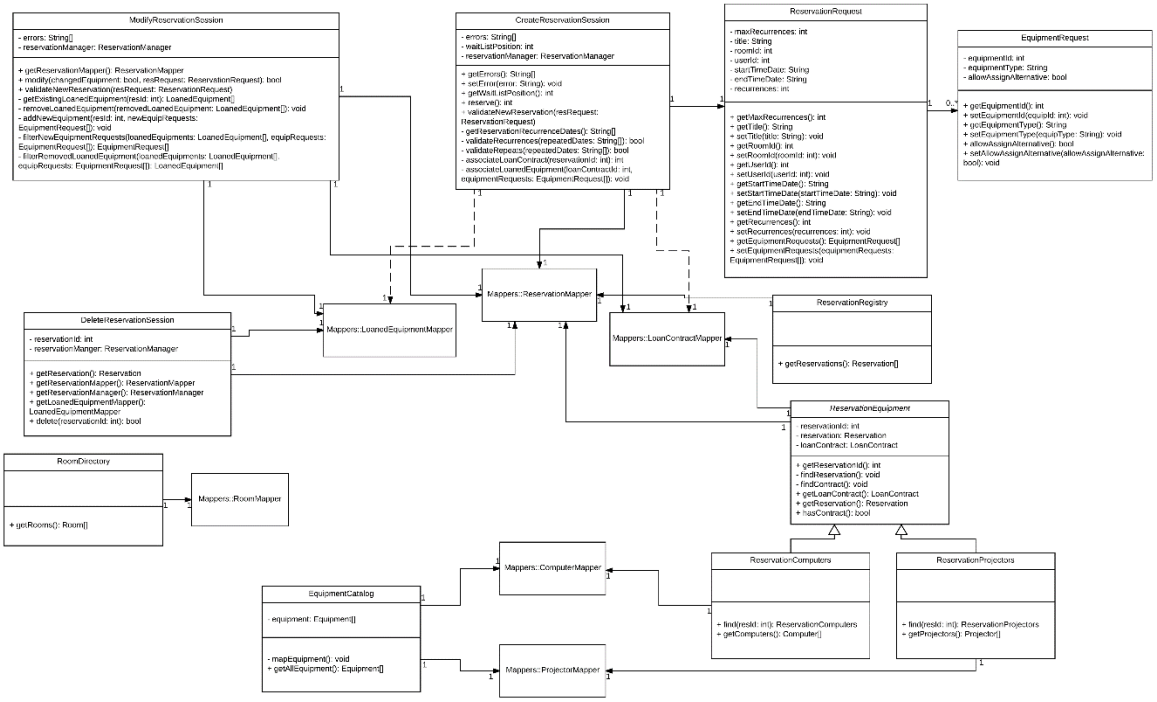

Figure 4: Class Diagram – Reservation Sessions Package

4.2.3 RESERVATION MANAGER

The Reservation Manager package contains helper classes that aid in creating, modifying and deleting reservations. It helps with conflict checking, waitlisting and managing equipment rentals.

