**Software Architecture Design**

Version 0.03

for

BookMe

Prepared by

|  |  |  |  |
| --- | --- | --- | --- |
| Emir Bozer  Nikolas De Vigne Blanchet  Ahmad Hyjaz Loudin  Mary Psaroudis  Leo Yu | | 26424724  27189877  27179294  27209193  27036736 | emir.bozer@gmail.com  nikdvb@gmail.com  hyjaz.loudin@gmail.com  marypsaroudis@gmail.com  yuleo@outlook.com |
|  |  |  |

|  |  |
| --- | --- |
| Instructor: | Constantinos Constantinides |
| Course: | SOEN343: Software Architecture and Design I |
| Date: | 23/11/2016 |

**Document history**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
|  | 0.01 | All the section added | Everyone |
| 20/11/2016 | 0.02 | Architectural requirements | Mary Psaroudis |
| 20/11/2016 | 0.03 | Formatting | Leo Yu |
|  |  |  |  |

Table of contents

[1. Introduction 4](#_Toc467450037)

[Purpose 4](#_Toc467450038)

[Scope 4](#_Toc467450039)

[Definitions, acronyms, and abbreviations 5](#_Toc467450040)

[2. Architectural representation 6](#_Toc467450041)

[6. Architectural requirements: goals and constrains 8](#_Toc467450042)

[Functional requirements (Use case view) 8](#_Toc467450043)

[Non-functional requirements 9](#_Toc467450044)

[7. Use case view (Scenarios) 10](#_Toc467450045)

[8. Logical view 11](#_Toc467450046)

[Layers, tiers etc. 11](#_Toc467450047)

[Subsystems 12](#_Toc467450048)

[Use case realizations 17](#_Toc467450049)

**List of figures**

Figure 1. The 4+1 View Model……………………………………………………………….…………………………………...6

Figure 2. Use Case Model……………………………………………………………………….………………………………...10

Figure 3. Book Me Layered Class Diagram……………………………………………….………………..………………11

Figure 4. Core Class Diagram…………………………………………………………………….……………………………...13

Figure 5. Mapping Class Diagram……………………………………………………………….……………………………..15

Figure 6. TDG Class Diagram……………………………………………………………………….…………………………….16

Figure 7. Make New Reservation Communication Diagram……………………………………………………….17

Figure 8. View Schedule Communication Diagram……………………………………….……………………………19

Figure 9. Modify Reservation Communication Diagram……………………………….…………………………...20

Figure 10. Cancel Reservation Communication Diagram……………………………………………………….…..21

Figure 11. Add to Waiting List Communication Diagram……………………………………………………………22

Figure 12. Initiate Action Communication Diagram………………………………………………………………....23

Figure 13. End Action Communication Diagram………………………………………………………………………..24

Figure 14. Update Waiting Communication Diagram…………………………………………………………………25

# Introduction

## Purpose

The purpose of the Software Architecture Design (SAD) document is to provide a comprehensive architectural views of the BookMe software, an on-line conference room reservation system. The document is composed of several sections: Architectural Representation, Architectural Requirements, multiple different Views, Size and Performance, and Quality. The Architectural Representation section provides a top-level architectural style of the BookMe system. The Architectural Requirements section describes how key functional and non-functional requirements from the Software Requirement Specification (SRS) affect the architecture. There will be multiple sections to describe of each individual view models provide to the 4+1 architectural view model.  The Size and Performance section details how architecture supports the key sizing and performance requirements. The Quality section evaluates how the software architecture contributes to the quality attributes of the BookMe System based on the ISO 205010. This document is intended for the developers of the BookMe software as it is possible to map the design of the software architecture into lines of codes.

## Scope

The scope of this SAD is to depict the overall architecture of the BookMe software. The BookMe software architecture follows the 4+1 architectural view model, which is composed of five individual views: Logical View, Use Case View, Development View, Process View, physical view. From these views, developers can code the actual software product by mapping the view models into code. Thus, the product’s code is heavily influenced by the SAD. If there is any change in the design of the software architecture, the product’s code must be updated in order to reflect on the new changes.

## Definitions, acronyms, and abbreviations

**RUP** Rational Unified Process

**UML** Unified Modeling Language

**SAD** Software Architecture Document

**SRS** Software Requirements Specification

# Architectural representation

The overall top-level architectural style of the BookMe system is model using the 4+1 architectural view model illustrated in Figure X. Each views describe the system in the point of the view of the different stakeholders: Designers, programmers, deployment manager and integrators. The four views are the logical view, the deployment view, the process view and the physical. Also, critical use cases shall be used to demonstrate the system architecture, thus serving as the use case (scenario) view.

