Hotel Management System

A report describing different models used to model a hotel management system

Henry Yang
Erik Karlkvist
Jesper Jaxing
Oskar Willman
Zoe Sanderson-Wall
Danielle Santos
Alexander Håkansson

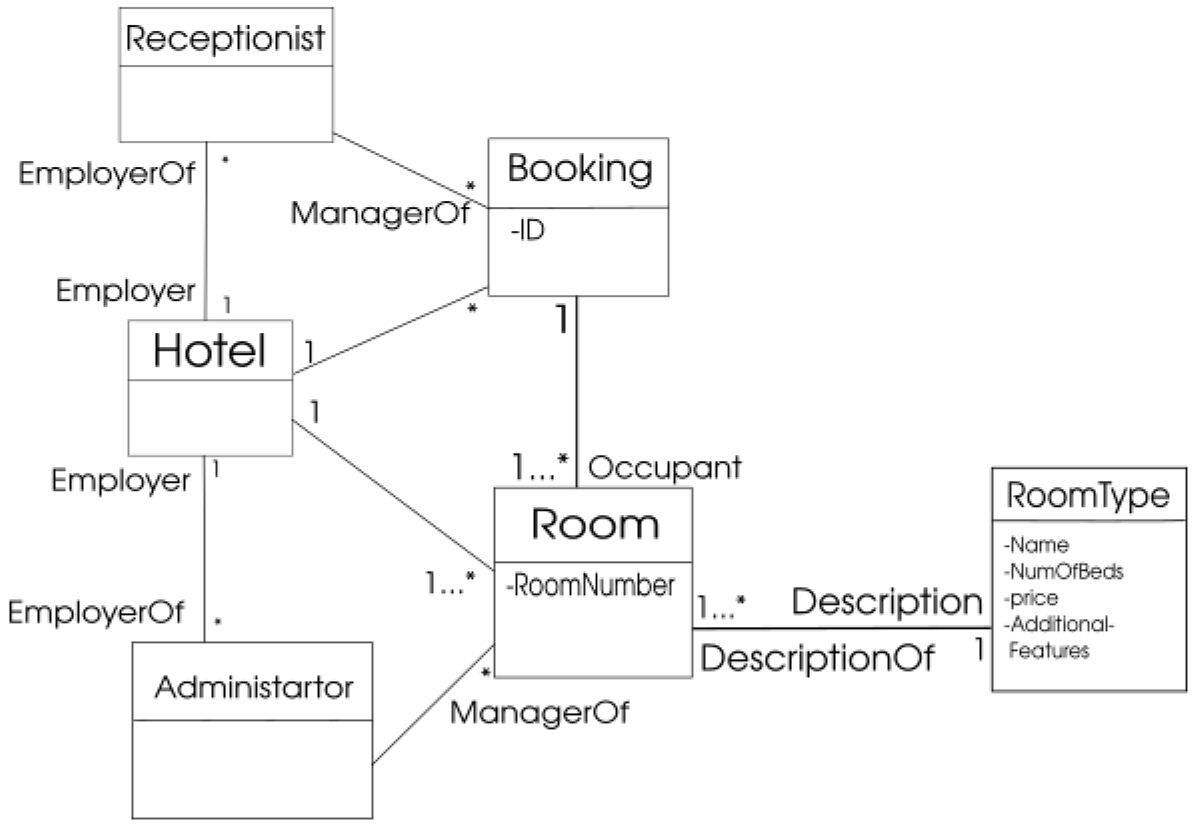
Chalmers University of Technology
University of Gothenburg
TDA593/DIT945
2016-12-23
Group 18

Introduction

This project focuses on different models for a hotel management system including the domain model, use case diagram, component diagram, state machine diagram, sequence diagram and class diagram. The goal of the project is to use the different models to visualise and understand how the hotel functions. The end goal was to model and create a simple hotel that can make a booking for one or more rooms, check these bookings in and out and provide basic management tools (e.g blocking rooms or adding new rooms). We were given a handful of use cases that had to be implemented. The last part of the project was code generation and implementations of test to check if the models represented a functional hotel.

Table of contents

Domain Model

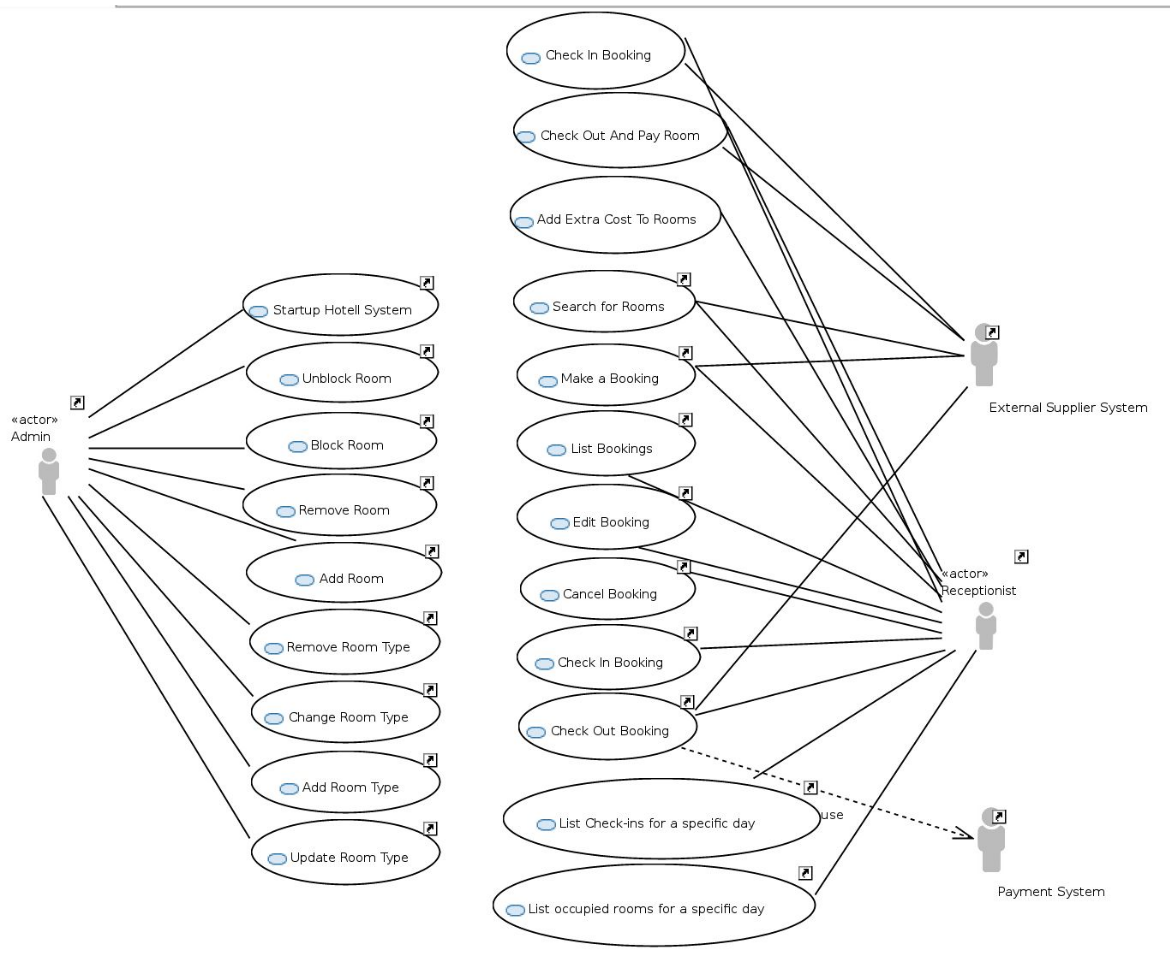


The system, known as “Hotel”, has any number of receptionists and administrators who handle bookings and rooms respectively. The receptionists and administrators are actors in the system. These actors do not explicitly have their own bookings or rooms, they simply manage them. A booking is a logical entity that describes how many rooms of specific types are requested by a single guest at a specified time period. The rooms represent physical rooms in the hotel that can be of different types.

The rooms and bookings are owned by the system. There can be any number of bookings, since there is no restriction on how far into the future a booking can be made. There is a finite number of physical rooms.

A booking has a booking ID which can be used to gain more information about the booking, such as the first and last name of the guest, as well as the start and end date of the booking. Each room has a number and can either be available or blocked depending on whether it can host guests or not. Each room in turn has a room type, which holds information about the room details, such as price, number of beds and additional features. Multiple rooms can share the same room type.

Use Case Diagram



The model includes four actors, two internal and two external. The Administrator and the Receptionist are internal actors that are implemented in this system. The Payment System and the External Supplier System are external actors that are provided and defined in the requirements specification. Each of these actors are involved in different use cases which are also defined in the requirements specification.

As the above use case diagram shows the use cases for the Administrator and the Receptionist are completely decoupled from each other. This is a good indication that the system can be split into smaller subsystems, which proved true in the final implementation. The Payment System is used as a black box for handling payments as guests check out, and the payment system can not access the hotel system itself. The external supplier system is a limited alternative to the receptionist for making bookings.

Testing

To ensure that all of the use cases are working as expected with respect to the requirements specification some test suites are implemented in a separate package. There are two test suites, one for the receptionist and one for the administrator. The test suite for the receptionist tests that all of the specified use cases for the receptionist can be performed and are working as expected and the suite for the administrator does the same but for the use cases for the administrator.

