Object Oriented Design

Analysis

Dr. Seán Russell

School of Computer Science,
University College Dublin

Lecture 06

Table of Contents

Section Contents

- Analysis and Design are similar activities
- We should understand what the purpose of Analysis
- Analysis is the task of describing **how** interacting object can perform the use cases
  - A big part of this is proving that the **task is possible**

- We use a technique known as **realisation**
- High-level interactions are developed for each use case/course of events
- Shows **how** task will achieved with the classes/objects we have defined
- Any new classes or updates to attributes and operations will be added to the class diagram
  - We start with the domain model from the previous phase
- Realisations are defined in UML using **interaction diagrams**
  - We will use sequence diagrams

- Analysis and design are parts of the same process
  - Analysis focuses on representing the application domain
  - Design focuses on representing the whole software product
- We will come to a more reliable difference later

Section Contents

- Object design is a very creative activity
  - Multiple possibilities, difficult to know which is best
- For each piece of functionality, we must decide
  - Where any required data will be stored
  - Where any required processing will be done
- The domain model may be a good starting point, but
  - Domain models usually do not show operations
  - There will always be more classes required in the final system

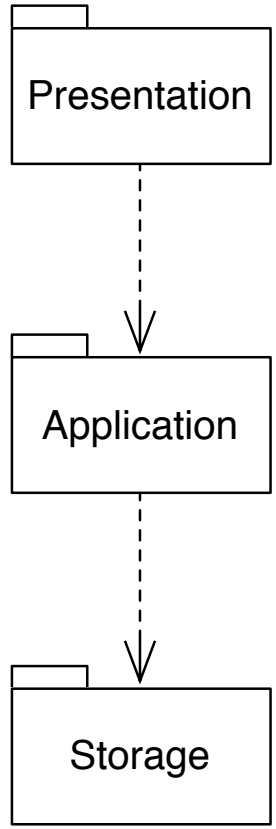
- A very commonly used principle is the idea of object responsibilities
- Objects can have two types of responsibilities:
  - Maintain some data
    - In attributes
    - In links to other objects
  - Support some processing
    - By performing calculations
    - By calling other objects methods

Section Contents

- There are many choices when it comes to how we can design software
- A number of patterns exist that that have been seen many times
- Provide us with examples of good and bad design choices
- Very high-level patterns are often described as software architecture

- Layered architectures are designed to help build stable designs
- The concept is to allocate different responsibilities to different subsystems
- Create clean interfaces between layers
- Can be a good architecture for desktop applications with GUI and data storage
- Key idea is management of dependencies

- Layers are represented by packages in UML and are shown as a tabbed folder
- Arrows in this diagram show **dependencies** between the packages
  - Upper layers can use lower layers
  - Lower layers are independent
- Storage layer responsible for managing data persistence



- The MVC pattern makes a clear distinction between parts of the system responsible for maintaining data and presenting data
- MVC recommends responsibilities be split between
  - **Model** classes responsible for maintaining data
  - **View** classes responsible for presenting data
  - **Control** classes responsible for managing functionality
- Key idea is management of dependencies (again)

- A distinction between model and view can be made at a system wide level, leading to identification of two layers in the architecture
  - Classes that maintain the system state and implement the business logic of the application belong in the **application** layer
  - Classes concerned with the user interface are placed in the **presentation** layer

- This architecture allows us a different view on the differences
  - Analysis is typically concerned with the behaviour and interaction of objects in the application layer
  - Design is concerned with object design at all layers of the architecture
- Requirements of the presentation and storage layers are often common to many systems

- Layered architectures are primarily concerned with isolating code from the effect of change
- The direction of the dependencies mean that we can change higher layers without effecting lower layers
- Any change we make to a lower layer may require that we change the code in the higher layer that uses it
- This is a key for maintainable software

Section Contents

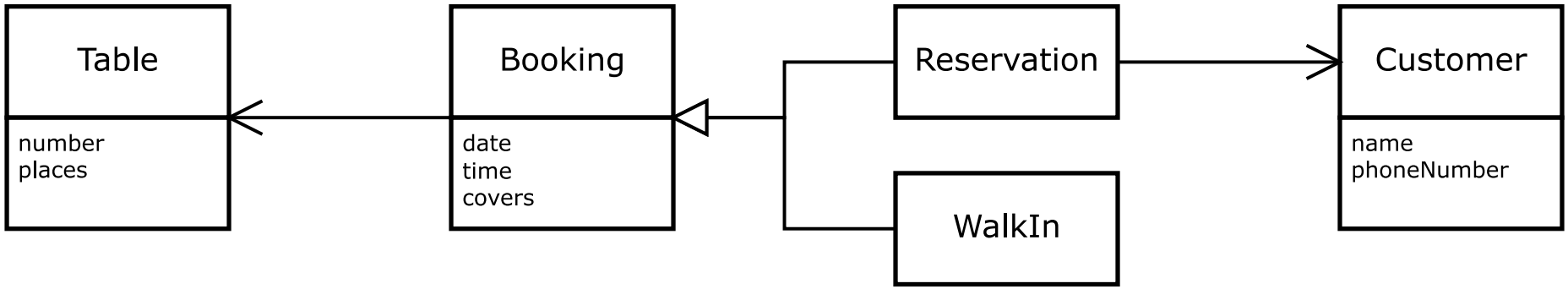
- In the following sections we are going to complete the realisation process for our use cases
  - In this case I will only do the basic course of events
- The use cases we have are
  1. Display date
  2. Add Reservation
  3. Add WalkIn
  4. Alter Booking
  5. Delete Booking

Basic Course of Events

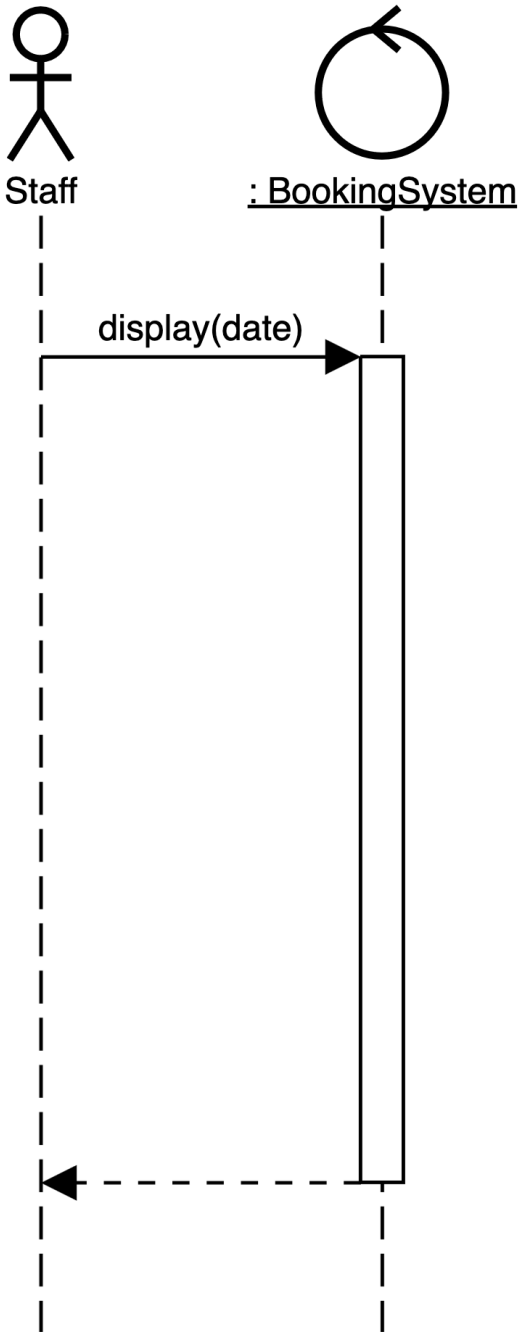
1. The user enters a date
2. The system displays the bookings for that date