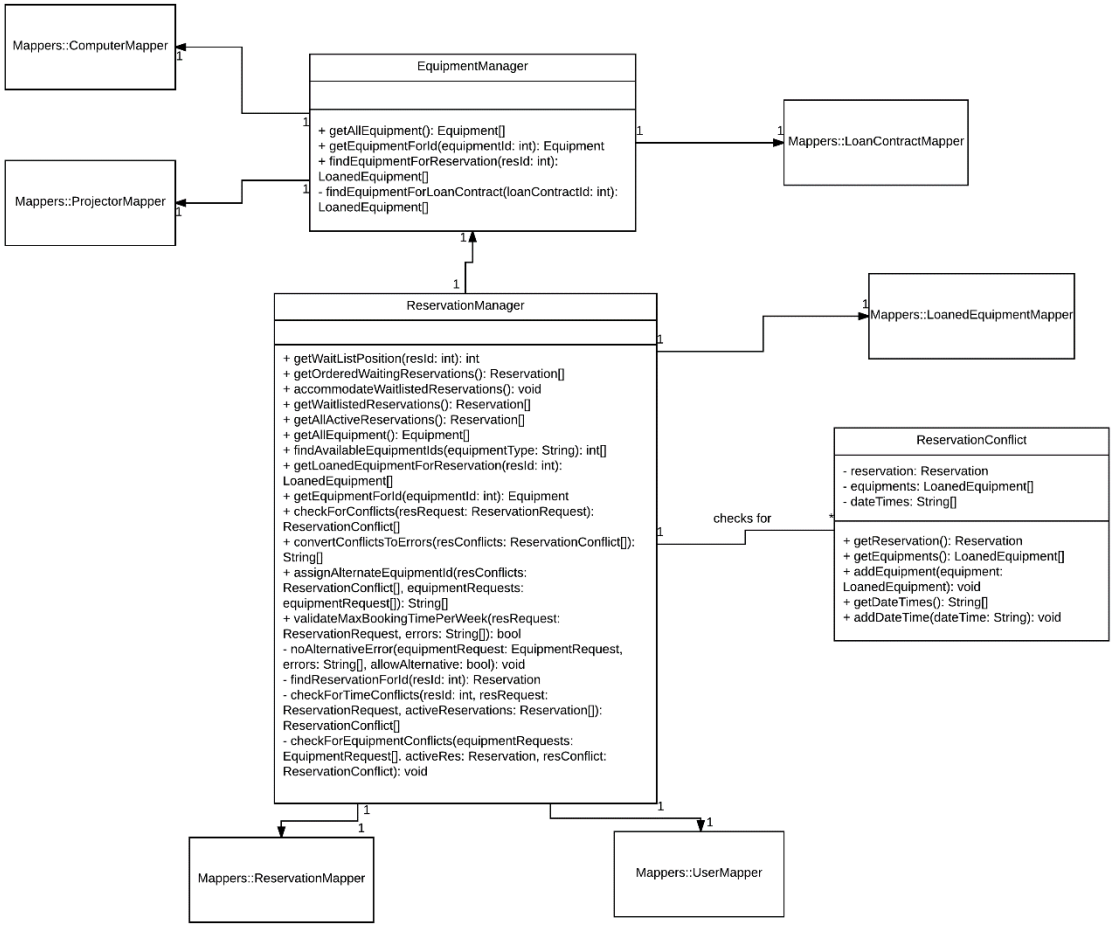

Figure 5: Class Diagram – Reservation Manager Package

4.2.4 DATA MAPPERS

The Data Mappers package contains classes that move data between the domain objects, table data gateways and the unit of work, in order to keep these packages independent of each other.

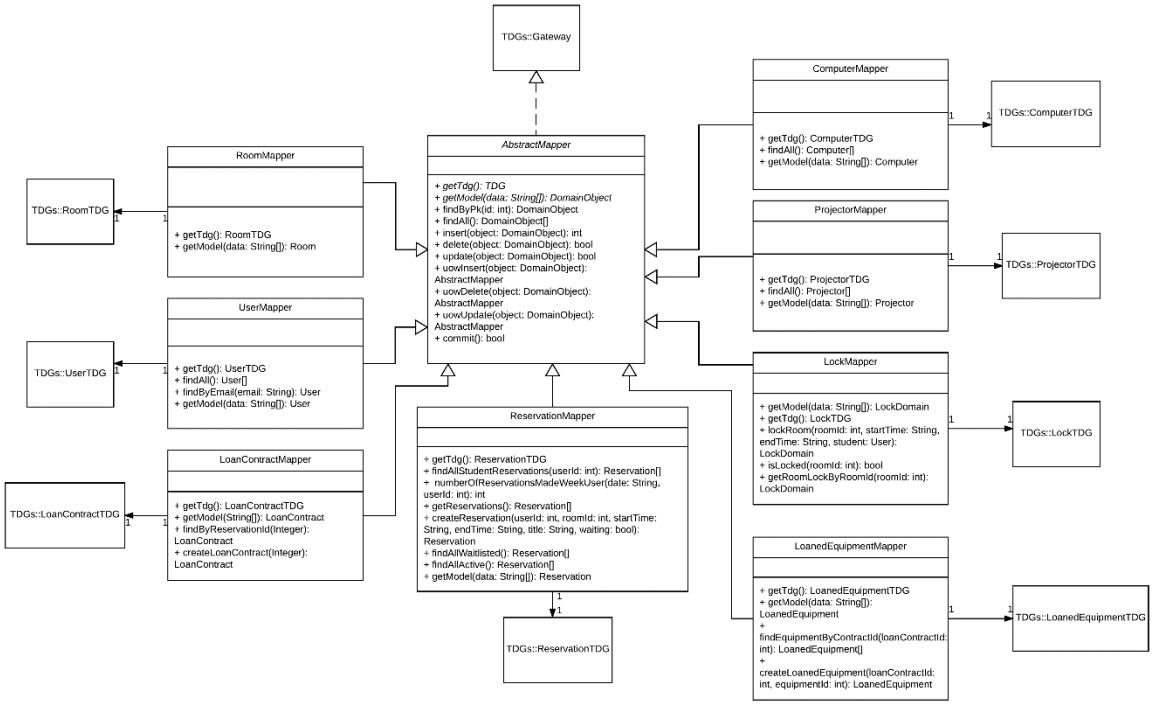

Figure 6: Class Diagram – Data Mappers Package

4.2.5 TABLE DATA GATEWAYS

The Table Data Gateways (TDGs) package lies in the Data Source layer and contains classes that interact with the database, so that the classes of the Application layer do not need to call the database directly.