In total there are 31 tests and every test is implemented as a JUnit test. At the beginning of every individual test the system is re-initialized to make sure none of the tests interfere with each other. That all of the tests pass implies that the implementation supports the use cases as expected.

There are also some provided black box tests that also test that the use cases are implemented correctly. In total there are 20 of these tests and all of them pass as well.

Component Diagram

Discussion

The hotel system is essentially two decoupled systems, a booking system and a room management system. We chose a decoupled design for the system so that the separate components can be reused in other projects or systems. Other benefits also include extensibility and modifiability.

As shown in the component diagram the two subsystems are only connected through one simple interface, which allows the booking system to retrieve the physical rooms that are available to host guests. The booking system then monitors which of these rooms are actually booked and occupied by guests. The room management system handles the physical rooms and monitors which rooms exist in the physical hotel, which rooms are unable to host guests (Blocked), and which room types exist. The room management system is only accessible by an administrator and the booking system can only be used by a receptionist or the “External Supplier System”.

External Supplier System, IAdministratorProvides, ICustomerProvides, BankComponent and IHotelCustomerProvides were all provided and further explained in the requirements specification.

Components

Hotel

The Hotel component is a white-box view of the hotel system. It reflects the scope of the system which encompasses five interfaces and four components. Two ports provide connection points to a number of external services.

Receptionist

A component representing the actor Receptionist. This component uses the IHotelBookingManager interface to interact with the booking component. The actor is represented in the diagram as a component because it is important to show how the receptionist interacts with the rest of the system.

RoomManagement

The RoomManagement is a component that provides interfaces for creating and managing rooms. These interfaces are used for communication with the Administrator component and the

BookingSystem component. The purpose of this component is to keep track of all existing rooms and their types. It also keep tracks of which rooms are blocked and not.

BookingSystem

One of the most important component of the hotel management system, the BookingSystem component provides interfaces for managing the bookings and other corresponding actions. It uses the RoomManagement component for retrieving information about rooms related to a booking. For example, searching for free rooms when creating a booking or changing room types when editing a booking. The purpose of this component is to monitor and manage bookings and which rooms are booked.

Administrator

The Administrator is a component that represents the Administrator actor. It uses the interfaces *IHotelRoomManager* and *IHotelStartupProvides* to interact with the hotel system. The interfaces are used to manage the available rooms in the hotel as well as booting up the hotel system respectively. The purpose of this component in the component diagram is to show how the Administrator actor will interact with the hotel system.

Interfaces

IHotelBookingManager

The IHotelBookingManager interface is used by the Receptionist component and generalises the HotelCustomerProvides interface. The purpose of this interface is to provide the receptionist with the means to perform all tasks that are accommodated for by the HotelCustomerProvides interface, such as confirming or initiating a booking, whilst supporting the additional tasks required of the receptionist that are not included in this interface. Such tasks include editing or cancelling a booking, listing all current bookings, and initiating check in. Essentially the HotelBookingManager interface provides a connection to the BookingSystem component.

- initiateCheckIn: Initiates the check in process, taking a booking ID number as input argument and returns the list of rooms (IRoom) that are linked to that booking id.
    - UC 2.1.3
- editBookingPeriod: Allows for the editing start and end dates of a current booking. The method requires three input arguments: bookingId, startDate and endDate.
    - UC 2.1.5
- cancelBooking: Cancels a booking. Takes the booking ID as input argument.
    - UC 2.1.6
- listBooking: Provides a list of all current bookings. Returns a list of IBookings.
    - UC 2.1.7

- listOccupiedRooms: Provides a list of occupied rooms for a specific date. Takes a date as input argument and returns a list of rooms..
    - UC 2.1.8
- listCheckIns: Provides a list of check ins for a specific period of time. Takes a start and end date as input arguments and returns a list of the check ins (IEvent) that exist between these dates.
    - UC 2.1.9
- listCheckOuts: Provides a list of check outs for a specific period of time. A start and end date are taken as input arguments and a list of check outs that exist between these dates is returned.
    - UC 2.1.10
- addExtraCostToRoom: Adds an extra cost to a specified room. The user provides the booking id, room number, description of the cost and the price of the cost.
  UC 2.1.13
- editBookingRooms: Edit the room type and the number of rooms of a specified booking id. It takes the booking id, number of rooms, room type (IRoom Type) as input.
    - UC 2.1.5

IHotelRoomProvider

IHotelRoomProvider is an interface towards the RoomManagement component. This interface is an abstraction between the booking part of the system and the physical room management by providing the booking component with the available rooms (existing unblocked rooms).

- getRooms: It is an important part in our system because it is responsible for returning all rooms current available to the booking system. On the booking manager side, it is used for getting the list of all rooms which are associated to the booking

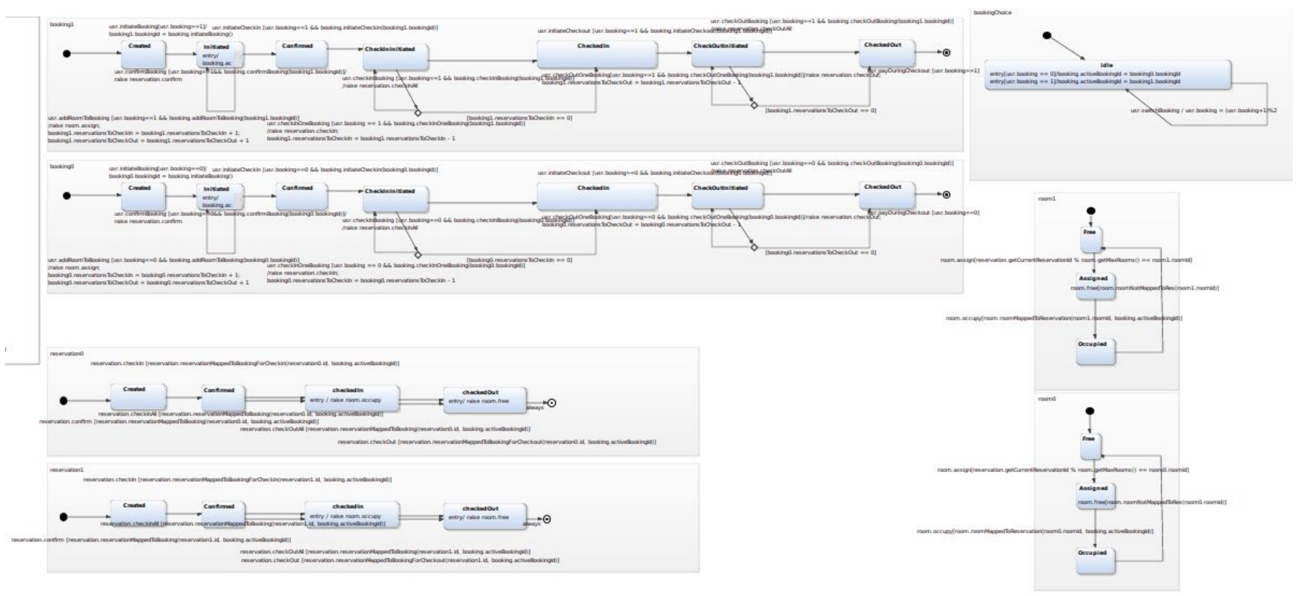
IHotelRoomManager

The HotelRoomManager interface provides all functionalities related to room management, where only the admin will be able to add or remove a room, change a room type, block or unblock a room. To do so, this interface inherits “getRooms” from IHotelRoomProvider being able to access the list of all rooms current in the hotel system.

- addRoomType: It adds a given room type to the system. This method initially requires four parameters (the room name, the price of the room, the number of beds and the description). Additionally, it will check if the room type with the given “room type name” already exists in the system.
    - UC 2.2.1.

- updateRoomType: It updates an existing room type. The administrator specifies which room type he/she wants to modify in order to update a room type. The admin also includes the new room name, the new price, the number of beds, the room number and description.
    UC 2.2.2.
- removeRoomType: It removes a given room type from the system. This method requires only one parameter "the name of the room type" and returns the room type (IRoomType) information.
    - UC 2.2.3.
- addRoom: It adds a new room to the system. The admin enters the room number and the room type (IRoomType). There cannot exist a room with the same room number.
    - UC 2.2.4.
- changeRoomType: It changes the type of a given room. Given a room number, the admin will change the type of that room by choosing a new type in the system. The type can not be the same room's current type.
    - UC 2.2.5.
- removeRoom: It removes a given room from the system. The admin gives a room number. The system checks if a customer is checked in into that room. If the room is unavailable. Returns the information of the room has been removed.
    - UC 2.2.6.
- blockRoom: It blocks a room in the system. The admin provides the number of the room he/she wants to block. The room is marked as blocked.
    - UC 2.2.7.
- unblockRoom: It unblocks a room in the system. The admin provides the number of the room he/she wants to unblock. The room is unblocked making possible new reservations.
    - UC 2.2.8.
- getRoomTypes: Returns a list of all room types.

State Machine Diagram

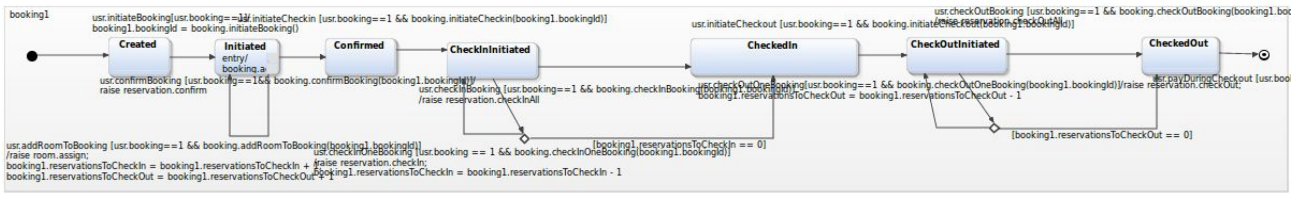


The figure above describes a state machine which models the program flow and execution pattern for when a booking is made. The diagram contains 7 regions. Two regions representing two bookings, two regions representing two different room reservations, two regions representing two different rooms, and the last region is a switch that allows the user to change which of the two bookings is active and can be affected by the different actions.