- The users interaction is a request to display the bookings on a date
- This can be modelled as a single message
- In response to this message, the system retrieves and displays the bookings for that date

- The message that begins the use case is called a **system message**
- It is passed from the relevant actor to the system
- We can assume that these messages have all of the parameters required to complete the use case

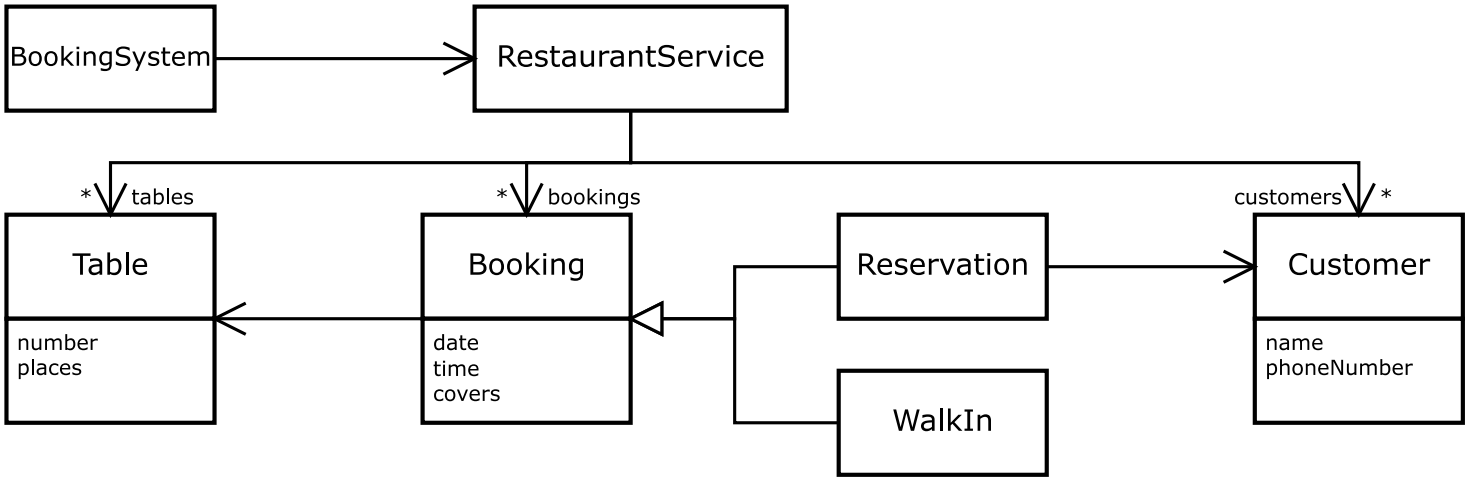


- Current model is missing a good object to receive this message
- The receiving object will be responsible for completing the use case
- One controller should be enough for our application

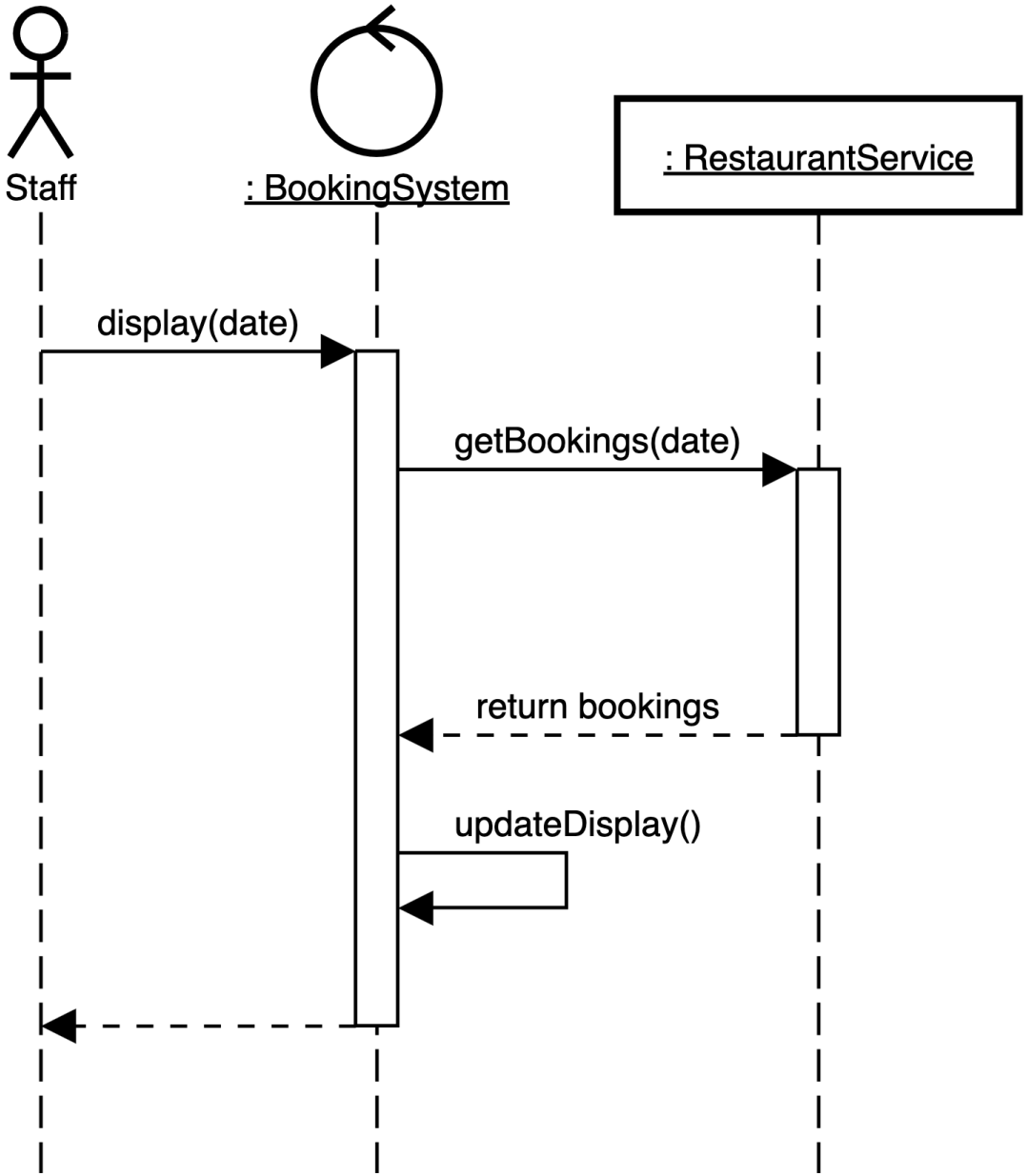

Staff
: BookingSystem
display(date)

- We have not fully realised the display booking use case
- We have not shown how some steps can be done
  - Find the bookings for the date
  - The update of the display to show them
- Which class remembers all of the bookings?