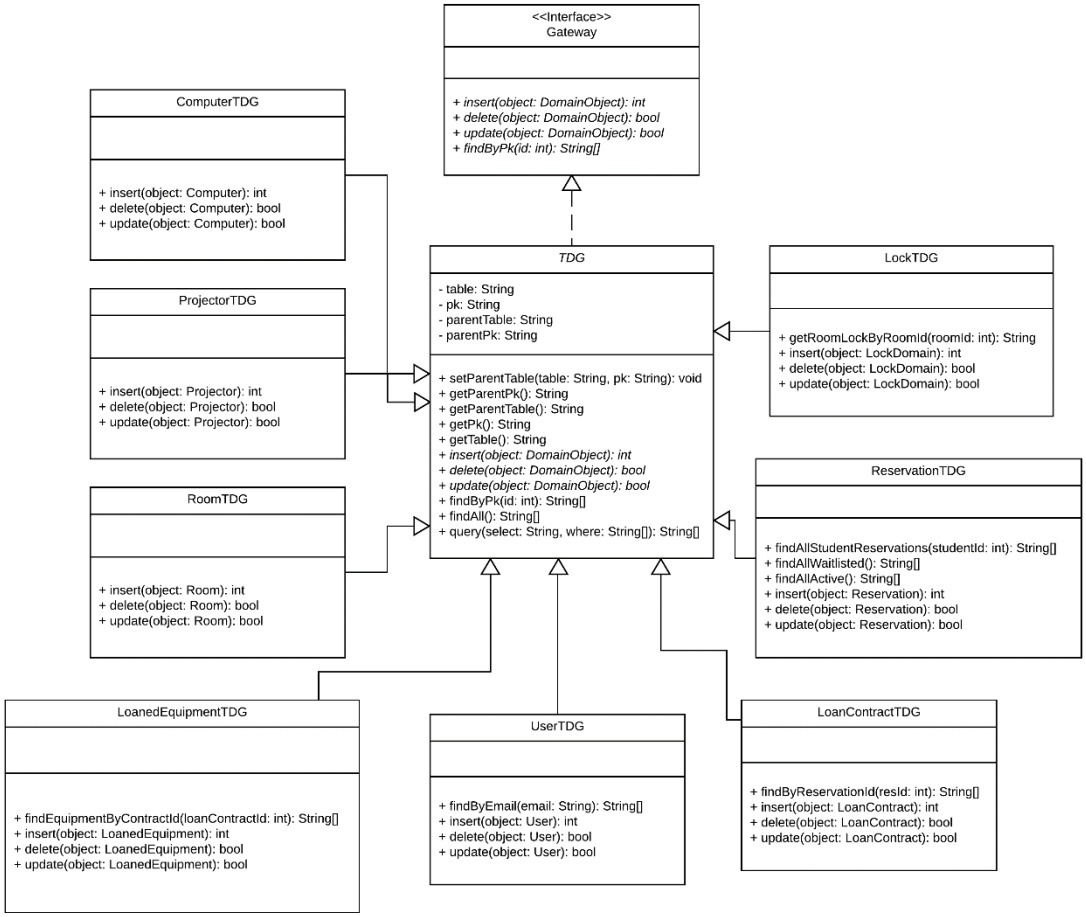

Figure 7: Class Diagram - TDG Package

4.2.6 UNIT OF WORK

The Unit of Work keeps track of which objects have been created, modified or deleted during a given transaction, then updates the database only at the end.

