myTaxiService

Design Document

Outline

- Problem
- Architecture of the system
    - Components
    - Deployment
    - Runtime
    - Styles
- Some algorithms

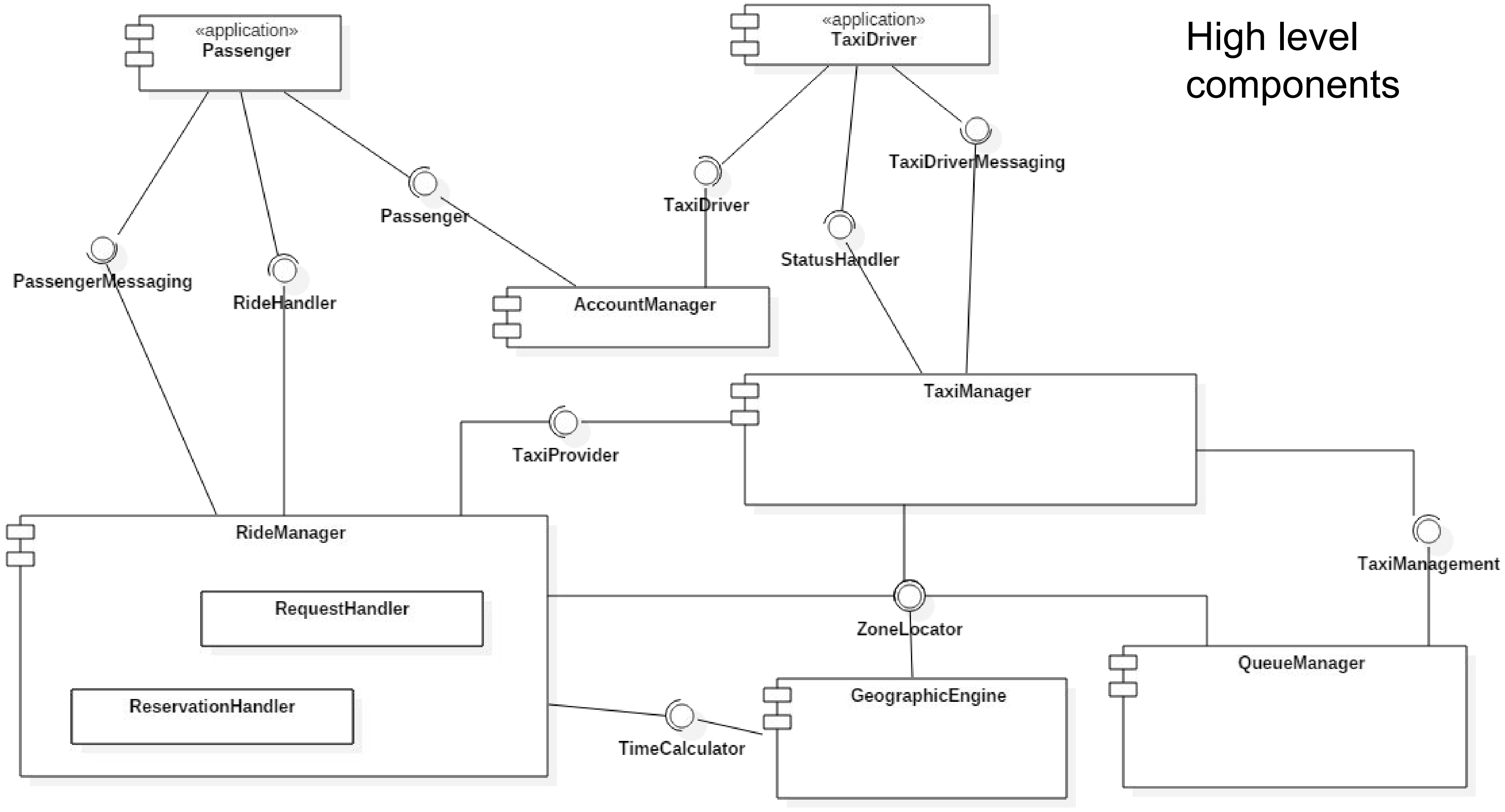
Problem

Let’s summarize:

- Application that permits passengers to call taxi drivers
  - (Immediate) requests and reservations possible
- Managing of taxi queues
  - Ordering policy