- We have not fully realised the display booking use case
- We have not shown how some steps can be done
  - Find the bookings for the date
  - The update of the display to show them
- Which class remembers all of the bookings?

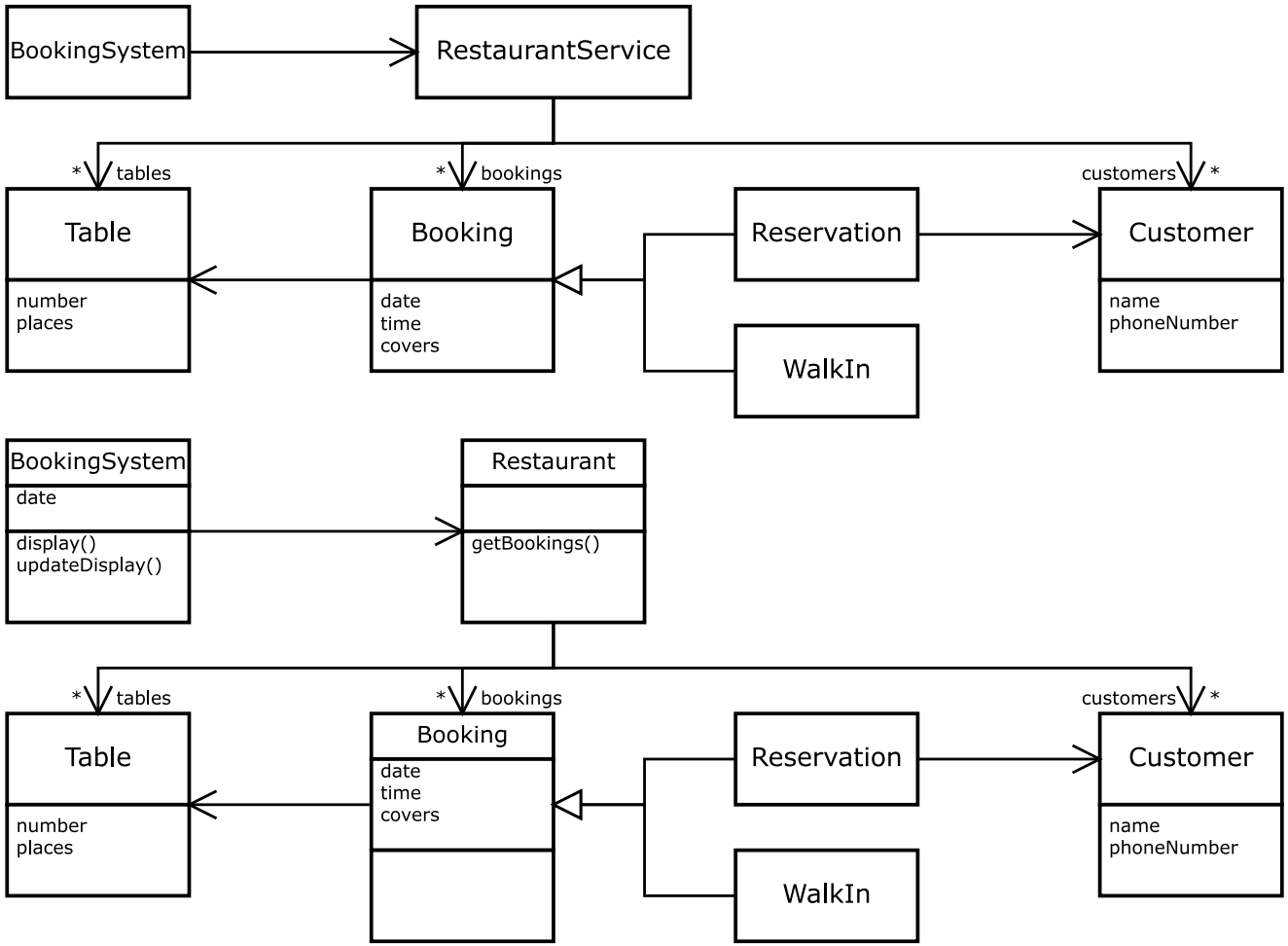


- We have now added the `RestaurantService` class
- This is a class with one overall responsibility, remembering the important objects in the system
  - I.e. Customers and Bookings
- We will assume that this class has method we can use to find, delete and save these objects
- In the design phase we will connect these methods to the database interface!


Staff
: BookingSystem
: RestaurantService
display(date)
getBookings(date)
return bookings
updateDisplay()

- This level of detail is sufficient for analysis
  - We needed to show it was possible
- Other details would be left to complete during the design phase
  - Details of the parameters of methods
  - Details of the types and return types

- In this process we have identified new classes and messages
- We should add this information to the class diagram
  - Operation added where messages are **received**
  - Associations added between communicating objects
    - Arrows show the direction of communication