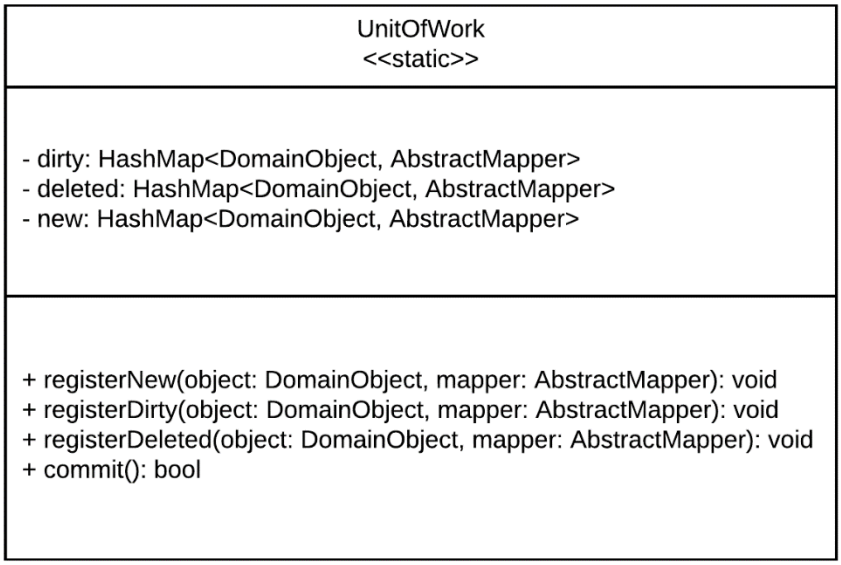

Figure 8: Class Diagram - Unit of Work Package

6 LOGICAL VIEW: INTERACTION DIAGRAMS

5.1 CREATE RESERVATION

The following sequence diagram shows the messages exchanged between classes when a user wishes to create a new reservation.

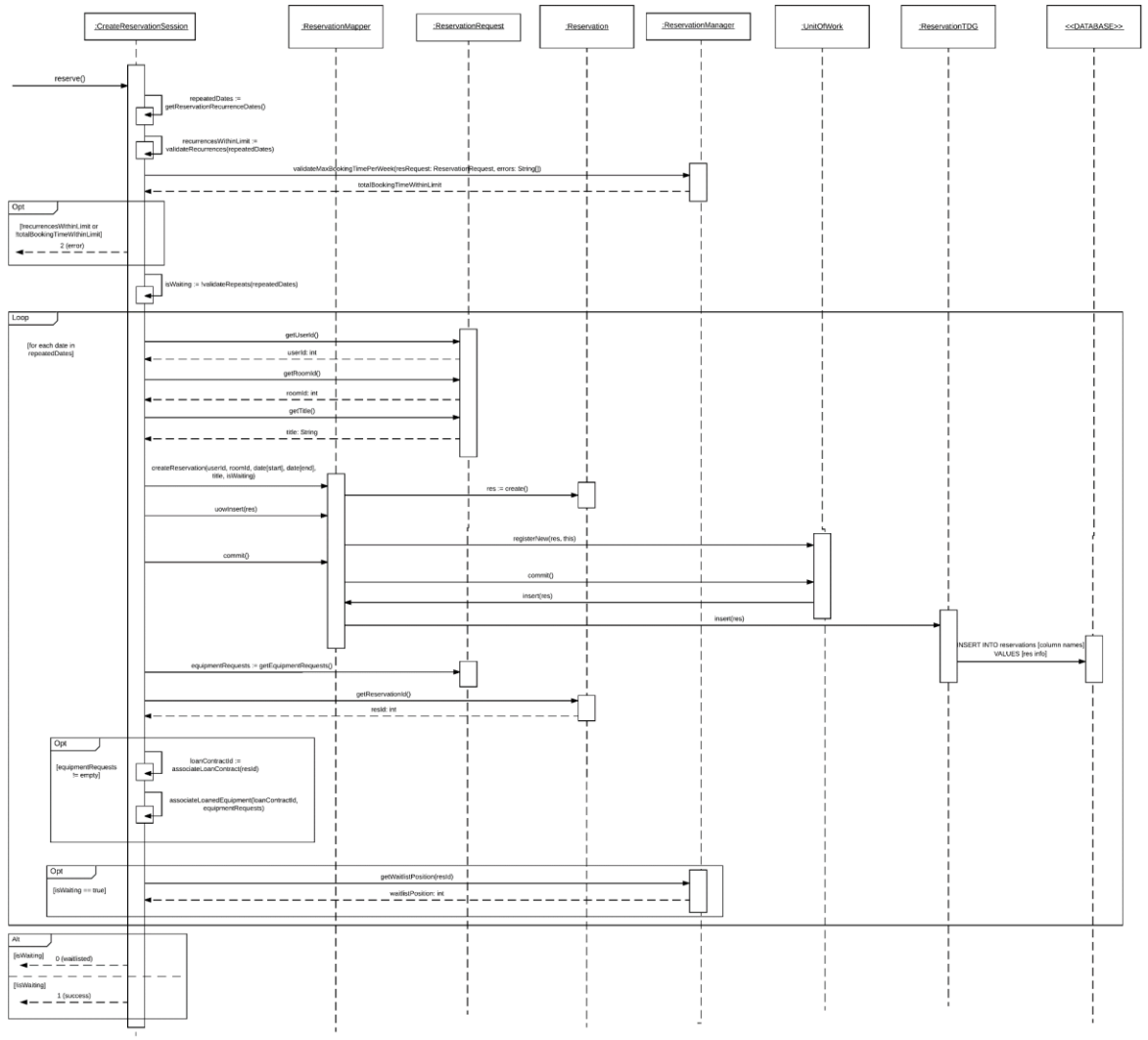

Figure 9: Sequence Diagram - Create Reservation

5.2 MODIFY RESERVATION

The following sequence diagram shows the messages exchanged between classes when a user wishes to modify one of their existing reservations.