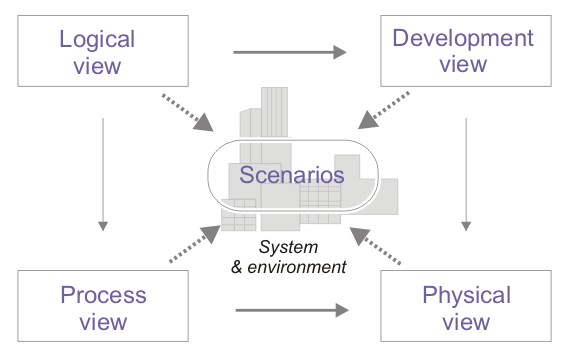


Figure 1. The 4+1 View Model

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. **Development view** (also known as Implementaion view): Audience: Programmers. The development view illustrates a system from a programmer's perspective and is concerned with software management. This view is also known as the implementation view. It uses the UML Component diagram to describe system components. UML Diagrams used to represent the development view include the **Package diagram**.
3. **Process view**: Audience: Integrators. The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system. The process view addresses concurrency, distribution, integrators, performance, and scalability, etc. UML Diagrams to represent process view include the **Activity diagram**.
4. **Physical view** (also known as deployment view): Audience: Deployment managers. The physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer, as well as the physical connections between these components. UML Diagrams used to represent physical view include the **Deployment diagram**.
5. **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 which become a fifth view. 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 Artefacts: **Use-Case Model**.

Note: This document will only cover the Logical View and the Use case view.

# Architectural requirements: goals and constrains

There are some key requirements and constraints that have a significant impact on the architecture of the system.

* The BookMe system must access the PostgreSQL database to retrieve/add/modify user and reservation information.
* The system must be secure. Authentication and authorization of users are required to be able to perform system operations.
* Data persistence will be applied using the object-relational database system PostgreSQL.

## Functional requirements (Use case view)

The significant use cases of the BookMe system are:

* Reserve a room
* Cancel an existing reservation
* Modify a reservation
* View my reservations
* View schedule

|  |  |  |  |
| --- | --- | --- | --- |
| **Source** | **Name** | **Architectural relevance** | **Addressed in:** |
| U3, UC4 | Reserve Room, Add to Waiting List | Allows the user to reserve a room at a given date and time. Should the room selected not be available, the user will have the option to be added to a waitlist. | Section 9 - Logical view: Use case realizations |

|  |  |  |  |
| --- | --- | --- | --- |
| **Source** | **Name** | **Architectural relevance** | **Addressed in:** |
| U5, UC6, UC7 | View my reservations, Modify Reservation, Cancel Reservation | Allows the user to view their reservations. The user can also modify or cancel an existing reservation. | Section 9 - Logical view: Use case realizations |

|  |  |  |  |
| --- | --- | --- | --- |
| **Source** | **Name** | **Architectural relevance** | **Addressed in:** |
| U8 | View schedule | Allows the user to view a calendar of reservations with the occupied time slots of rooms. | Section 9 - Logical view: Use case realizations |

## Non-functional requirements

|  |  |  |  |
| --- | --- | --- | --- |
| **Source** | **Name** | **Architectural relevance** | **Addressed in:** |
| SRS | Design Constrains | Implementation of an object-relational structural and behavioral patterns for the database. | Section 9 - Logical view: Layer, tiers etc. |