The behaviour of the state machine can be described as follows; When a booking is created in a hotel a series of reservations can be made for that booking, each associated with a physical room in the hotel. When the guests that requested the booking check in to the hotel they occupy the associated physical rooms. Finally when they finish their stay at the hotel they check out and pay for the booking. The rooms then become free again. Note that reservations can be individually checked in and out.

In the next few subsections, the state machine is broken down in smaller pieces and discussed separately in greater detail.

Bookings


booking1
usr.initiateBooking[usr.booking==1]/
booking1.bookingId = booking.initiateBooking()
usr.initiateCheckin [usr.booking==1 && booking.initiateCheckin(booking1.bookingId)]
Created
Initiated
entry/
booking.a
Confirmed
CheckInInitiated
usr.confirmBooking [usr.booking==1&& booking.confirmBooking(booking1.bookingId)]/
raise reservation.confirm
usr.checkInBooking [usr.booking==1 && booking.checkInBooking(booking1.bookingId)]
/raise reservation.checkInAll
usr.addRoomToBooking [usr.booking==1 && booking.addRoomToBooking(booking1.bookingId)]
/raise room.assign;
booking1.reservationsToCheckIn = booking1.reservationsToCheckIn + 1
booking1.reservationsToCheckOut = booking1.reservationsToCheckOut + 1
usr.checkInOneBooking [usr.booking == 1 && booking.checkInOneBooking(booking1.bookingId)]
/raise reservation.checkIn;
booking1.reservationsToCheckIn = booking1.reservationsToCheckIn - 1
[booking1.reservationsToCheckIn == 0]
usr.initiateCheckout [usr.booking==1 && booking.initiateCheckout(booking1.bookingId)]
CheckedIn
CheckOutInitiated
CheckedOut
usr.checkOutBooking [usr.booking==1 && booking.checkOutBooking(booking1.boo
/raise reservation.checkOutAll
usr.checkOutOneBooking[usr.booking==1 && booking.checkOutOneBooking(booking1.bookingId)]/raise reservation.checkOut;
booking1.reservationsToCheckOut = booking1.reservationsToCheckOut - 1
usr.payDuringCheckout [usr.boo
[booking1.reservationsToCheckOut == 0]

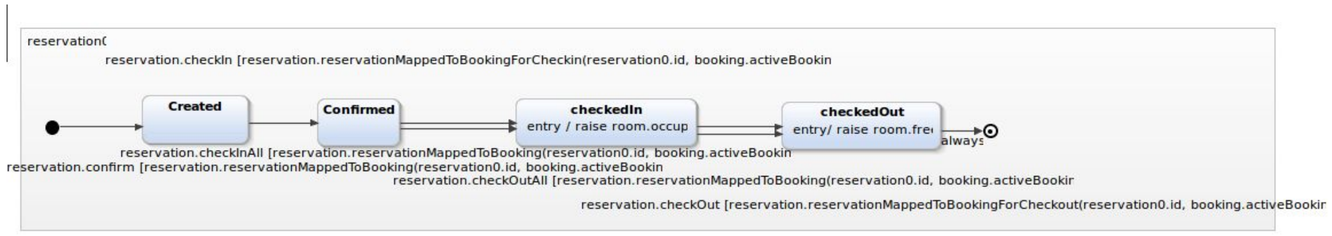
The figure above shows a region representing a booking. The booking region depicts the life cycle of a booking in the hotel management system. There are two booking regions, each with a unique ID but otherwise identical in behaviour. The purpose of having multiple booking regions is to be able to show more complex flows in the state machine. In the state machine the booking will enter the “Created” state by default, however in the actual hotel management system the booking will not be created until needed but since this state machine only models the lifespan of bookings and not the entire system they are initialised immediately. When the booking is created and a user of the system wants to add rooms the booking first has to be initialised. When the booking is initialised it is assigned a unique ID that is used to identify it throughout the rest of the booking process. When the initialisation is done the booking will enter the “Initiated” state. Once initiated, a user can add reservations for rooms to the booking. Multiple reservations can be added to the booking as represented by the looping transition.

When at least one reservation has been added to the booking it can be confirmed. Once confirmed it enters the “Confirmed” state and no new reservations can be added to the booking. The booking must be confirmed before any of the reserved rooms can be checked in.

Once the booking is confirmed the reserved rooms can be checked in. They can either be checked in one by one or all at once. It is always possible to check in all remaining non-checked in reservations with a single action. When at least one reservation is checked in the “CheckInInitiated” state will be entered. Once in this state a user can either check in more rooms separately or check in all remaining reservations at once. When all of the reservations are checked in the booking will automatically enter the “CheckedIn” state.

When all reservations have been checked in and the booking is in a “CheckedIn” state it is possible to check out the reservations. This works in a similar way as checking in. Reservations can either be checked out one by one or all at once. When checking out a reservation the booking enters the “CheckOutInitiated” state and remains in this state until all reservations are checked out. Once all reservations are checked out the booking will automatically enter the “CheckedOut” state which is the last state of the booking.

Reservation

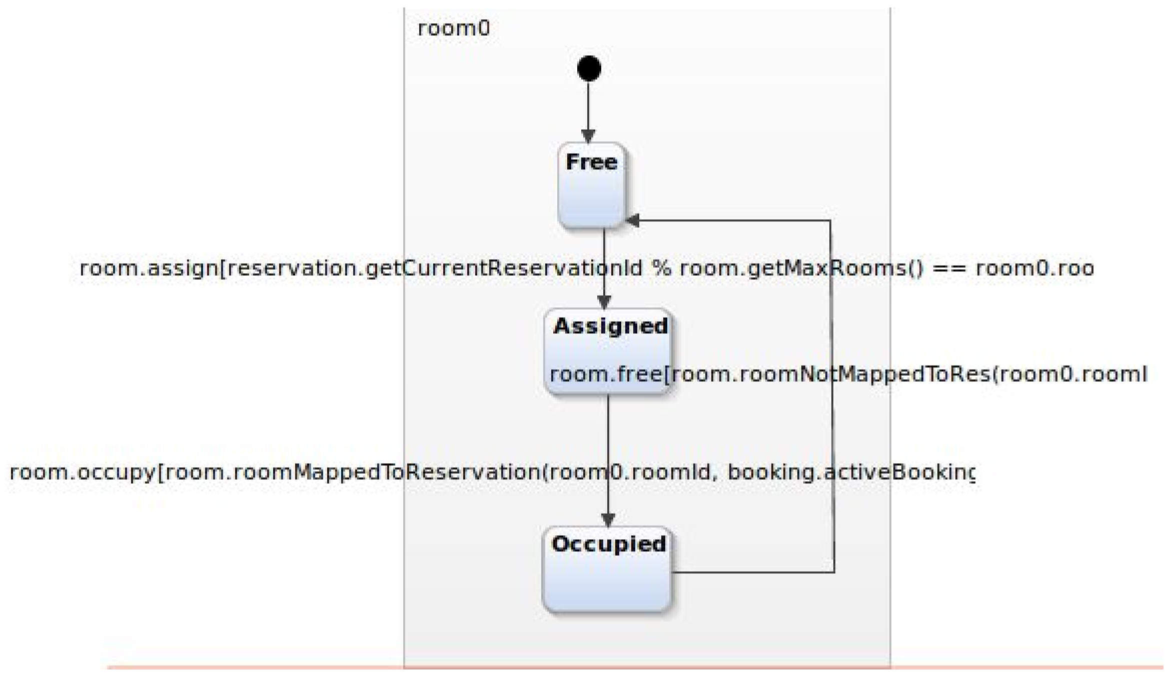

reservation(
reservation.checkIn [reservation.reservationMappedToBookingForCheckin(reservation0.id, booking.activeBookin
Created
Confirmed
checkedIn
entry / raise room.occup
checkedOut
entry/ raise room.fre
always
reservation.checkInAll [reservation.reservationMappedToBooking(reservation0.id, booking.activeBookin
reservation.confirm [reservation.reservationMappedToBooking(reservation0.id, booking.activeBookin
reservation.checkOutAll [reservation.reservationMappedToBooking(reservation0.id, booking.activeBookir
reservation.checkOut [reservation.reservationMappedToBookingForCheckout(reservation0.id, booking.activeBookir

The reservation state machines represent the lifecycle of a logical reservation. The purpose of a reservation region is to connect a booking with a room.

The reservations have four states: Created, Confirmed, CheckedIn and CheckedOut. The Created state is an initial state and is immediately active. In order to transition from the Created to the Confirmed state it is required that the reservation is mapped to a booking. This mapping is performed in the confirmBooking operation and states which reservations belong to which booking. This guard is used between all the state in the reservation.