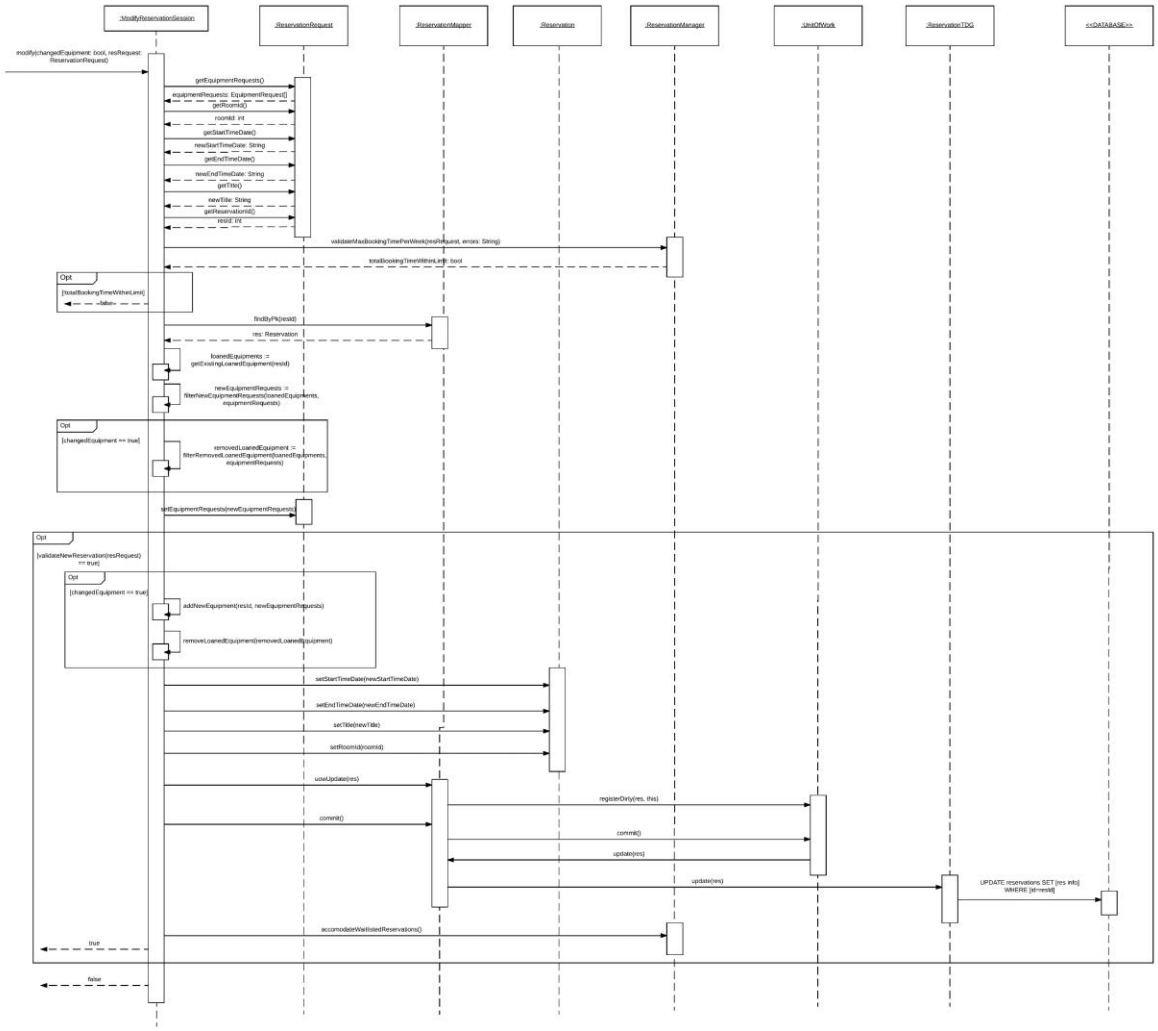

Figure 10: Sequence Diagram - Modify Reservation

5.3 DELETE RESERVATION

The following sequence diagram shows the messages exchanged between classes when a user wishes to cancel one of their existing reservations.