# Use case view (Scenarios)

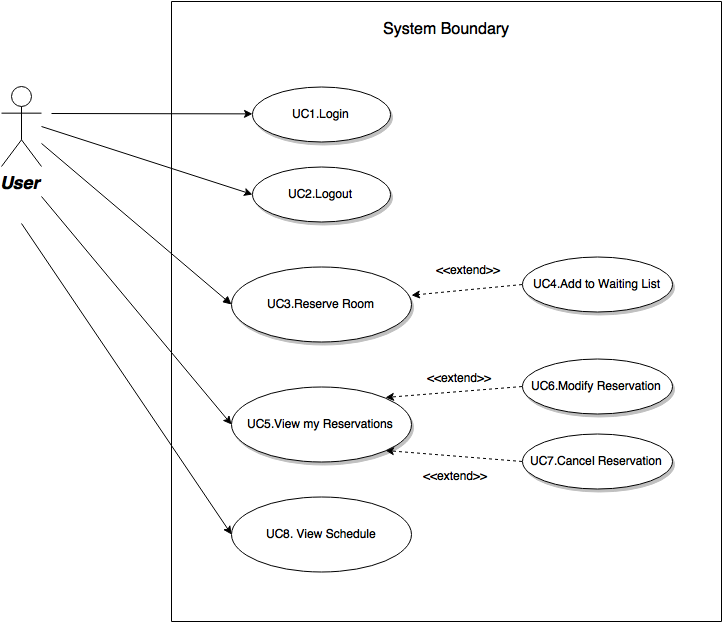


Figure 2. Use Case Model

# Logical view

## Layers, tiers etc.

In terms of the logical view, the top-level architecture style of the BookMe software follows the three-layered system of the Enterprise Application pattern. It is composed of a presentation layer, a domain layer and a data source layer. The Presentation layer contains the graphical user interface (GUI) of the BookMe application. The Domain layer contains the logic that governs the software. The Data Source layer contains the persistent storage mechanism of the BookMe system.

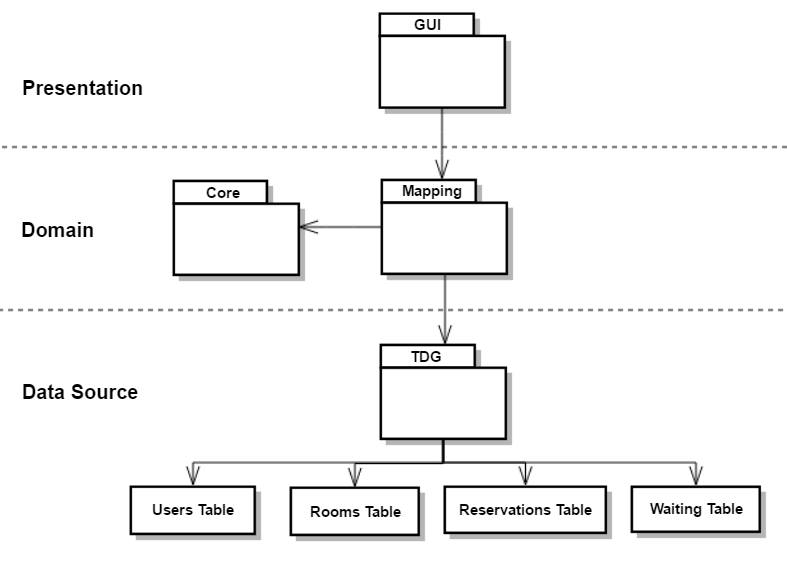


Figure 3. Book Me Layered Class Diagram

## Subsystems

The decomposition of the BookMe system is based on the three different layers and the specific functions that the subsystem provides. All GUIs that the user utilizes to interact with the BookMe software will be inside the GUI package. The Mapping system is a package composed of all classes related to three architectural patterns: Data Mapper, Identity Map and Unit of Work. Core subsystem is a package that contains all logics and classes needed to execute the application. The Table Data Gateway (TDG) subsystem is a package that contains all the TDG of the system. The relation between these subsystems are described below.

To start with, the GUI package sends inputs to the system by communicating with the Mapping package to execute a process. The Mapping package is set up in between the Core package and TDG package in order to move data between them while reducing the amount of coupling and database call. The TDG packages have access to the database tables for when the Mappers packages sends messages to interact with the databases.

**Architecturally significant design packages**

Notes: In order to improve readability and comprehension of the class diagrams, methods that are not pertinent to the communication diagrams or architecture of the system may not appear in the class diagrams. This includes accessors, mutators, auxiliary and generic methods.

**Core Subsystem**

The Core subsystem is the main subsystem of the BookMe web application. The package contains all the methods and the classes needed to satisfy the requirements and the constraints of the software project. Despite creating, modifying and deleting instances of reservation objects, the Core subsystem contains no code related to the interaction of the database