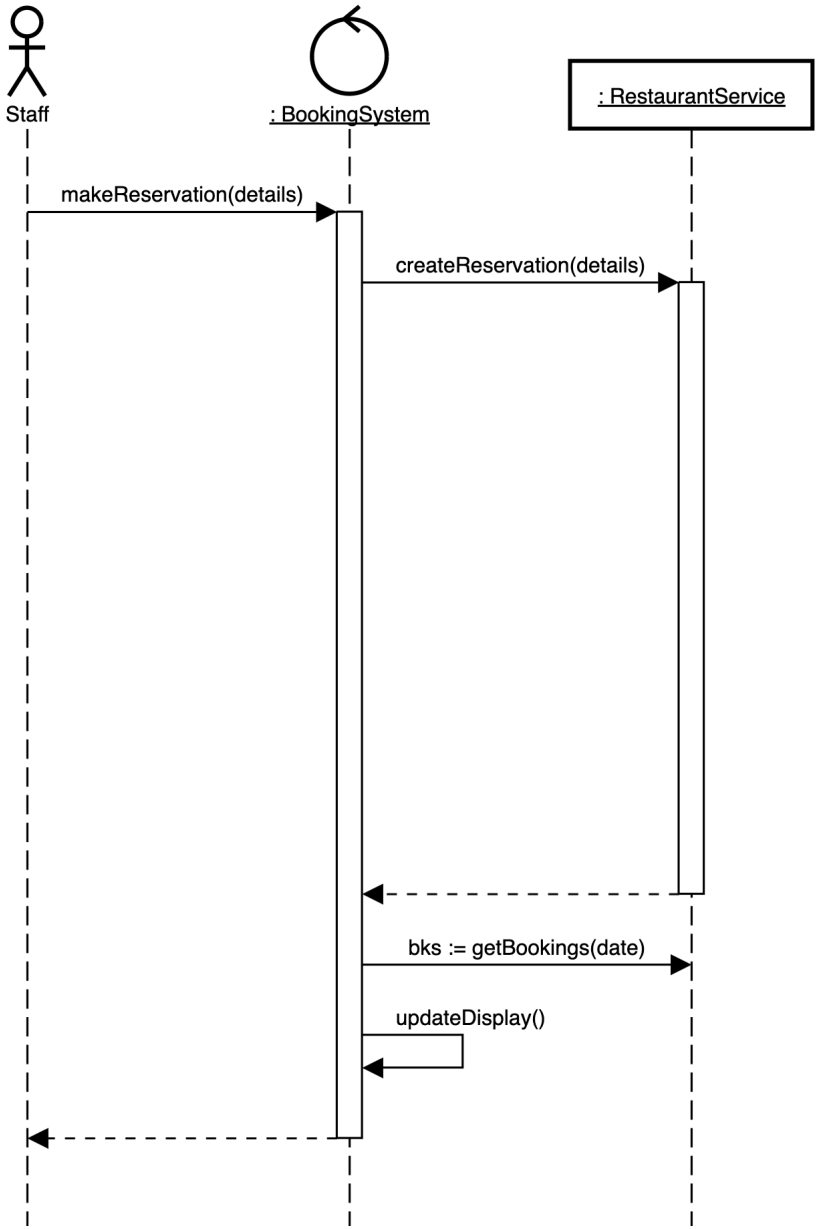

BookingSystem
RestaurantService
* tables
* bookings
customers *
Table
number
places
Booking
date
time
covers
Reservation
WalkIn
Customer
name
phoneNumber
BookingSystem
date
display()
updateDisplay()
Restaurant
getBookings()
* tables
* bookings
customers *
Table
number
places
Booking
date
time
covers
Reservation
WalkIn
Customer
name
phoneNumber

Section Contents

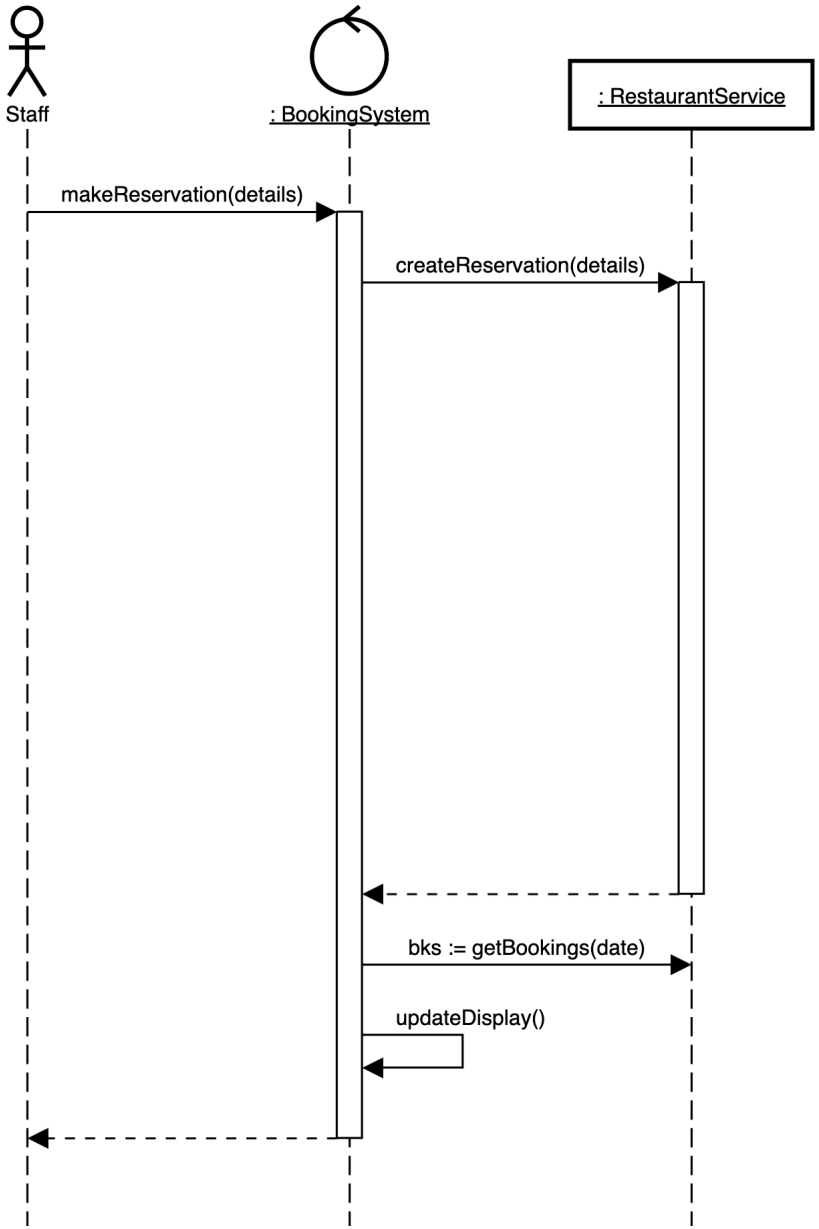
Basic Course of Events

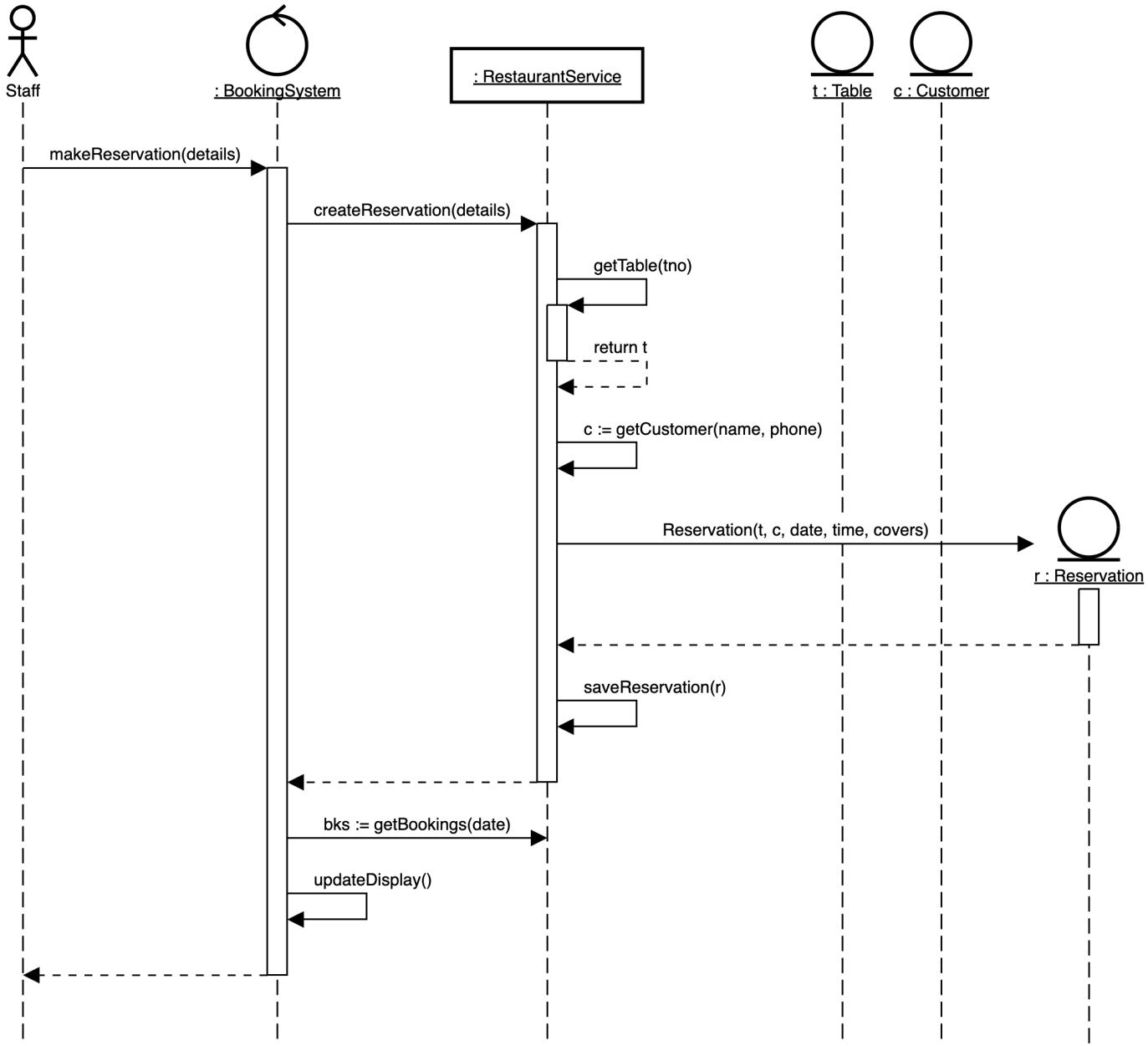
1. The receptionist performs the `Display date` use case
2. There is a suitable table available; the receptionist enters the customers details and booking details
3. The system records the new booking