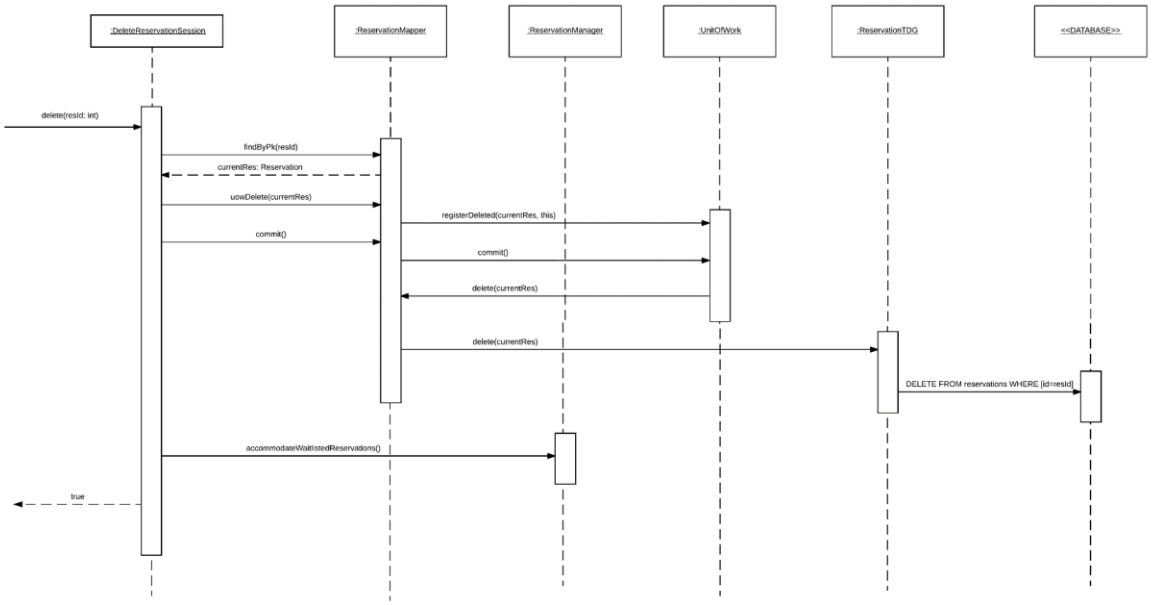

Figure 11: Sequence Diagram - Delete Reservation

5.4 VALIDATE NEW RESERVATION

When creating a new reservation or modifying an existing one, it is important to check for conflicts with other users' reservations as well as verify the availability of requested equipment. Our system performs such checks using the checkForConflicts and the assignAlternateEquipmentId helper methods in the Reservation Manager class. Here, we include an additional sequence diagram showing the calling of these methods and the processing of returned data.