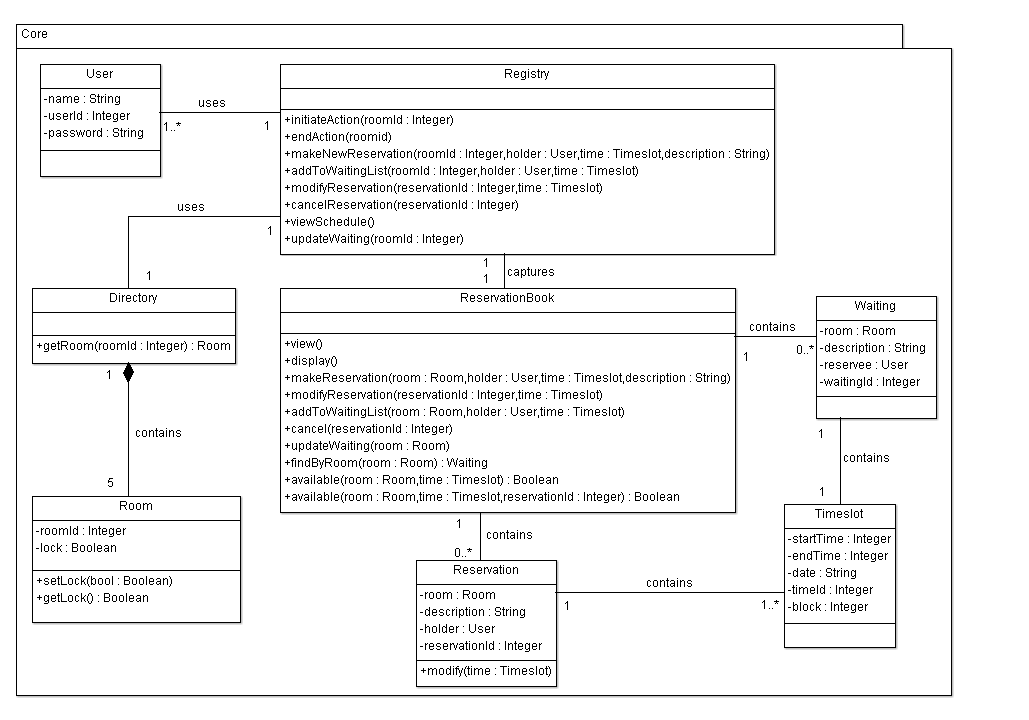


Figure 4. Core Class Diagram

**Mapping Subsystem**

        The Mapping package contains three architectural patterns: Data Mapper, Identity Map and Unit of Work. The Data Mapper pattern is represented as Mapper objects that are in charge of setting up a communication between the Core package and the TDG package in order to move data between them while keeping them independent from each other. The Identity Map pattern is represented as IdentityMap objects that prevent loading the same data from the database twice. The Unit of Work pattern is represented as a single class instance that is in charge of keeping track of every changes during a process and committing those changes at the end of a process.