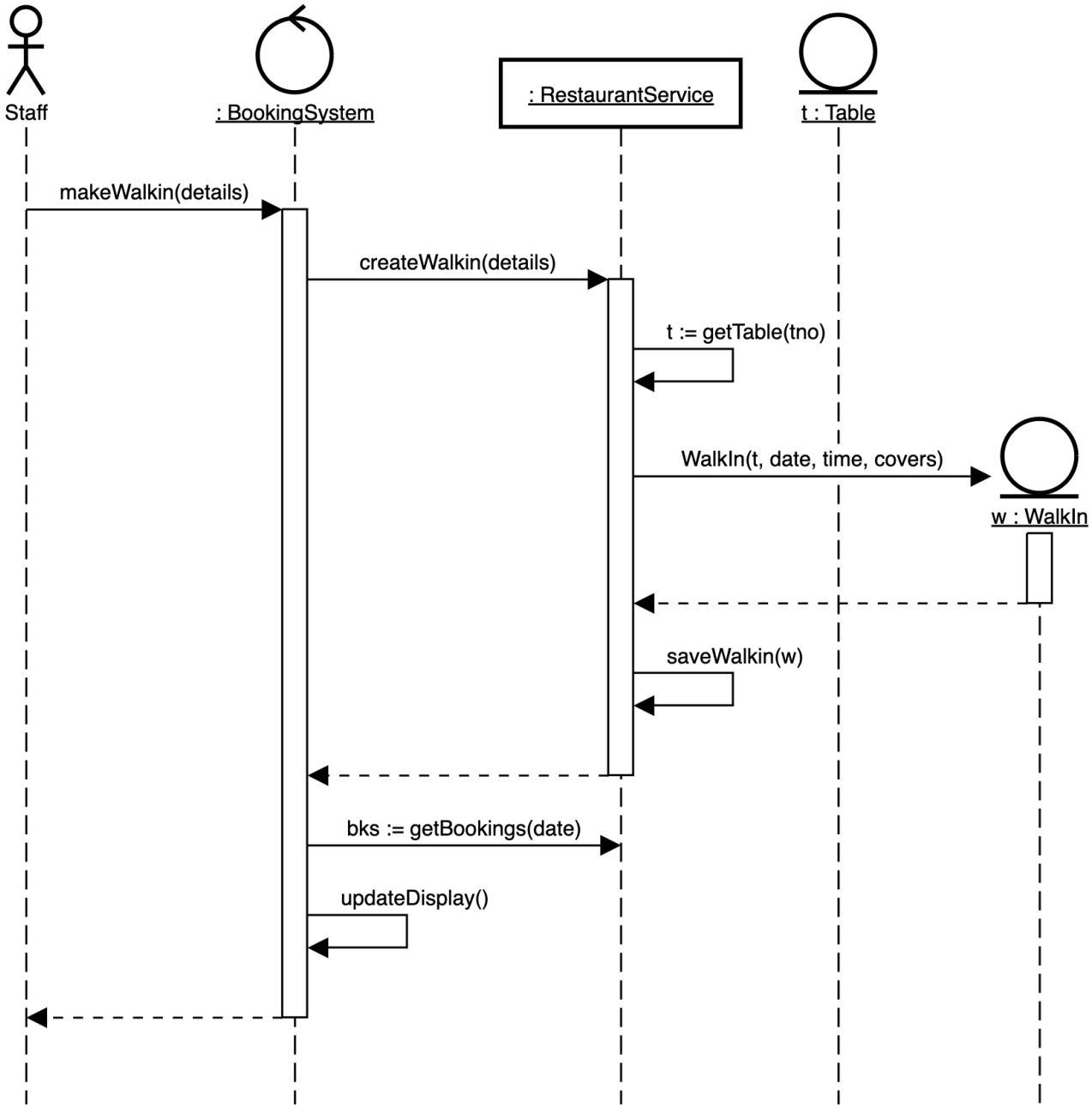
- Again, we will model user interaction as a single system message
- We must choose which class has the responsibilities to create bookings