High level view

High level components



Component view

Account Manager

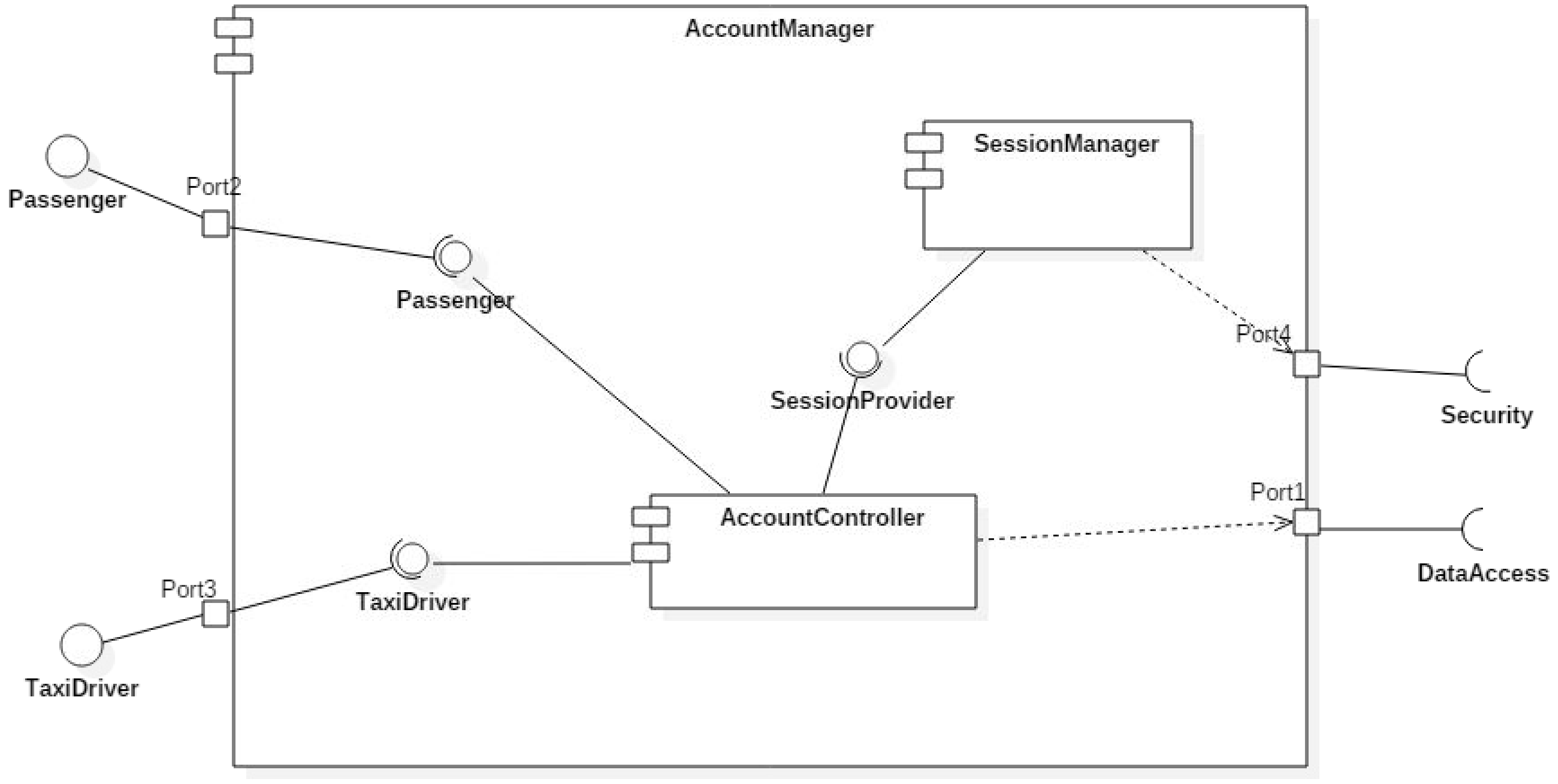
Passenger

- Passenger login(username, password)
  - Given valid credentials, allows the passenger to receive a session and to login.
- void logout(passenger)
  - Given a valid session, it deletes it.
- boolean register(username, password)
  - It creates a new account for a passenger with the credentials provided.
- boolean deleteAccount(username, password)
  - Given valid credentials, it deletes the account associated.