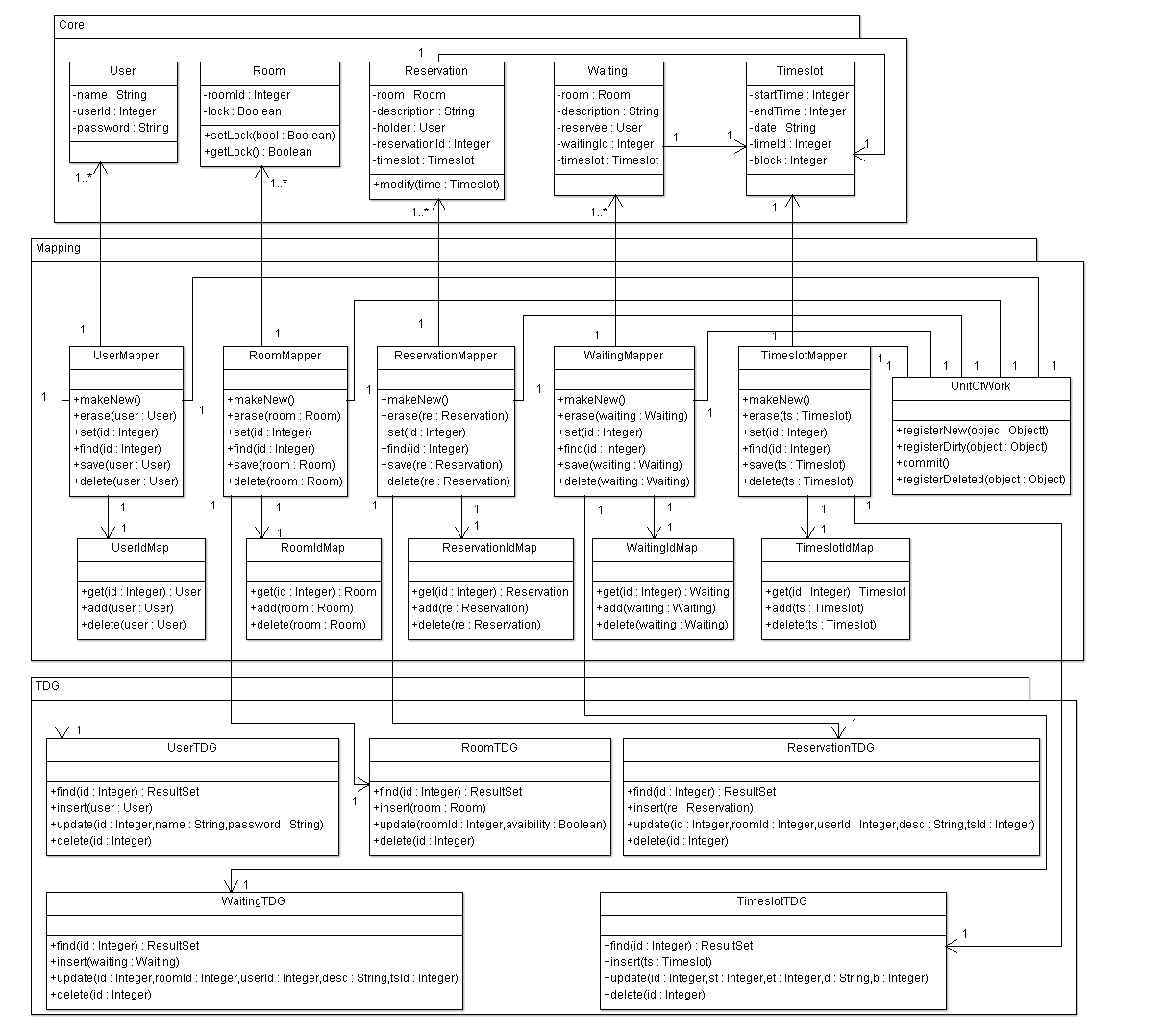


Figure 5. Mapping Class Diagram

**TDG Subsystem**

        The TDG subsystem follows the Table Data Gateway (TDG) pattern. The package contains TDG objects that acts as an intermediary between the data mapper and the database. Each data mapper objects have a corresponding TDG objects that will provide an interface for accessing the database. Hence, its role is to push data back and forth between the data mapper and the data base.

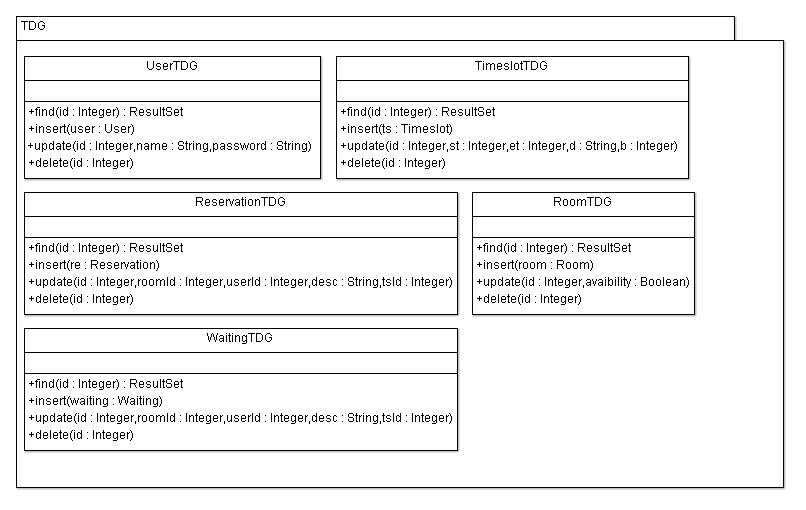


Figure 6. TDG Class Diagram

## Use case realizations

|  |
| --- |
| **Contract CO1**: Make New Reservation  Operation: makeNewReservation(roomId, holder, time, description)  Cross References: Use Case Reserve Room  Preconditons:   * A ReservationBook session is underway   Postconditions:   * Reservation instance r was created * Attributes of r were initialized * r is added to ReservationBook |