When a reservation transitions from Confirmed to CheckedIn it will be connected to a room, and when the reservation transitions to CheckedOut the room is decoupled from the reservation. The state machine will never stay in the CheckedOut state because when a reservation is CheckOut it is finished. There are two ways to do the transition to CheckedIn and CheckedOut, this is because in the booking region it is possible to check in one room or all rooms. If the booking region choses to check in all rooms not yet checked in the reservation only needs to check that it is mapped to that booking. If the booking region wants to check in a room separately, all the reservation regions need to check if they are mapped to be the next one to be checked in.

Room


room0
Free
room.assign[reservation.getCurrentReservationId % room.getMaxRooms() == room0.roo
Assigned
room.free[room.roomNotMappedToRes(room0.rooml
room.occupy[room.roomMappedToReservation(room0.roomId, booking.activeBookinc
Occupied

The room state machines represent the lifecycle of the actual rooms of the hotel system. They include three states and can transition between Free, Assigned and Occupied. As specified in the requirements of the assignment 3, it is assumed that all rooms are of the same type. The initial state of the room is Free. Once a booking is initiated and the user attempts to add a new room to that booking id, the statechart evaluates whether any room in the system is available. If a room is available the room's state is set to "Assigned". If the customer wants to reserve a new room, another room is added to that booking id and a second room is marked as "Assigned" in the statechart.

When a customer checks into the hotel system, the state machine raises an event transition in the room side "room occupy". The guard "room.roomMappedToReservation(room0.roomId)" is a condition that checks if the room is mapped to the reservation id. If the guard result returns true, the room will take the transition and be marked as "Occupied". The room's state is changed to "Free" once a customer checks out from the system.

Sequence Diagrams

The sequence diagram differs slightly from the actual state machine. In the sequence diagram the reservation is created only when a room is actually added to the booking, whereas in the state machine all reservations are by default in a created state.

Scenarios

- Check in one room
- Concurrent booking and reservation
- Check in rooms separately
- Check out rooms separately
- Complete a booking

Check in one room

For diagram see appendix A.1

This sequence diagram shows the process of checking in a single room. The sequence starts with the user initiating the booking. When a booking is initiated rooms can be added, and in this case only one room is added to the booking. Adding a room will create a room reservation and assign a room as reserved.

To continue to the check-in phase the booking must be confirmed by the user, which also confirms the reservation. When the booking is then checked in by the user the reservation is marked as checked in and the room is assigned as occupied.

Concurrent booking and reservation

For diagram see appendix A.2

In this scenario the user wants to be able to initiate two bookings at the same time and add one room to the first booking and two to the second. This should assign three rooms (not connected to the booking/reservation). When one of the bookings is checked in, an arbitrary room is occupied for each reservation of that booking.

We chose to model it as if the booking created the reservation. Although that is not exactly as in the state machine, we decided that it made more sense then to have a fixed number of reservation.

Check in rooms separately

For diagram see appendix A.3

The sequence diagram shows the scenario of checking in two rooms separately considering the rooms are associated with the same booking Id (booking 0) and assuming the booking is already confirmed. The scenario starts when the user begins the check in, and the system first evaluates if the given booking ID corresponds to an existing booking in the system. If the condition returns true the booking is marked as checked in and the system raises an event which checks in the reservation. The reservation is only marked as checked in if the guard that checks that the reservation belongs to the booking is true.

Whilst the system is in the “check in” state and the user attempts to check in another room for that booking, the system marks the next available room as “occupied” and relates that room to the booking ID. This continues until the entire booking is checked in. The figure in appendix A.3 shows that when the user checks in reservation 1, the system raises an event which marks the room 1 as occupied. This scenario covers the use case 2.1.11.

Check out rooms separately

For diagram see appendix A.4

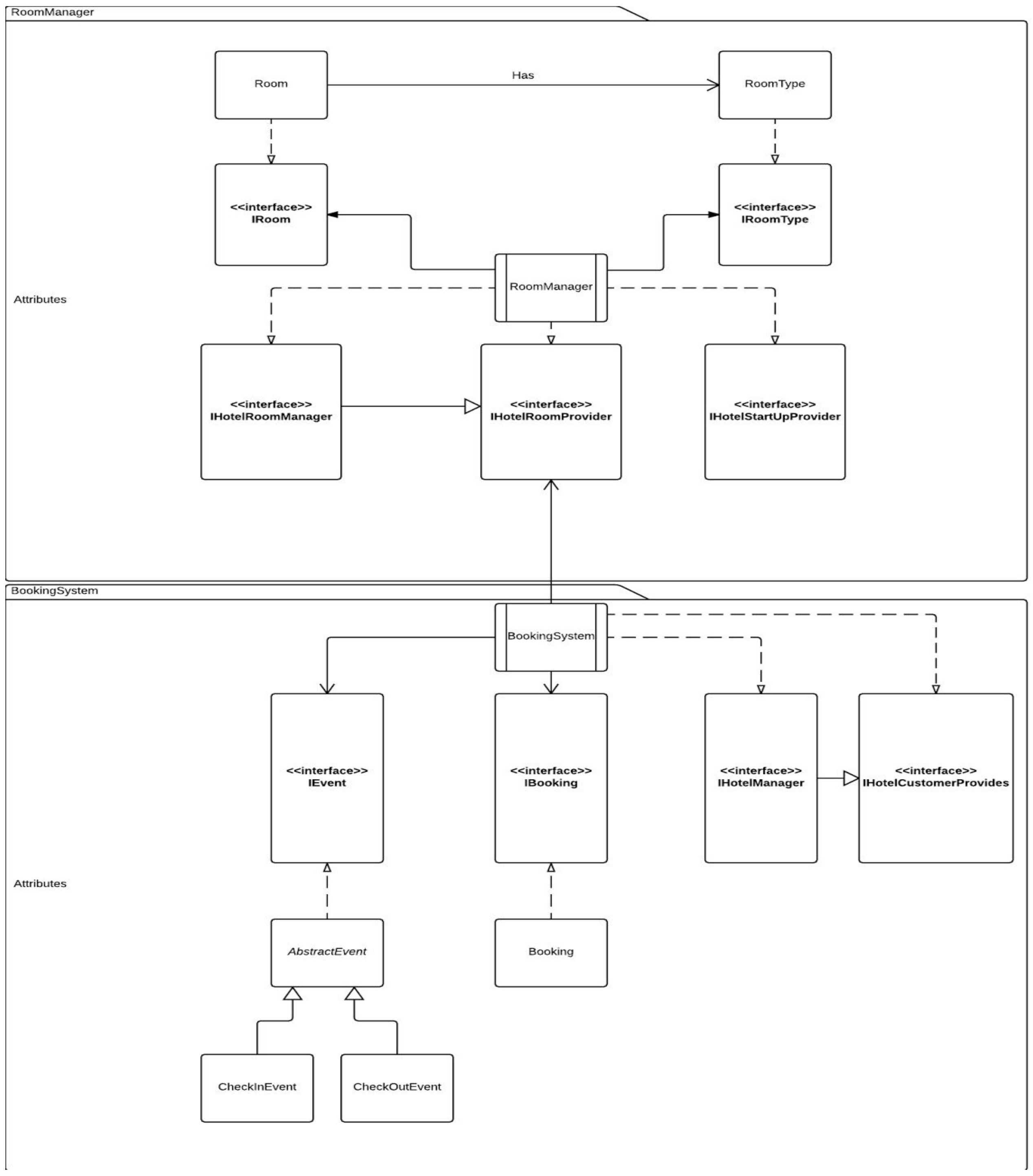
This sequence diagram represents checking out one room from a booking that booked two rooms. It is assumed that the user has already booked and checked in two rooms before this sequence diagram takes place. In the diagram, the user checks out one room from a booking, which in turn checks out one reservation. The reservation frees up the accompanying room and then the reservation is destroyed. The other reservation and room is not affected by this sequence.

Complete a booking

For diagram see appendix A5

This sequence diagram represents a complete booking from start to end of an arbitrary room, as it is implemented in the state machine. A booking is initiated by the user and a room is added to the booking. This creates a reservation, which temporarily marks an arbitrary room as assigned. When the booking is later confirmed the reservation is also confirmed and the room is marked as assigned until the booking ends. The booking is then checked in, checking in the reservation and in turn marking a room as occupied. It remains occupied until the booking is checked out, which checks out the reservation and marks the room as free. The user then pays for the booking.

Class Diagram



Overview of the class diagram, the diagram used to generate can be found in appendix A6.

The class diagram represents the logical view of the hotel system divided into two main packages. This approximately translates into the two components in the component diagram that are not actors, namely RoomManager and BookingSystem. They are each responsible for one of the two major tasks of the system. The RoomManager package handles the administrative tasks such as adding new rooms and room types, blocking/unblocking rooms or updating existing rooms and room types. The BookingSystem package handles the receptionists' interaction with the system, that is making bookings or checking in. Just as shown, these two packages interact through a single simple interface.

Booking System Package

IHotelCustomerProvides

Interface used by the Receptionist and the External Supplier System for making bookings checking out and paying.