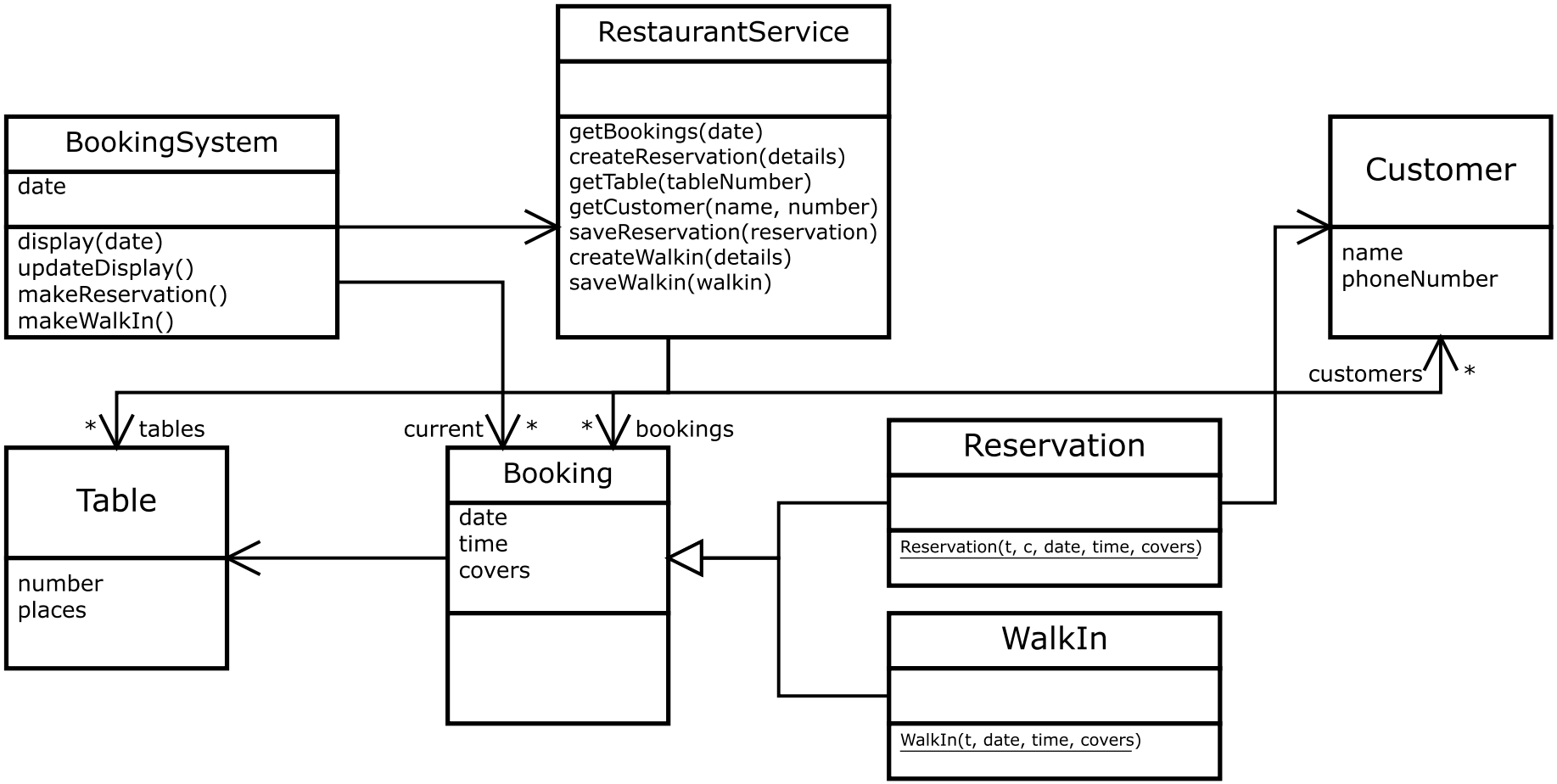

Staff
: BookingSystem
: RestaurantService
makeReservation(details)
createReservation(details)
bks := getBookings(date)
updateDisplay()

- We need to consider what happens in the restaurant object when it receives the createBooking message
  - We must find the correct customer and table objects
  - These are required to construct the reservation object


Staff
: BookingSystem
: RestaurantService
makeReservation(details)
createReservation(details)
bks := getBookings(date)
updateDisplay()


Staff
: BookingSystem
: RestaurantService
t : Table
c : Customer
makeReservation(details)
createReservation(details)
getTable(tno)
return t
c := getCustomer(name, phone)
Reservation(t, c, date, time, covers)
r : Reservation
saveReservation(r)
bks := getBookings(date)
updateDisplay()

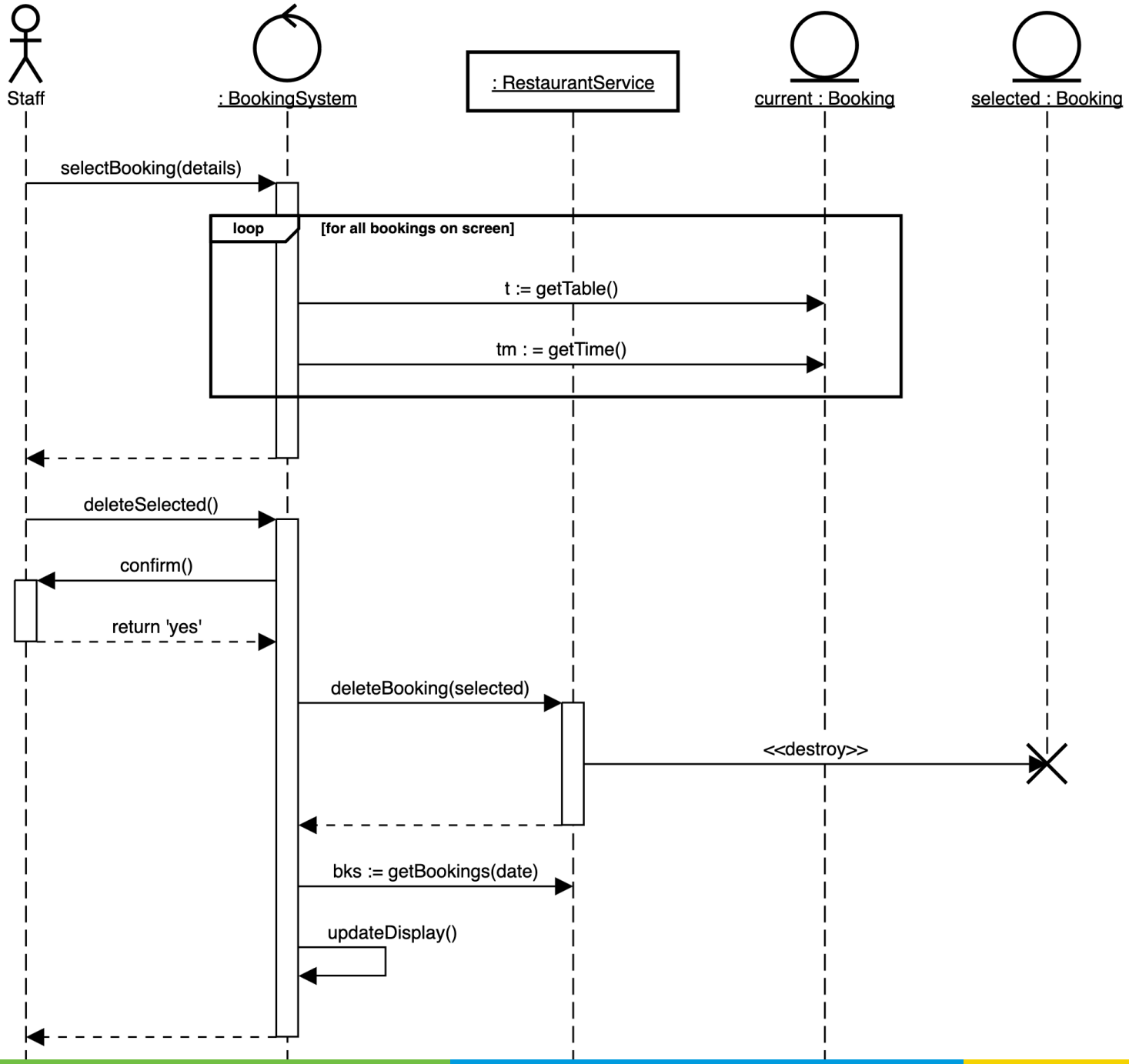

Staff
: BookingSystem
: RestaurantService
t : Table
makeWalkin(details)
createWalkin(details)
t := getTable(tno)
WalkIn(t, date, time, covers)
w : WalkIn
saveWalkin(w)
bks := getBookings(date)
updateDisplay()