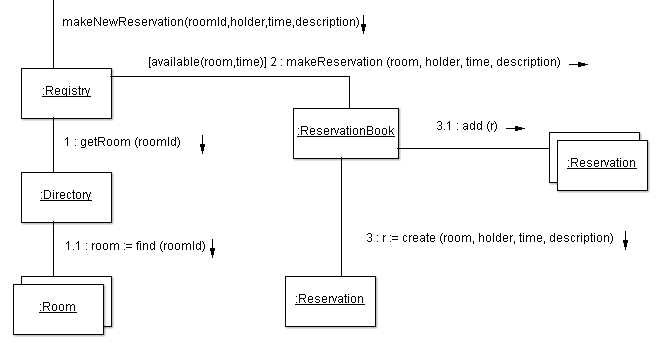
****

Figure 7. Make New Reservation Communication Diagram

This communication diagram represents the make new reservation operation and operation contract, which was built from the specified system operations and the system sequence diagram (ssd) pertaining to the reserve room use case. For instance, the ssd for the make new reservation use case specified that the system would need a make new reservation operation, and therefore it was added to the system operations and then further detailed in its operation contract by specifying its pre and post conditions. The above diagram satisfies these conditions since it shows how the operation creates a Reservation instance and initializes it, and also shows that it is added to the reservationBook. The process of creating sequence diagrams from use cases, then listing the identified system operations, and then creating operation contracts for each of these operations was the process used to produce each of the following communication diagrams.

When the user calls the makeNewReservation method, it is passed to an instance of the Registry class, which passes a message to the directory to find the specified room in its Room collection. If the room is available at the specified time, a message from Registry is passed to ReservationBook, and from there a message is passed to Reservation to create an object of its type, which is captured and then added by ReservationBook to its own Reservation collection.

When the find(room id) message is passed from the Directory object,  the find method of the room mapper is called, and then the RoomIdMap is checked to see if the map exists in memory, if it does not then the mapper asks the table data gateway (tdg) to retrieve it from the database, it then adds it to the identity map to show that it has been created in memory. If it already exists in memory then it is simply retrieved. The same is done for the Timeslot objects contained in the room. When the create message is passed to the Reservation class the makeNew() method of the mapper is called , which registers the new object into the unit of work, adds it to the identity map, and whenever the unit of work is committed, then the objects gets saved into the database by passing a message from the mapper to the tdg.

|  |
| --- |
| **Contract CO2**: View Schedule  Operation: viewSchedule()  Cross References: Use Case View Schedule  Preconditons:   * User is logged in. * A ReservationBook session is underway.   Postconditions:   * A schedule is created from information contained in the ReservationBook |

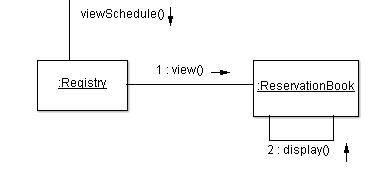


Figure 8. View Schedule Communication Diagram

If a user calls the viewSchedule() it is passed to an instance of the Registry class. Then the message from the Registry is passed to ReservationBook. From there, ReservationBook calls the display() method from itself and displays the schedule.

|  |
| --- |
| **Contract CO3**: Modify Reservation  Operation: modifyReservation(reservationId, time)  Cross References: Use Case Modify Reservation  Preconditons:   * User is viewing his/her reservations * The user’s myReservations is non-empty   Postconditions:   * Reservation entry attributes updated |

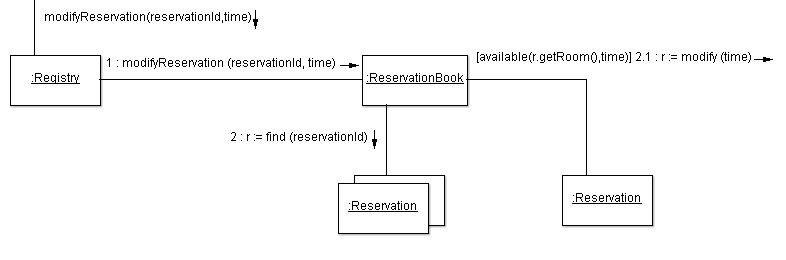


Figure 9. Modify Reservation Communication Diagram

This communication diagram represents the modify reservation operation. When the user modifies the time for one of his reservations a message is sent to the registry with the reservation’s id and the modified time. The registry then passes this message to the reservation book, which calls the find method on its reservation collection and captures the specified reservation. The reservation book then passes a message to the captured reservation to get the room and then checks if that room is available at the time the user specified, if it is then the reservation’s time is modified. When the reservation book tries to find the specified reservation, the reservation mapper checks if it exists in memory with the help of the reservation identity map, if it does not then the mapper asks the tdg to retrieve it from the database. When the reservation’s time is modified, its mapper registers it as “dirty” in the unit of work since its state has been changed.