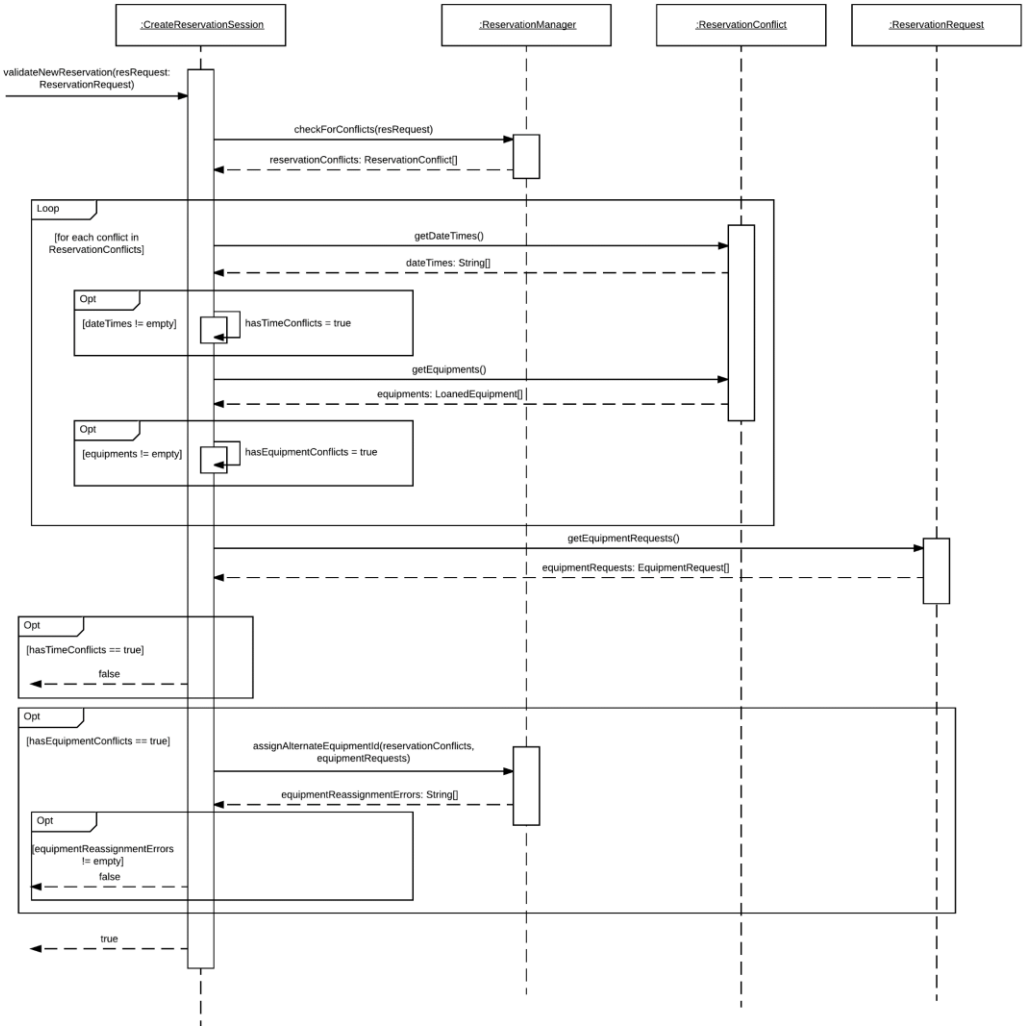

Figure 12: Sequence Diagram - Validate New Reservation

7 DATA VIEW

7.1 ENTITY RELATIONSHIP SCHEMA

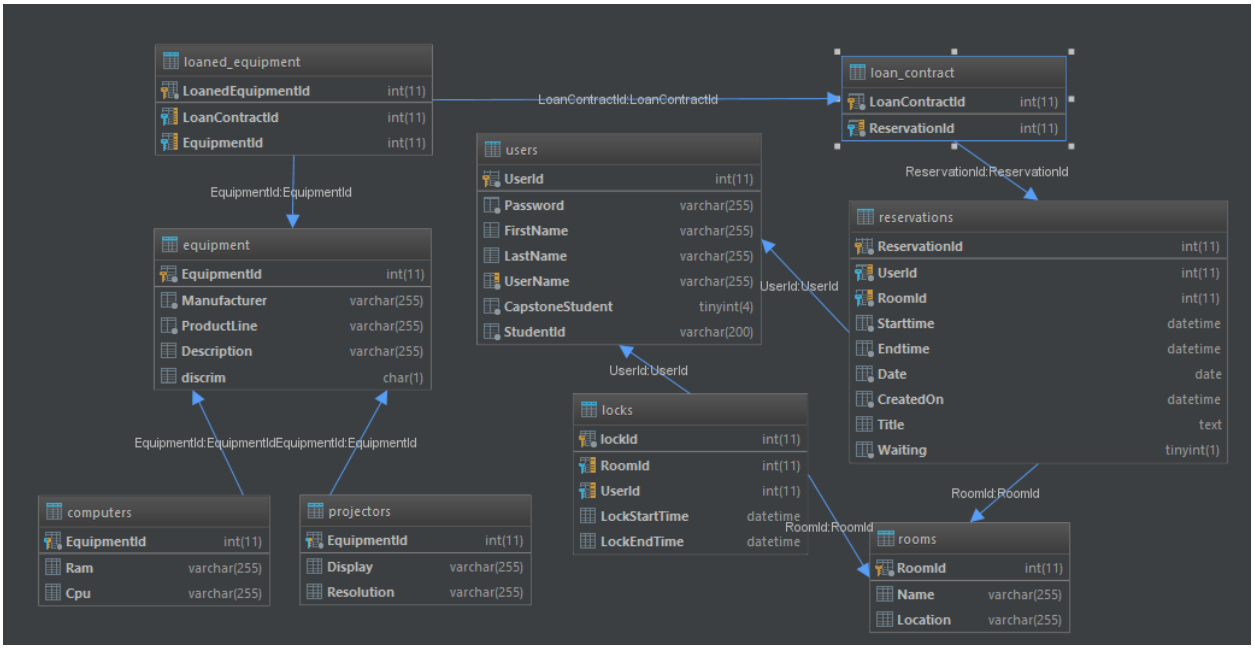

Figure 13: Database ER Schema

7.2 ENTITY RELATIONSHIP DIAGRAM

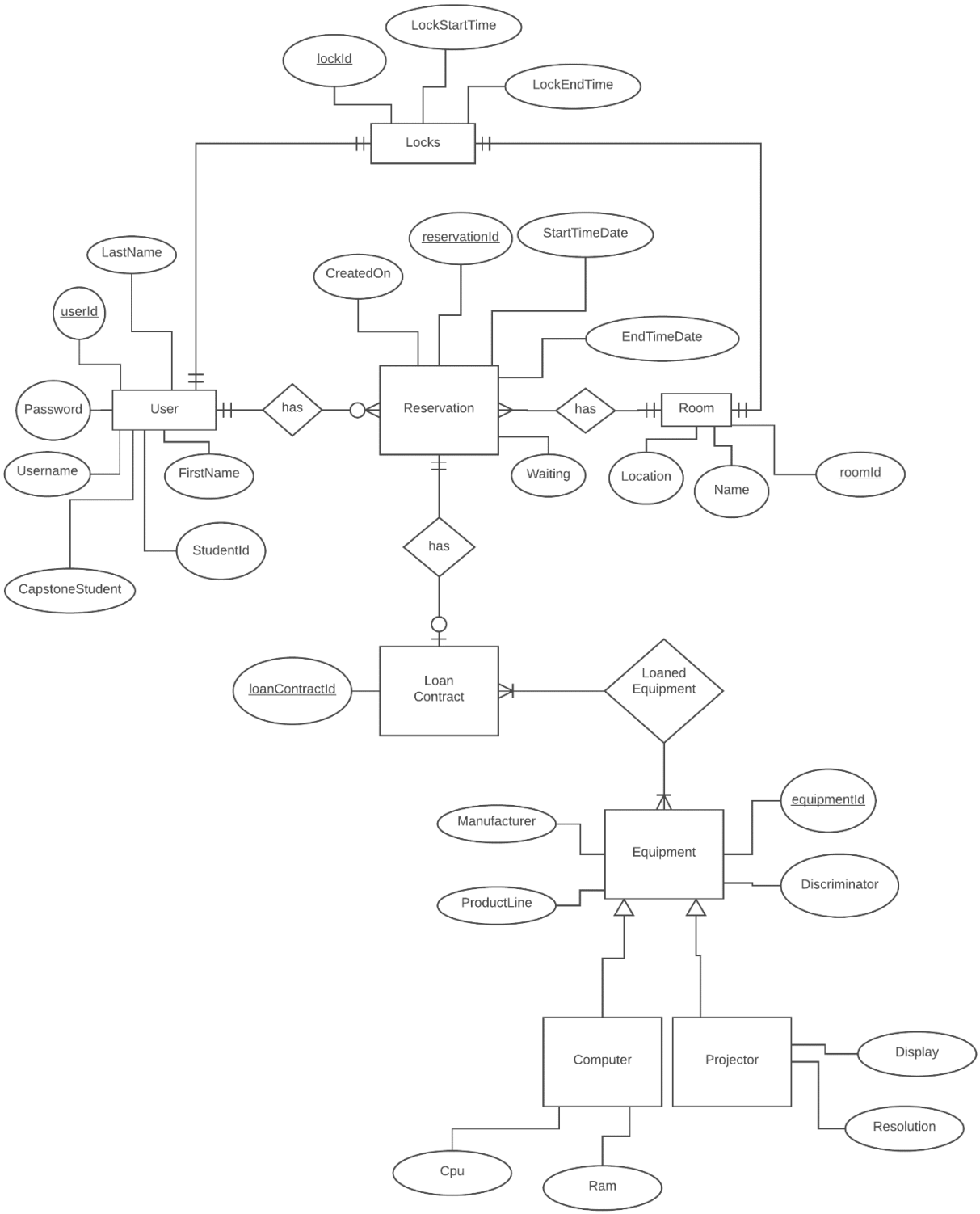

Figure 14: Database ER Diagram

8 PATTERN MIGRATION FROM OOP TO AOP

To incorporate aspect-oriented programming into our application, we used the Go AOP framework for PHP [1]. This framework supports all the major features of AOP, including aspects, advices, joinpoints, pointcuts and introductions.

The architectural pattern we chose to migrate from OOP to AOP was the table data gateway (TDG) pattern. We chose this pattern because, when reviewing the code, we noticed a lot of code duplication among TDG classes. Specifically, the insert(obj), update(obj) and delete(obj) methods of each TDG class contain several statements common to all of them. To eliminate this code duplication, we created an aspect class called TDGAspect. The class contains three around advices which intercept the execution of the insert, update and delete methods, respectively, for every TDG class. They perform logic common to each of these methods, including exception handling and setting the return values. As part of the Go framework requirements, the TDGAspect was registered with the ApplicationAspectKernel class.

The advantage of migrating the TDG pattern from OOP to AOP is that the code of the TDG classes is now cleaner and more readable, due to the moving of duplicated logic into the aspect class. The disadvantage of the migration is the added dependencies of the Go AOP framework. Any user wishing to run the project on their local server must take the time to install the framework and its dependencies.

[1] http://go.aopphp.com/