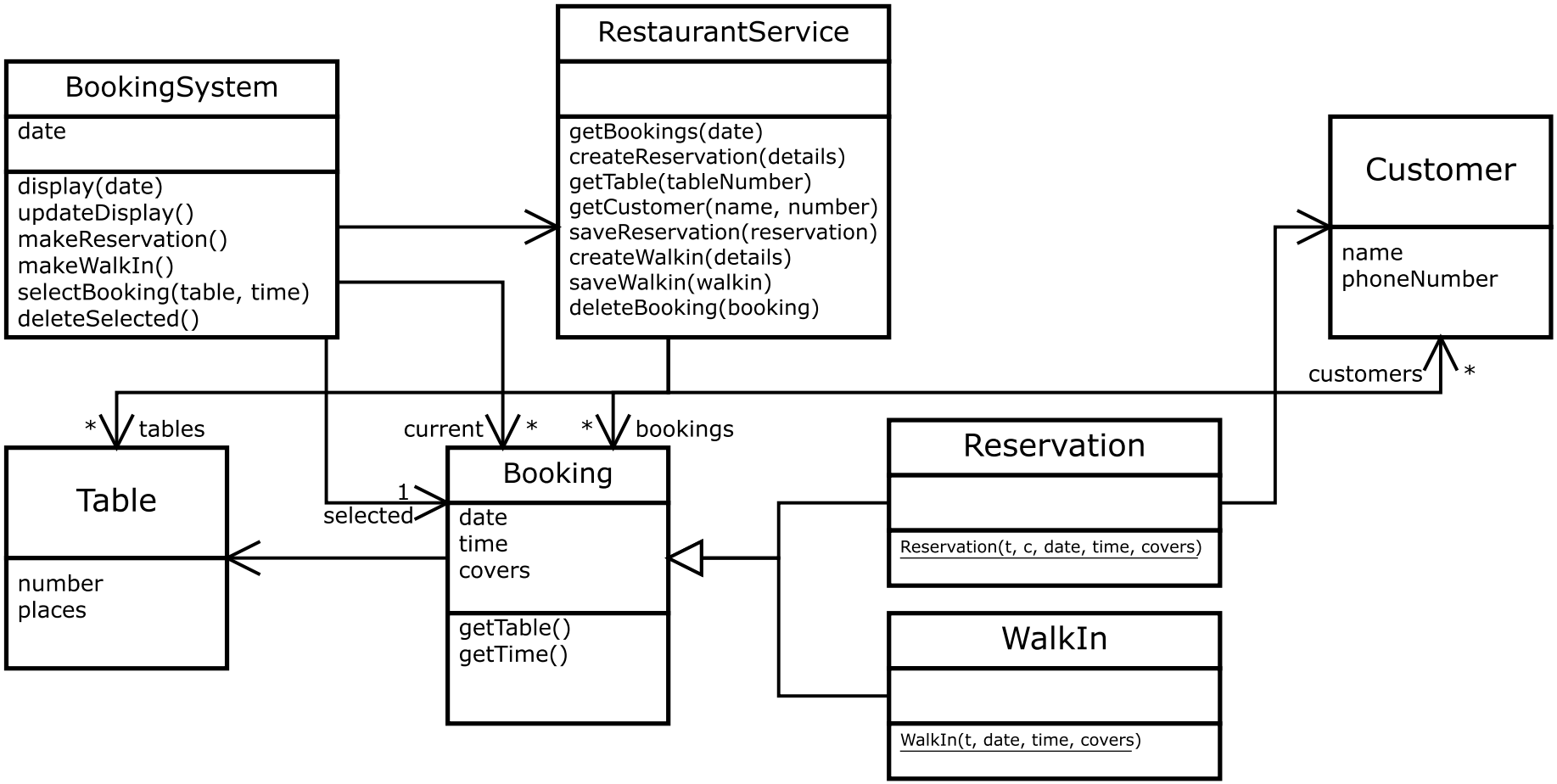

RestaurantService
getBookings(date)
createReservation(details)
getTable(tableNumber)
getCustomer(name, number)
saveReservation(reservation)
createWalkin(details)
saveWalkin(walkin)
BookingSystem
date
display(date)
updateDisplay()
makeReservation()
makeWalkIn()
Customer
name
phoneNumber
customers
*
*
tables
current
*
*
bookings
Table
number
places
Booking
date
time
covers
Reservation
Reservation(t, c, date, time, covers)
WalkIn
WalkIn(t, date, time, covers)

- The reservation object does not exist at the start of the use case
- This is why it is not shown at the top of the diagram
- The icon appears at the point it is created