|  |
| --- |
| **Contract CO4**: Cancel Reservation  Operation: cancelReservation(reservationId)  Cross References: Use Case Modify Reservation  Preconditons:   * User is viewing his/her reservations * The user’s myReservations is non-empty   Postconditions:   * Reservation entry removed from ReservationBook |

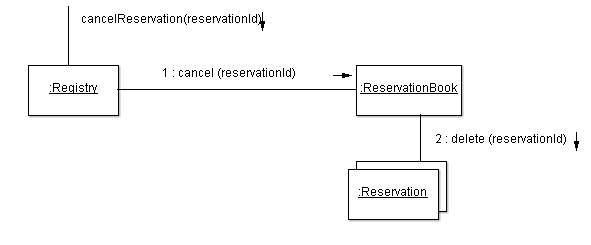


Figure 10. Cancel Reservation Communication Diagram

If a user wants to cancel a reservation, cancelReservation() method is called, after it is called it is passed to an instance of the Registry class. After that, a message from Registry is passed to ReservationBook, and from there a message is passed to Reservation collection in with the reservationID in order to delete the specified reservation from the collection.

|  |
| --- |
| **Contract CO5**: Add to Waiting List  Operation:  addToWaitingList(roomId, holder, time,description)  Cross References: Use Case Add to Waiting List  Preconditons:   * The user has initiated a Reservation Session * The user has tried to make a new reservation, but the room is unavailable   Postconditions:   * A WaitingList instance w was created * Attributes of w were initialized * Instance w was added to user’s myWaitingList |

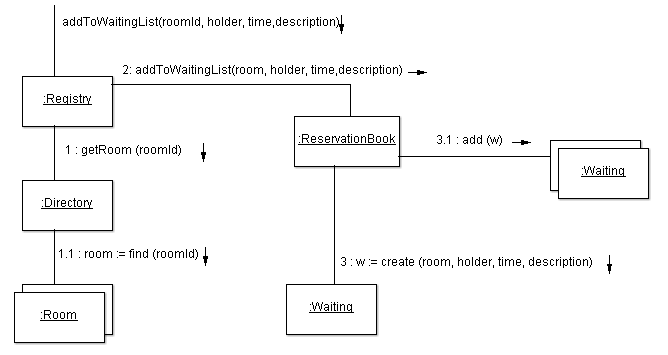


Figure 11. Add to Waiting List Communication Diagram

If a user tries to makeNewRezervation() and the time slots are taken addToWaitingList() method is called. After it is called, it is passed to an instance of the Registry class, which passes a message to the directory to find the specified room in its Room collection. Because the room is not available at that time, a message from Registry is passed to ReservationBook, and from there a message is passed to Waiting to create an object of its type, which is captured and then added by ReservationBook to its own Waiting collection.

When the find(room id) message is passed from the Directory object,  the find method of the room mapper is called, and then the RoomIdMap is checked to see if the map exists in memory, if it does not then the mapper asks the table data gateway (tdg) to retrieve it from the database, it then adds it to the identity map to show that it has been created in memory. If it already exists in memory then it is simply retrieved.

|  |
| --- |
| **Contract CO6**: Initiate Action  Operation:  initiateAction(roomId)  Cross References: Use Case Reserve Room, Add to Waiting List, Modify Reservation,  Cancel Reservation  Preconditons:   * The user has initiated a Reservation Session   Postconditions:   * The room’s lock is set to true |

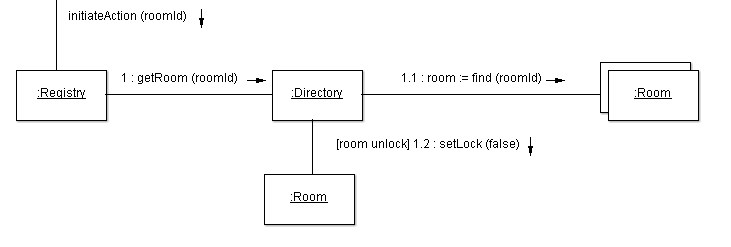


Figure 12. Initiate Action Communication Diagram

This communication diagram represents the initiate action operation. When the user initiates an action such as reserving a room, adding him/herself to a waiting list, and modifying or canceling a reservation, an initiate action message is passed to the registry along with the id of the room to which the operation is related. Once the registry gets this message, it sends a get room message to the directory, which then sends a find message to its collection of rooms, once it finds the room, it “locks” the room’s lock. When the find room message is passed, the room mapper checks if the room exists in memory with the identity map, if it does it retrieves it, if it does not the mapper asks the tdg to retrieve it from the database. When the room’s lock is “locked” the room’s mapper registers the object as “dirty” in the unit of work since its state has been changed.