- getFreeRooms: Returns the rooms (FreeRoomTypeDTO) available in the system.
- initiateBooking: The user provides the first name, last name, start and end dates. The system creates a booking with the given information. It will return the booking id if the booking is created, or -1 otherwise.
- addRoomToBooking: It takes the room type description and the booking id as a parameters. A room with that given type description is assigned to that booking id. Returns true if the room is added, false otherwise.
- confirmBooking: Takes the booking id as input. A booking can only be confirmed if a reservation with that id is created in the system and one (or more) room is already assigned to it.
- initiateCheckout: Checks out a booking from the system. Given the booking id, the system will check if that id already exists. If so, the system verifies how many rooms are currently checked in to the booking and calculates the final price of every room (the price of the room plus the extra cost price). A check out event is created with the given booking id, the rooms are marked as free and the date of check out. Returns the total price of the booking.
- payDuringCheckout: The user provides the first name, last name and the credit card details of the customer. PayDuringCheckout makes possible pay for all rooms related to a booking. It uses the "payRoomDuringCheckOut" method to be possible to pay for each room in the list.
- initateRoomCheckOut: This method initiates the check out for a specific room. It returns the price of the room along with any additional added costs or -1 if the booking does not exist.
- payRoomDuringCheckOut: Completes the check out for a specific room. It uses the specified credit card details and pays for the room and marks it as available.
- checkInRoom: This method checks in one room to a booking. It requires the description of the room type and the booking id. The system checks if the given room type corresponds to one of the rooms in the list connected to that booking. The system marks that room as

checked in. If more rooms are assigned to that booking, the user can check in each room separately. It returns the room number of the room has been checked in.

IHotelBookingManager

Interface for the Receptionist to perform tasks that the External Supplier System is not allowed to - such task are, listing all bookings editing or cancel bookings.

- initateCheckin: Given a booking id returns a list of rooms possible to check in. If incorrect booking id the method returns an empty list.
- editBookingPeriod: Given a booking id and a new period (start and end date), tries to edit the period. Returns true if successful and false otherwise.
- cancelBooking: Given a booking id, cancels the booking and thereby frees the rooms. Returns true if successful and false otherwise.
- listBookings: Returns a list of all existing bookings.
- listOccupiedRooms: Given a date returns a list of all rooms that are hosting guests that day, i.e. associated to bookings that are checked in.
- listCheckins: Returns a list of all checkInEvents.
- listCheckouts: Returns a list of all checkOutEvents.
- addExtraCostToRoom: Given a booking id, a room and a price adds that price temporary to the room.
- editBookingRooms: Given a booking id, a room type and number of rooms of that type the user wants, removes or adds rooms to end up at that number of rooms.

IBooking

The Booking interface corresponds to the actual bookings in the system. A booking is created initially with the first and last names of a guest and the intended checking in and out periods. When the receptionist adds a room to a booking, each room is added to a list and assigned to the booking id. This means that every booking in the hotel system has a list of rooms linked to it. The list can contain zero or more rooms. Moreover a booking can be edited and change for example the period of stay (start and end dates). To do so the receptionist can “edit” a booking with the booking id and the new corresponding dates as the inputs. Once a booking is confirmed it can be checked in to the system.

The “checking in and out” processes create an event which can be of type “check in” or “check out” respectively. The event is added to a list and stores the booking id, the event type and the date/time of each process. These events are required in order to list check ins and check outs for a specified period.

The last process in the life cycle of a booking is “pay” which is done through the external payment service.

- getRooms: Returns a list of rooms assigned to a booking.
- getFirstName: Returns the first name of the person associated with a booking.
- getLastName: Returns the last name of the person associated with a booking.
- getId: Gets the id of a booking.
- getStartDate: Gets the start date of a booking.
- getEndDate: Gets the end date of a booking.
- setStartDate: Make possible editing a booking by changing the start date.
- setEndDate: Editing the end date of a booking.
- addRoom: Adds one of the free room to the list of rooms assigned to a booking.
- checkInRoom: Requires a room (IRoom) as input. It removes the given room from the list of rooms assigned to the booking and adds that room to the list of rooms have been checked in.
- getCheckedInRooms: Returns the list of rooms have been checked in to the booking.
- checkOutRoom: The user gives the room to check out. The system removes that room from the list of checked in.

IEvent

Represents the check in and check out events of the hotel system. Each event stores a type which can be “check in” or “check out”, a booking id and the date/time of each occurrence in the system.

- getTimeStamp: Returns the time and date of an event.
- getType: Gets the type of an event (if check in or check out event).
- getBookingId: Gets the booking id for a specific event.

Room Manager Package

IHotelRoomProvider

This interface is used to link the Booking System Package with the Room Manager Package, essentially it provides the booking system with the rooms able to host guests.

- getRooms: returns a list of all rooms that are not blocked.

IHotelManager

Interface for the Administrator to manage the hotel, it enables tasks as adding and removing rooms or room types, blocking and unblocking or edit the rooms and room types.

- addRoomType: Given a name for the room type, a price, number of beds and a description creates a new room type with these attributes. If a room type with the same name already exists the method will fail and return false, otherwise it returns true.
- updateRoomType: Given a room type and all attributes for a room type, changes the room types current attributes to the given ones.
- getRoomTypes: returns a list of all room types.
- removeRoomType: Given a room type, tries to remove it. Returns the removed room type.
- addRoom: Given a room number and a room type creates a new room. If there exist a room with the same room number the method fails and returns false, otherwise it returns true.
- changeRoomType: Given a room number and a room type, changes the room type of the room with the given room number and returns true, if no such room number exists returns false.
- removeRoom: Given a room number, tries to remove a room. If someone is currently staying at that room it cannot be removed and the method returns false, otherwise the room is removed and it returns true.
- blockRoom: Given a room number blocks a room.

IHotelStartUpProvider

Used by the administrator to initialise the hotel.

- startUp: Given a number of rooms creates that many rooms.

IRoom

The Room interface represents the physical rooms of the hotel system. Each room has a type, a number and can be blocked or unblocked in the system. A room is linked to a booking once the receptionist adds a room to it. Once this room has been added, the receptionist can retrieve, edit or modify the values of some features of the room. The receptionist can, for example, get the price and the number of beds of a room, modify the room type and the number of rooms when editing a booking, set an extra cost for a room and get that cost when paying or checking out a booking. This is done through the booking interface.

On the other hand the admin can set a room as blocked, making it impossible for the receptionist to add this room to a booking. The admin can also edit a room type in the system. IRoom has a room type IRoomType and therefore all behaviours of a IRoomType make up a room in the system.

- getRoomType: Returns a room type (IRoomType).
- getRoomNumber: Returns the room number.
- setRoomType: Sets a new room type.
- setIsBlocked: Sets true/false when a room is blocked/unblocked.

- isBlocked: Checks the current status of a room in the system (if the room is available or blocked).
- getExtraCostDescription: Returns the description of the extra cost of a room.
- setExtraCostDescription: Sets the description of the extra cost of a room.
- getExtraCostPrice: Returns the price of the extra cost.
- addExtraCost: Adds an extra cost to a room.
- setOccupied: Sets true/false whether a room is occupied or not.
- isOccupied: Returns if the room is occupied or not.

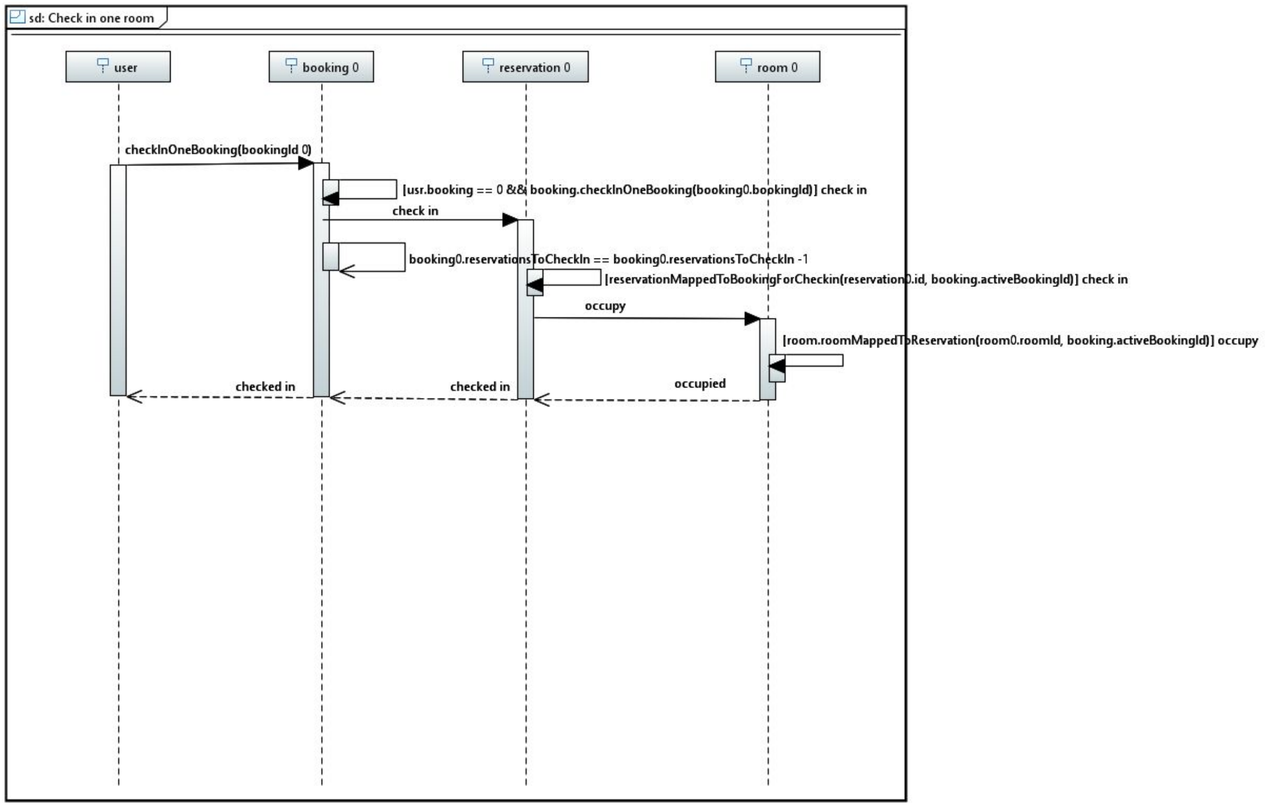
IRoomType

The interface IRoomType contains the main methods responsible for retrieving or changing a value of a room type.