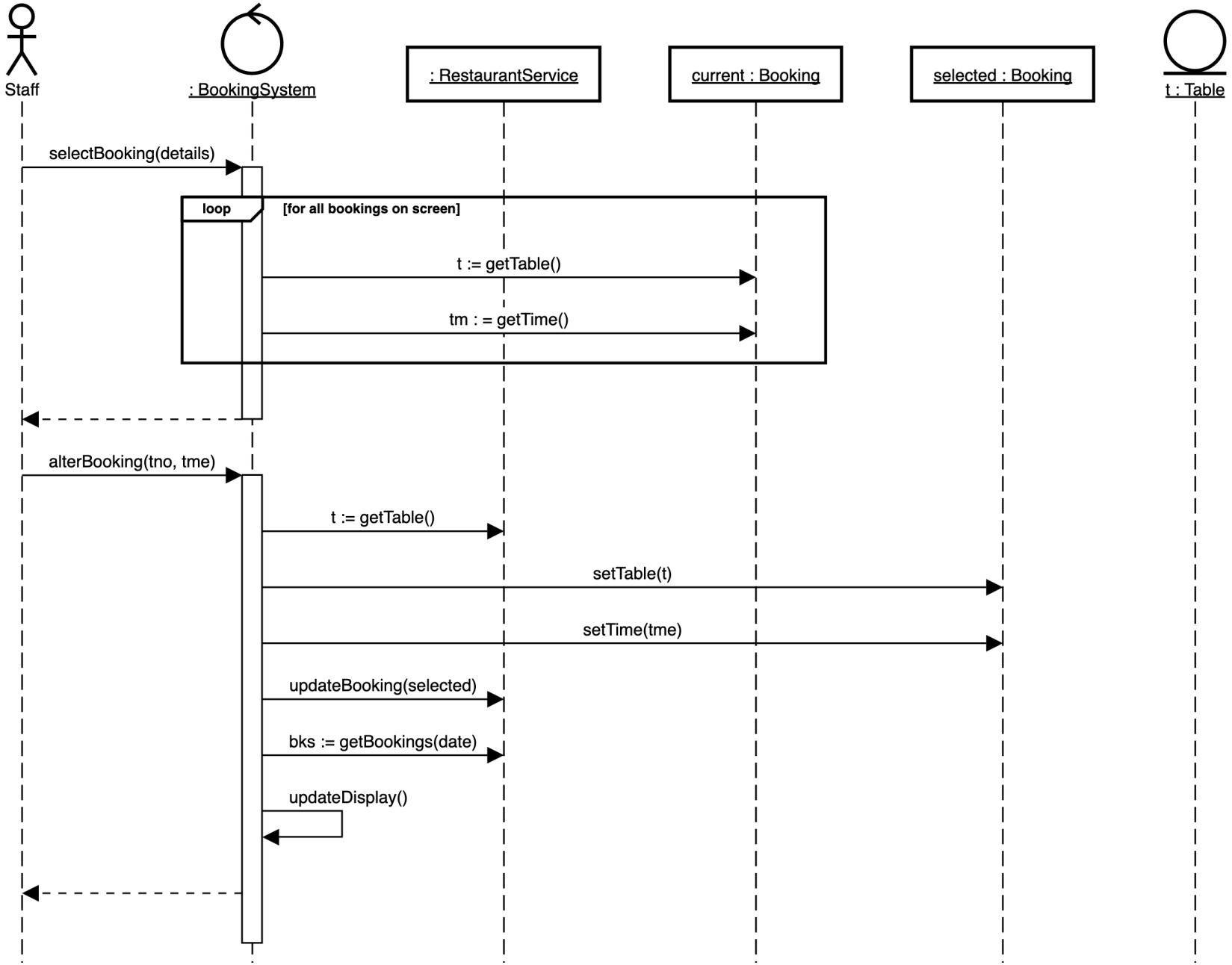
Section Contents

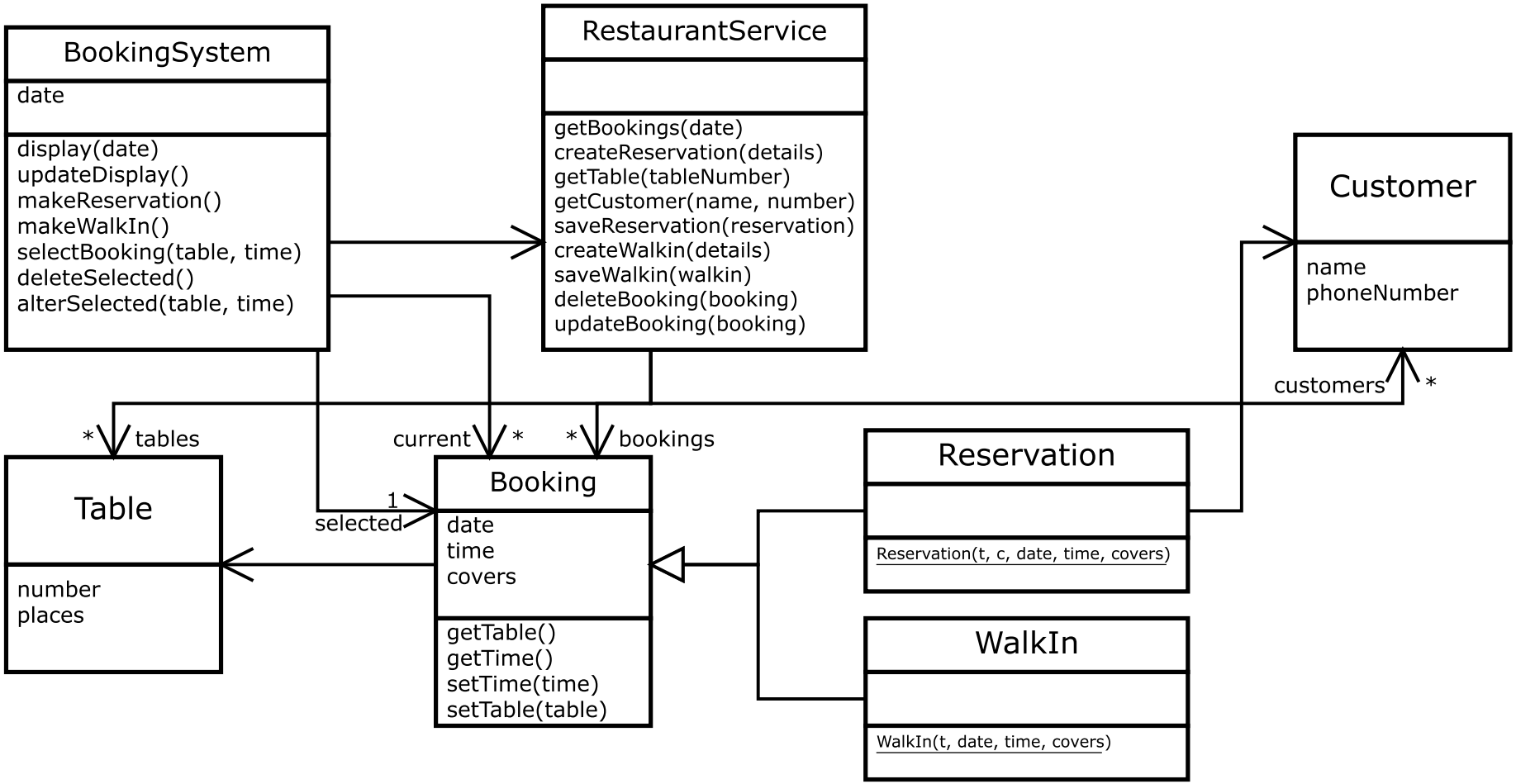
- To delete or update an individual booking we first have to select it
- This means that the use case will be in two stages
  - Finding the correct booking
  - Updating/changing/deleting it


Staff
: BookingSystem
: RestaurantService
current : Booking
selected : Booking
selectBooking(details)
loop
[for all bookings on screen]
t := getTable()
tm : = getTime()
deleteSelected()
confirm()
return 'yes'
deleteBooking(selected)
<<destroy>>
bks := getBookings(date)
updateDisplay()


BookingSystem
date
display(date)
updateDisplay()
makeReservation()
makeWalkIn()
selectBooking(table, time)
deleteSelected()
RestaurantService
getBookings(date)
createReservation(details)
getTable(tableNumber)
getCustomer(name, number)
saveReservation(reservation)
createWalkin(details)
saveWalkin(walkin)
deleteBooking(booking)
Customer
name
phoneNumber
customers
*
*
tables
current
*
*
bookings
Table
number
places
1
selected
Booking
date
time
covers
getTable()
getTime()
Reservation
Reservation(t, c, date, time, covers)
WalkIn
WalkIn(t, date, time, covers)

Section Contents


Staff
: BookingSystem
: RestaurantService
current : Booking
selected : Booking
t : Table
selectBooking(details)
loop
[for all bookings on screen]
t := getTable()
tm : = getTime()
alterBooking(tno, tme)
t := getTable()
setTable(t)
setTime(tme)
updateBooking(selected)
bks := getBookings(date)
updateDisplay()


BookingSystem
date
display(date)
updateDisplay()
makeReservation()
makeWalkIn()
selectBooking(table, time)
deleteSelected()
alterSelected(table, time)
RestaurantService
getBookings(date)
createReservation(details)
getTable(tableNumber)
getCustomer(name, number)
saveReservation(reservation)
createWalkin(details)
saveWalkin(walkin)
deleteBooking(booking)
updateBooking(booking)
Customer
name
phoneNumber
customers
*
*
tables
current
*
*
bookings
Table
number
places
1
selected
Booking
date
time
covers
getTable()
getTime()
setTime(time)
setTable(table)
Reservation
Reservation(t, c, date, time, covers)
WalkIn
WalkIn(t, date, time, covers)

Section Contents

- We have only realised the basic course of events for each use case
- We should complete the process by realising the alternate and exceptional courses of events
  - This will contain steps we need to consider
- It is usually preferable to complete these as separate diagrams rather than show the alternative flows on a single diagram