|  |
| --- |
| **Contract CO7**: End Action  Operation:  endAction(roomId)  Cross References: Use Case Reserve Room, Add to Waiting List, Modify Reservation,                 Cancel Reservation  Preconditons:   * The initiateAction(roomId) operation has been called   Postconditions:   * The room’s lock is set to false |

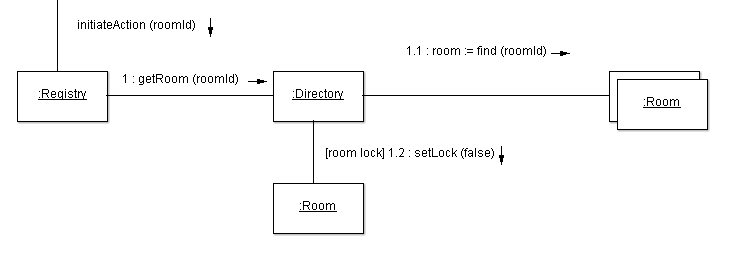


Figure 13. End Action Communication Diagram

This communication diagram represents the end action operation. When the user ends an action such as reserving a room, adding him/herself to a waiting list, and modifying or canceling a reservation, an end action message is passed to the registry along with the id of the room to which the operation was related. Once the registry gets this message, it sends a get room message to the directory, which in turn passes a find room message to its collection of rooms. Once the room is found an unlock message is passed to it and its lock is “unlocked”. Similarly to the initiate action operation diagram, when the find room message is passed, the room mapper checks if the room exists in memory with the identity map, if it does, it retrieves it, if it does not the mapper asks the tdg to retrieve it from the database. When the room’s lock is “unlocked” the room’s mapper registers the object as “dirty” in the unit of work since its state has been changed.

|  |
| --- |
| **Contract CO8**: Update Waiting  Operation: updateWaiting(roomId)  Cross References: Modify Reservation, Cancel Reservation  Preconditons:   * The modifyReservation(reservationId) or cancelReservation(reservationId) operation has been called   Postconditions:   * If room is available at Waiting w’s timeslot, an instance of Reservation r is created * Reservation r’s attributes are initialized with the w’s attributes * Reservation r is added to reservationBook and w is removed from reservationBook |

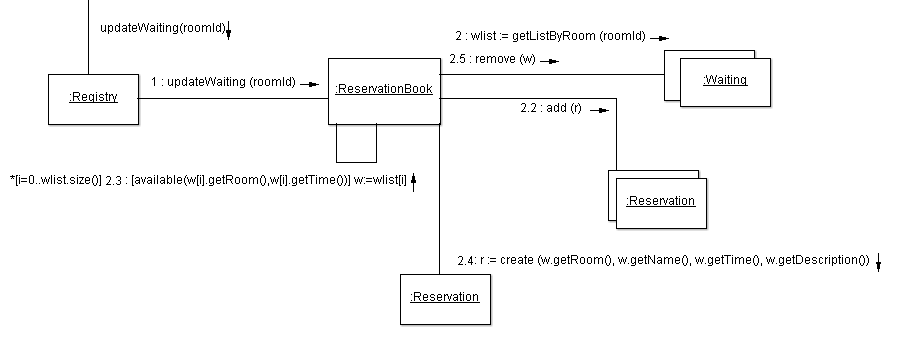


Figure 14. Update Waiting Communication Diagram

This communication diagram represents the update waiting list operation. Whenever the user modifies or cancels a reservation an update waiting list message is sent to the registry. The registry then passes an update waiting list message to the reservation book with the modified or canceled room’s id and time. The reservation book then passes a message to its queue of users waiting for a room, which finds if an instance in the waiting list can be made into a reservation because of the previous reservation modification or cancellation. If such an instance in the waiting list exists it is removed and then the reservation book creates a reservation with the information provided by it and stores it in its reservation collection. The process of finding waiting list instances to be made into reservations is done until none of them can be converted. Furthermore, when an instance in the waiting list is being retrieved, a message is sent to the waiting list mapper, which then asks its identity map if the object already exists in memory, if it does it is retrieved, if not the mapper asks the tdg to retrieve it from the database. When a waiting list instance is removed, its mapper registers it as deleted in the unit of work.