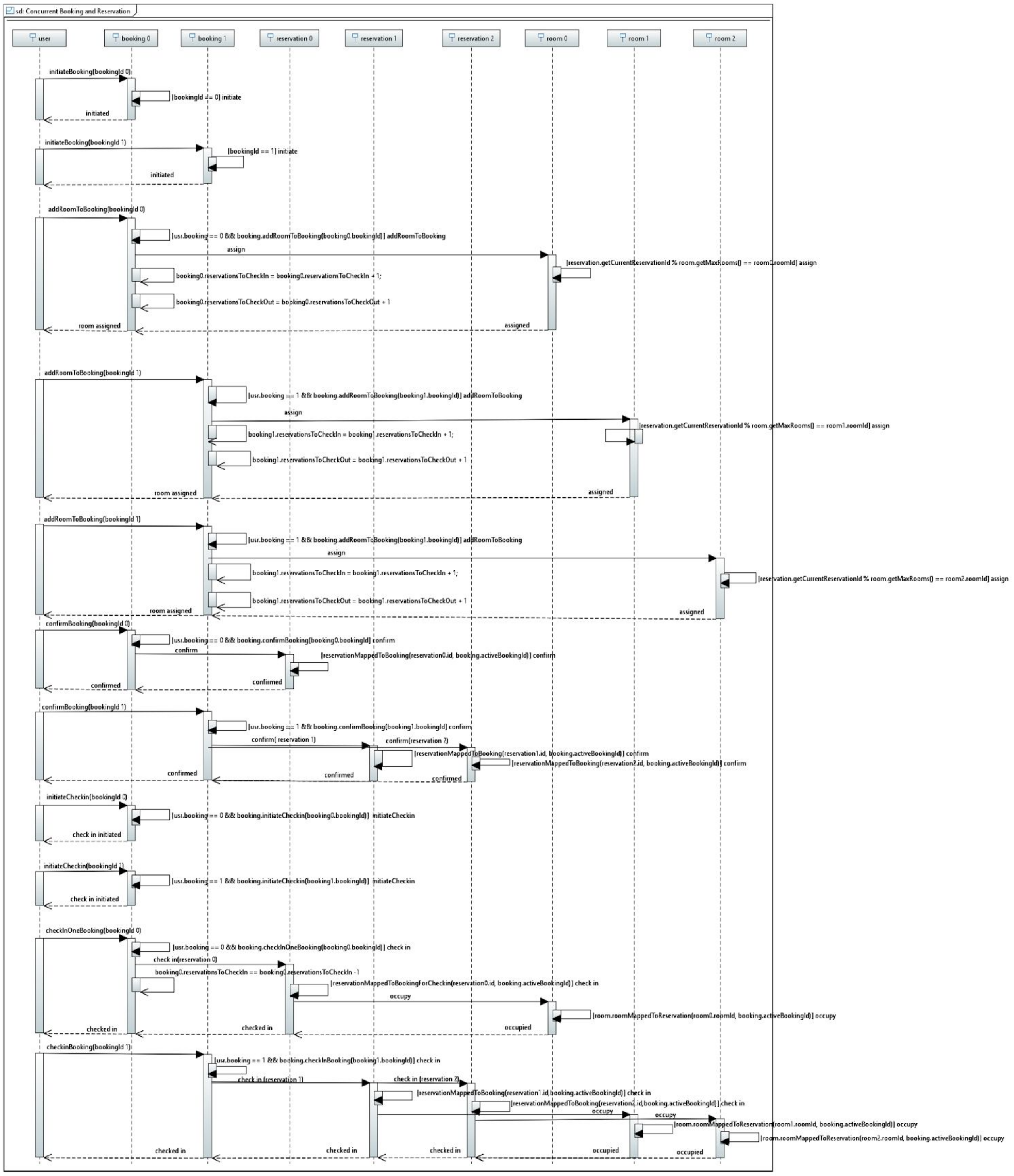
- getName: Gets the name of a room type.
- getPrice: Gets the price of a room type.
- getNumberOfBeds: Returns the number of beds of a room.
- getDescription: Gets the description.
- setName: Sets the name of a room.
- setPrice: Sets the price of a room.
- setNumberOfBeds: Sets the number of beds.
- setDescription: Sets the description.

Appendix

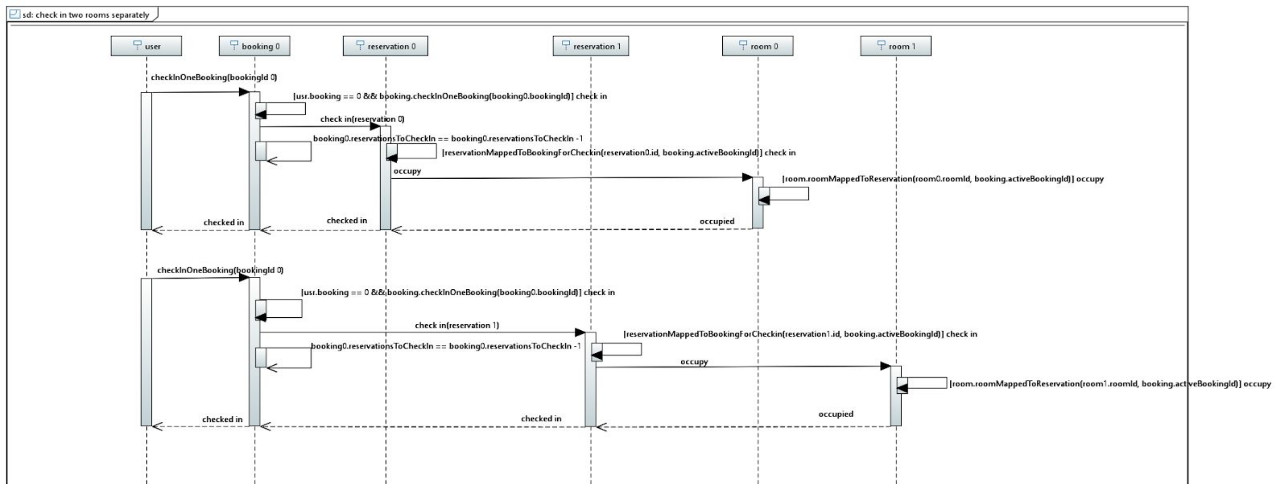
A.1


sd: Check in one room
user
booking 0
reservation 0
room 0
checkInOneBooking(bookingId 0)
[usr.booking == 0 && booking.checkInOneBooking(booking0.bookingId)] check in
check in
booking0.reservationsToCheckIn == booking0.reservationsToCheckIn -1
[reservationMappedToBookingForCheckin(reservation0.id, booking.activeBookingId)] check in
occupy
[room.roomMappedToReservation(room0.roomId, booking.activeBookingId)] occupy
checked in
checked in
occupied