Account Manager

TaxiDriver

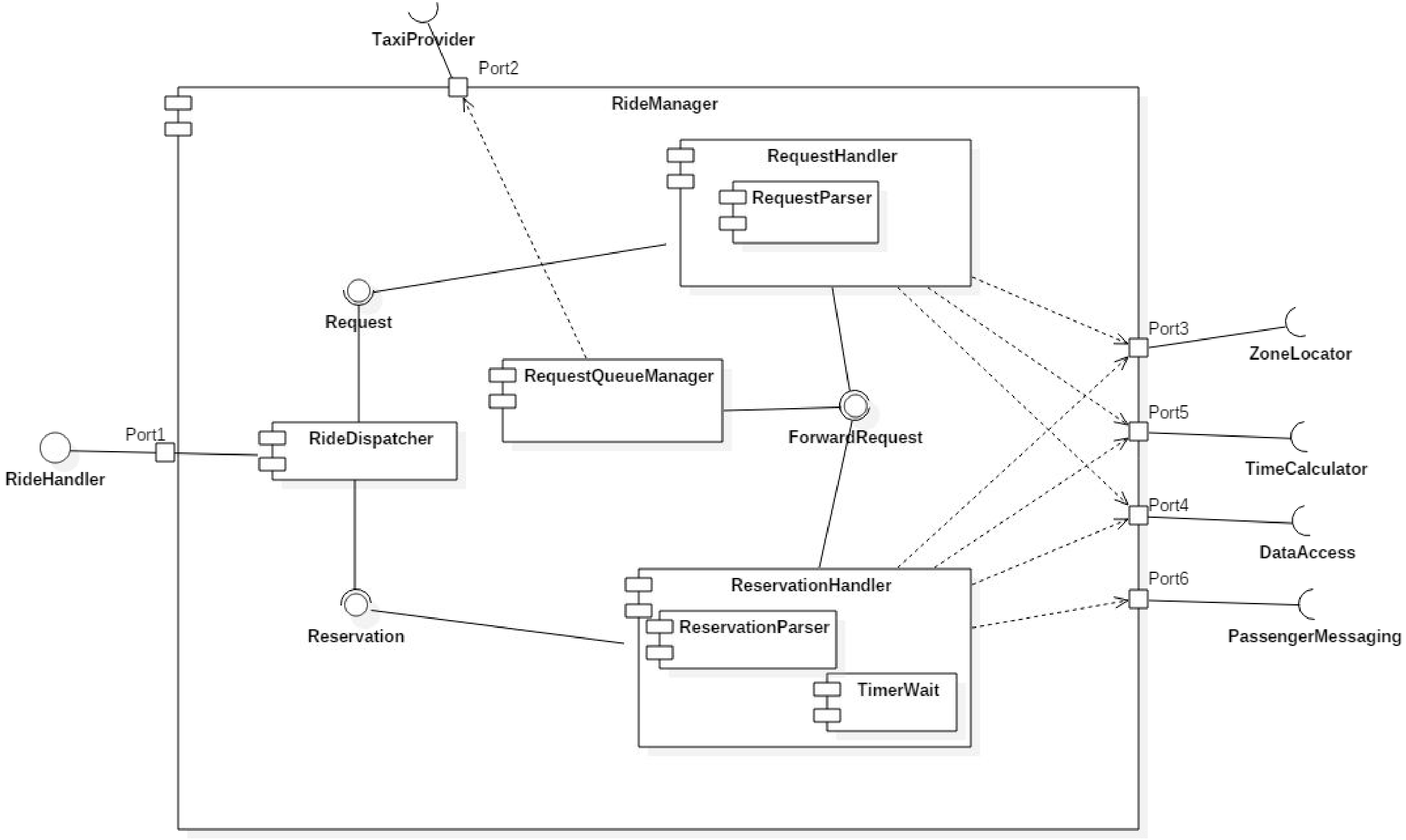
- TaxiDriver login(username, password)
  - Given valid credentials, allows the taxi driver to receive a session and to login.
- void logout(TaxiDriver taxiDriver)
  - Given a valid session, it deletes it.


AccountManager
SessionManager
Passenger
Port2
Passenger
SessionProvider
Port4
Security
Port1
AccountController
DataAccess
Port3
TaxiDriver
TaxiDriver

Ride Manager

RideHandler

- Response makeRequest(location, numberOfPassenger, Passenger passenger)
    - It starts the activity of making a request. It returns either informations about the taxi driver who is coming, or an invitation to try later.
- Response makeReservation(source, destination, numOfPass, date, time, Passenger passenger)
    - It starts the activity of making a reservation. If the time of the reservation is at least 2 hours later, it return a positive message.
- List<Reservation> getReservations(Passenger passenger)
    - It returns all the reservations of a given passenger.
- boolean deleteReservation(Passenger passenger, Reservation reservation)
    - It deletes a reservation of a passenger.