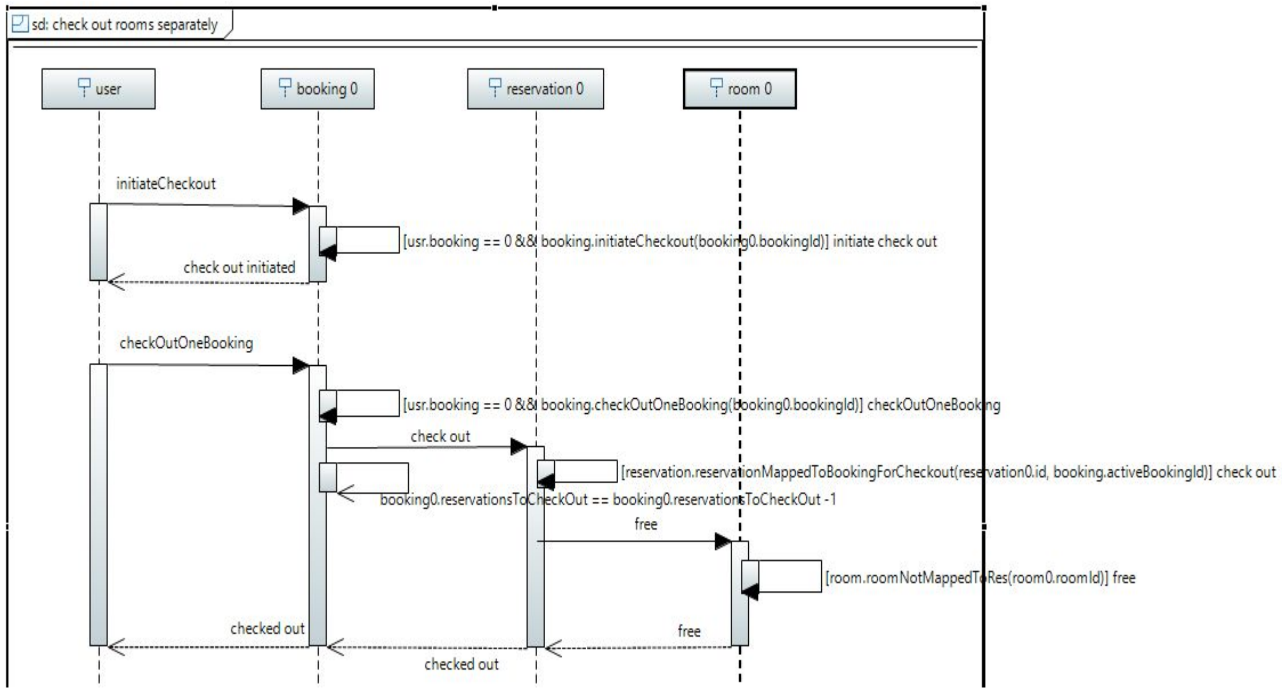
A.2


sd: Concurrent Booking and Reservation
user
booking 0
booking 1
reservation 0
reservation 1
reservation 2
room 0
room 1
room 2
initiateBooking(bookingId 0)
[bookingId == 0] initiate
initiated
initiateBooking(bookingId 1)
[bookingId == 1] initiate
initiated
addRoomToBooking(bookingId 0)
[usr.booking == 0 && booking.addRoomToBooking(booking0.bookingId)] addRoomToBooking
assign
[reservation.getCurrentReservationId % room.getMaxRooms() == room0.roomId] assign
booking0.reservationsToCheckIn = booking0.reservationsToCheckIn + 1;
booking0.reservationsToCheckOut = booking0.reservationsToCheckOut + 1
room assigned
assigned
addRoomToBooking(bookingId 1)
[usr.booking == 1 && booking.addRoomToBooking(booking1.bookingId)] addRoomToBooking
assign
[reservation.getCurrentReservationId % room.getMaxRooms() == room1.roomId] assign
booking1.reservationsToCheckIn = booking1.reservationsToCheckIn + 1;
booking1.reservationsToCheckOut = booking1.reservationsToCheckOut + 1
room assigned
assigned
addRoomToBooking(bookingId 1)
[usr.booking == 1 && booking.addRoomToBooking(booking1.bookingId)] addRoomToBooking
assign
booking1.reservationsToCheckIn = booking1.reservationsToCheckIn + 1;
[reservation.getCurrentReservationId % room.getMaxRooms() == room2.roomId] assign
booking1.reservationsToCheckOut = booking1.reservationsToCheckOut + 1
room assigned
assigned
confirmBooking(bookingId 0)
[usr.booking == 0 && booking.confirmBooking(booking0.bookingId)] confirm
confirm
[reservationMappedToBooking(reservation0.id, booking.activeBookingId)] confirm
confirmed
confirmed
confirmBooking(bookingId 1)
[usr.booking == 1 && booking.confirmBooking(booking1.bookingId)] confirm
confirm(reservation 1)
confirm(reservation 2)
[reservationMappedToBooking(reservation1.id, booking.activeBookingId)] confirm
[reservationMappedToBooking(reservation2.id, booking.activeBookingId)] confirm
confirmed
confirmed
confirmed
initiateCheckin(bookingId 0)
[usr.booking == 0 && booking.initiateCheckin(booking0.bookingId)] initiateCheckin
check in initiated
initiateCheckin(bookingId 1)
[usr.booking == 1 && booking.initiateCheckin(booking1.bookingId)] initiateCheckin
check in initiated
checkInOneBooking(bookingId 0)
[usr.booking == 0 && booking.checkInOneBooking(booking0.bookingId)] check in
check in(reservation 0)
booking0.reservationsToCheckIn == booking0.reservationsToCheckIn -1
[reservationMappedToBookingForCheckin(reservation0.id, booking.activeBookingId)] check in
occupy
[room.roomMappedToReservation(room0.roomId, booking.activeBookingId)] occupy
checked in
checked in
occupied
checkinBooking(bookingId 1)
[usr.booking == 1 && booking.checkInBooking(booking1.bookingId)] check in
check in (reservation 1)
check in (reservation 2)
[reservationMappedToBooking(reservation1.id,booking.activeBookingId)] check in
[reservationMappedToBooking(reservation2.id,booking.activeBookingId)] check in
occupy
occupy
[room.roomMappedToReservation(room1.roomId, booking.activeBookingId)] occupy
[room.roomMappedToReservation(room2.roomId, booking.activeBookingId)] occupy
checked in
checked in
checked in
occupied
occupied

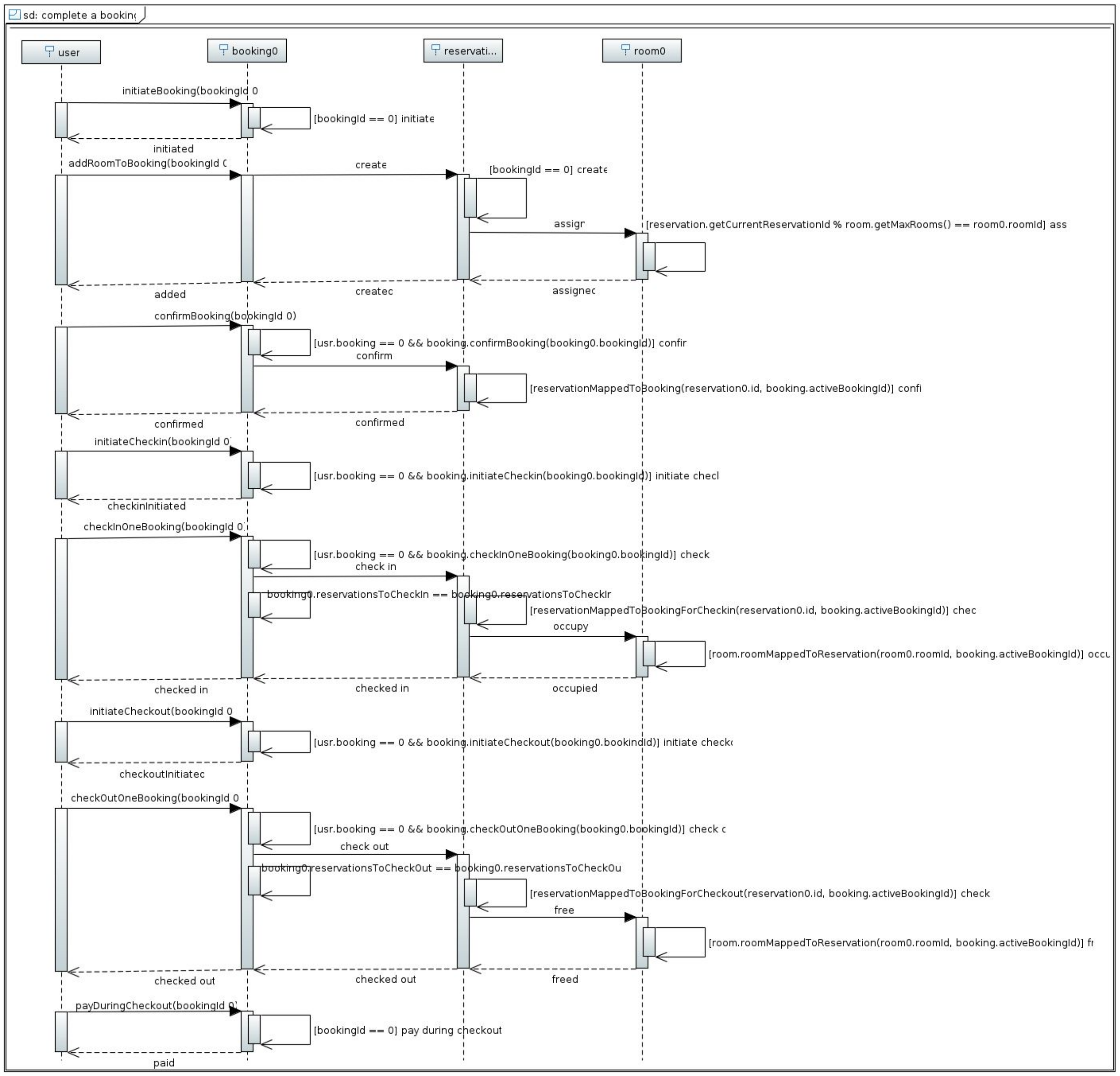
A.3


sd: check in two rooms separately
user
booking 0
reservation 0
reservation 1
room 0
room 1
checkInOneBooking(bookingId 0)
[usr.booking == 0 && booking.checkInOneBooking(booking0.bookingId)] check in
check in(reservation 0)
booking0.reservationsToCheckIn == booking0.reservationsToCheckIn -1
[reservationMappedToBookingForCheckin(reservation0.id, booking.activeBookingId)] check in
occupy
[room.roomMappedToReservation(room0.roomId, booking.activeBookingId)] occupy
checked in
checked in
occupied
checkInOneBooking(bookingId 0)
[usr.booking == 0 && booking.checkInOneBooking(booking0.bookingId)] check in
check in(reservation 1)
[reservationMappedToBookingForCheckin(reservation1.id, booking.activeBookingId)] check in
booking0.reservationsToCheckIn == booking0.reservationsToCheckIn -1
occupy
[room.roomMappedToReservation(room1.roomId, booking.activeBookingId)] occupy
checked in
checked in
occupied

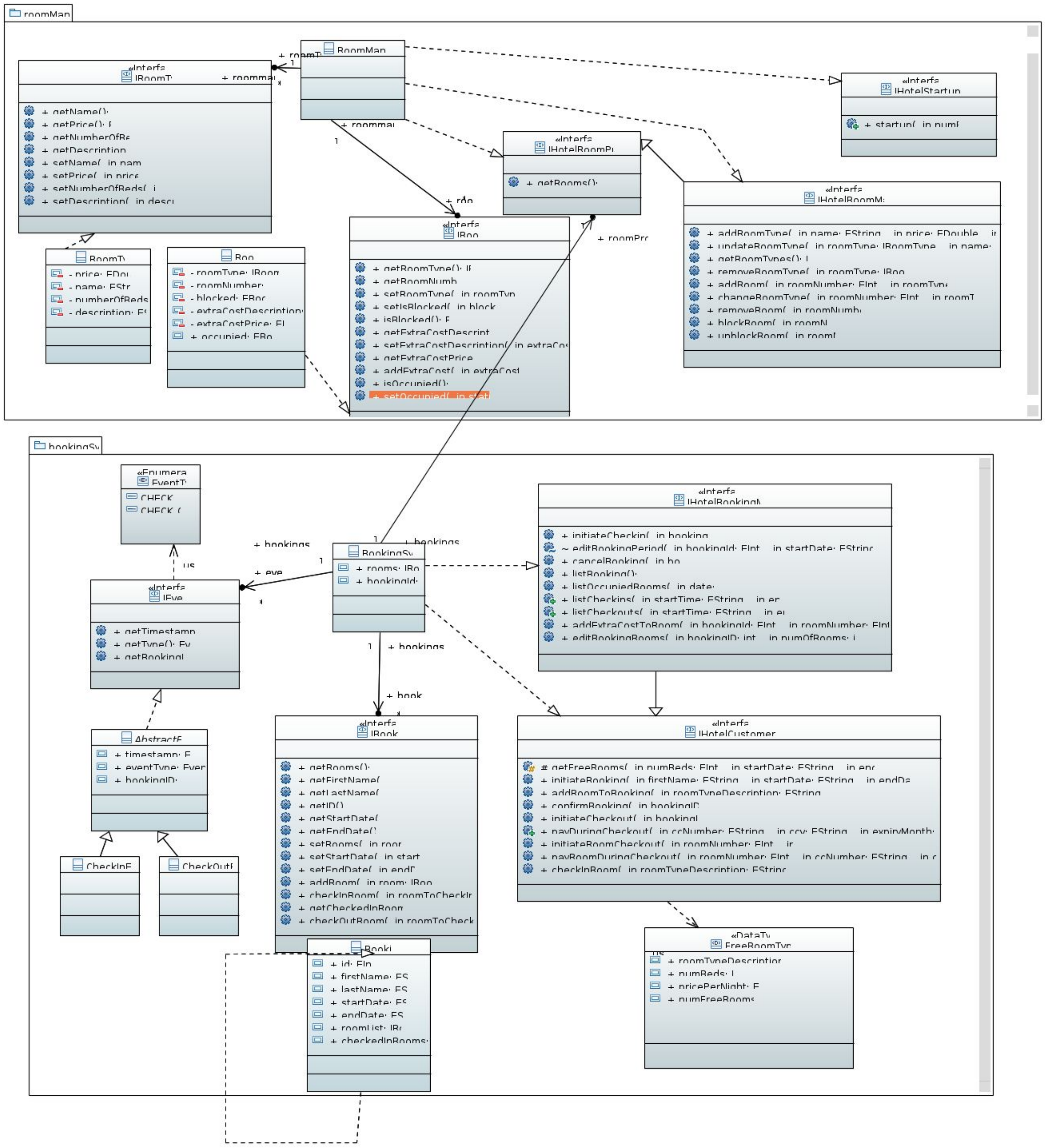
A4


sd: check out rooms separately
user
booking 0
reservation 0
room 0
initiateCheckout
[usr.booking == 0 && booking.initiateCheckout(booking0.bookingId)] initiate check out
check out initiated
checkOutOneBooking
[usr.booking == 0 && booking.checkOutOneBooking(booking0.bookingId)] checkOutOneBooking
check out
[reservation.reservationMappedToBookingForCheckout(reservation0.id, booking.activeBookingId)] check out
booking0.reservationsToCheckOut == booking0.reservationsToCheckOut -1
free
[room.roomNotMappedToRes(room0.roomId)] free
checked out
free
checked out

A5


sd: complete a bookin
user
booking0
reservati...
room0
initiateBooking(bookingId 0
[bookingId == 0] initiate
initiated
addRoomToBooking(bookingId (
create
[bookingId == 0] create
assigr
[reservation.getCurrentReservationId % room.getMaxRooms() == room0.roomId] ass
added
createc
assignec
confirmBooking(bookingId 0)
[usr.booking == 0 && booking.confirmBooking(booking0.bookingId)] confir
confirm
[reservationMappedToBooking(reservation0.id, booking.activeBookingId)] confi
confirmed
confirmed
initiateCheckin(bookingId 0
[usr.booking == 0 && booking.initiateCheckin(booking0.bookingId)] initiate checl
checkinInitiated
checkInOneBooking(bookingId 0
[usr.booking == 0 && booking.checkInOneBooking(booking0.bookingId)] check
check in
booking0.reservationsToCheckIn == booking0.reservationsToCheckIr
[reservationMappedToBookingForCheckin(reservation0.id, booking.activeBookingId)] chec
occupy
[room.roomMappedToReservation(room0.roomId, booking.activeBookingId)] occu
checked in
checked in
occupied
initiateCheckout(bookingId 0
[usr.booking == 0 && booking.initiateCheckout(booking0.bookindId)] initiate check
checkoutInitiatec
checkOutOneBooking(bookingId 0
[usr.booking == 0 && booking.checkOutOneBooking(booking0.bookingId)] check c
check out
booking0.reservationsToCheckOut == booking0.reservationsToCheckOu
[reservationMappedToBookingForCheckout(reservation0.id, booking.activeBookingId)] check
free
[room.roomMappedToReservation(room0.roomId, booking.activeBookingId)] fr
checked out
checked out
freed
payDuringCheckout(bookingId 0
[bookingId == 0] pay during checkout
paid

A6



roomMan
RoomMan
«Interfa IRoomTy
+ roommai
+ roomT
+ getName():
+ getPrice(): F
+ getNumberOfBe
+ getDescription
+ setName(in nam
+ setPrice(in price
+ setNumberOfBeds(i
+ setDescription(in descr
RoomTy
- price: EDou
- name: EStr
- numberOfBeds
- description: ES
Roo
- roomType: IRoom
- roomNumber:
- blocked: EBoo
- extraCostDescription:
- extraCostPrice: EI
+ occupied: EBo
+ roommai
«Interfa IHotelRoomPr
+ getRooms():
+ roo
«Interfa IRoo
+ getRoomType(): IR
+ getRoomNumb
+ setRoomType(in roomTyp
+ setIsBlocked(in block
+ isBlocked(): E
+ getExtraCostDescript
+ setExtraCostDescription(in extraCo
+ getExtraCostPrice
+ addExtraCost(in extraCost
+ isOccupied():
+ setOccupied(in stat
+ roomPro
«Interfa IHotelStartup
+ startup(in numb
«Interfa IHotelRoomM
+ addRoomType(in name: EString in price: EDouble in
+ updateRoomType(in roomType: IRoomType in name:
+ getRoomTypes(): I
+ removeRoomType(in roomType: IRoo
+ addRoom(in roomNumber: EInt in roomType
+ changeRoomType(in roomNumber: EInt in roomT
+ removeRoom(in roomNumb
+ blockRoom(in roomN
+ unblockRoom(in roomN
bookingSy
«Enumera EventTy
CHECK
CHECK
us
«Interfa IEve
+ getTimestamp
+ getType(): Ev
+ getBookingI
+ bookings
+ eve
BookingSy
+ rooms: IRoo
+ bookingId:
+ bookings
+ bookings
+ book
AbstractE
+ timestamp: E
+ eventType: Even
+ bookingID:
CheckInE
CheckOutE
«Interfa IHotelBookingM
+ initiateCheckin(in booking
~ editBookingPeriod(in bookingId: EInt in startDate: EString
+ cancelBooking(in boo
+ listBooking():
+ listOccupiedRooms(in date:
+ listCheckins(in startTime: EString in en
+ listCheckouts(in startTime: EString in en
+ addExtraCostToRoom(in bookingId: EInt in roomNumber: EInt
+ editBookingRooms(in bookingID: int in numOfRooms: i
«Interfa IBook
+ getRooms():
+ getFirstName(
+ getLastName(
+ getID()
+ getStartDate(
+ getEndDate()
+ setRooms(in roor
+ setStartDate(in start
+ setEndDate(in endD
+ addRoom(in room: IRoo
+ checkInRoom(in roomToCheckIn
+ getCheckedInRoom
+ checkOutRoom(in roomToCheck
Booki
+ id: EIn
+ firstName: ES
+ lastName: ES
+ startDate: ES
+ endDate: ES
+ roomList: IRo
+ checkedInRooms:
«Interfa IHotelCustomer
# getFreeRooms(in numBeds: EInt in startDate: EString in end
+ initiateBooking(in firstName: EString in startDate: EString in endDa
+ addRoomToBooking(in roomTypeDescription: EString
+ confirmBooking(in bookingID
+ initiateCheckout(in bookingI
+ payDuringCheckout(in ccNumber: EString in ccv: EString in expiryMonth:
+ initiateRoomCheckout(in roomNumber: EInt in
+ payRoomDuringCheckout(in roomNumber: EInt in ccNumber: EString in c
+ checkInRoom(in roomTypeDescription: EString
«DataTy FreeRoomTyp
+ roomTypeDescription
+ numBeds: I
+ pricePerNight: E
+ numFreeRooms