TaxiProvider
Port2
RideManager
RequestHandler
RequestParser
Request
RequestQueueManager
ForwardRequest
Port1
RideHandler
RideDispatcher
Reservation
ReservationHandler
ReservationParser
TimerWait
Port3
ZoneLocator
Port5
TimeCalculator
Port4
DataAccess
Port6
PassengerMessaging

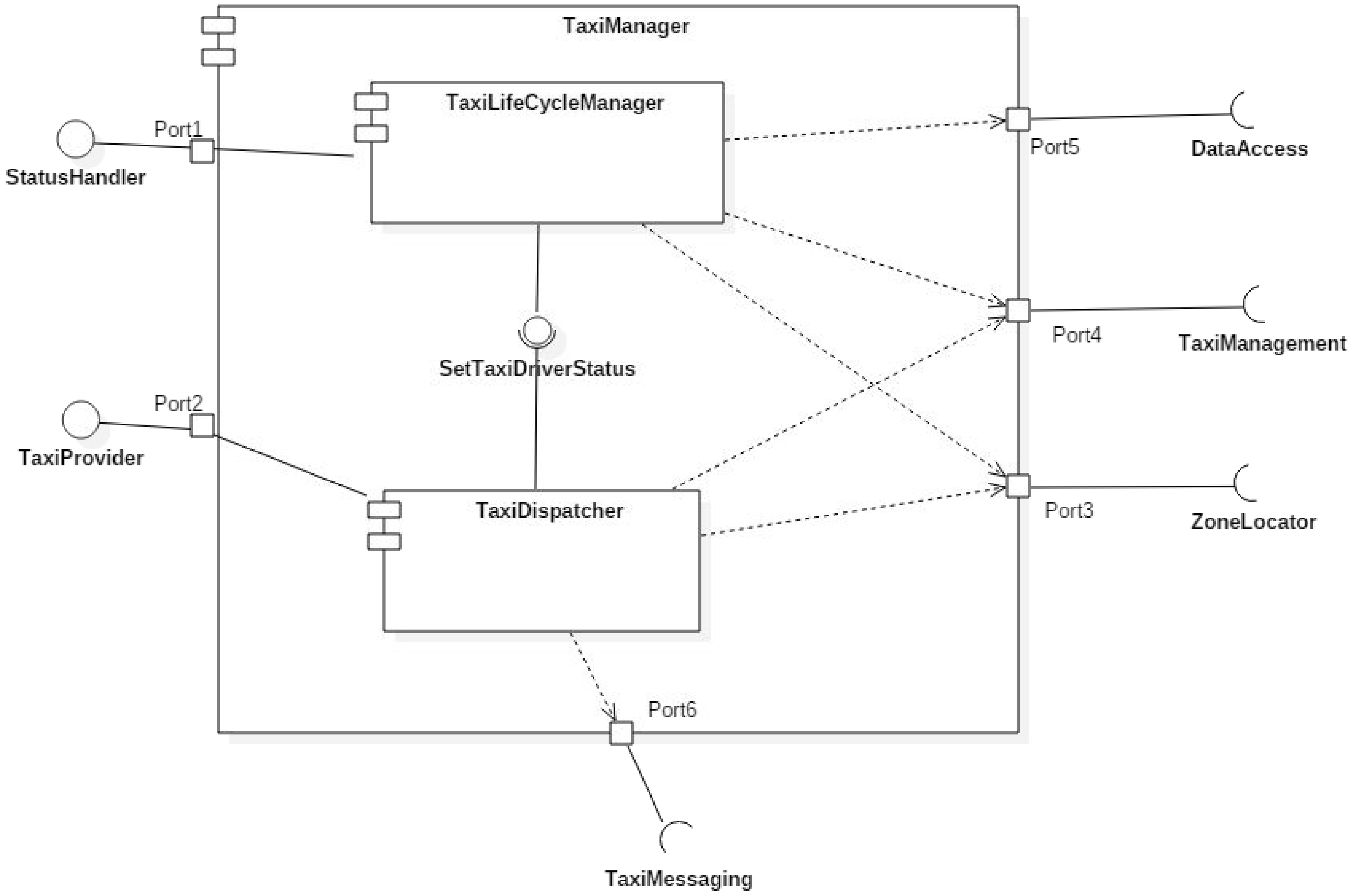
Taxi Manager

StatusHandler

- boolean setAvailabilityStatus(TaxiDriver taxiDriver, AvailabilityStatus status)
  - It assigns a new availability status to a taxi driver.
- void setLocation(TaxiDriver taxiDriver, Location location)
  - It updates the location of a taxi driver.

TaxiProvider

- Response provideTaxi(Location location)
  - Given a valid location, if there is at least one available taxi driver, it returns it.


TaxiManager
TaxiLifeCycleManager
Port1
StatusHandler
Port5
DataAccess
SetTaxiDriverStatus
Port4
TaxiManagement
Port2
TaxiProvider
TaxiDispatcher
Port3
ZoneLocator
Port6
TaxiMessaging

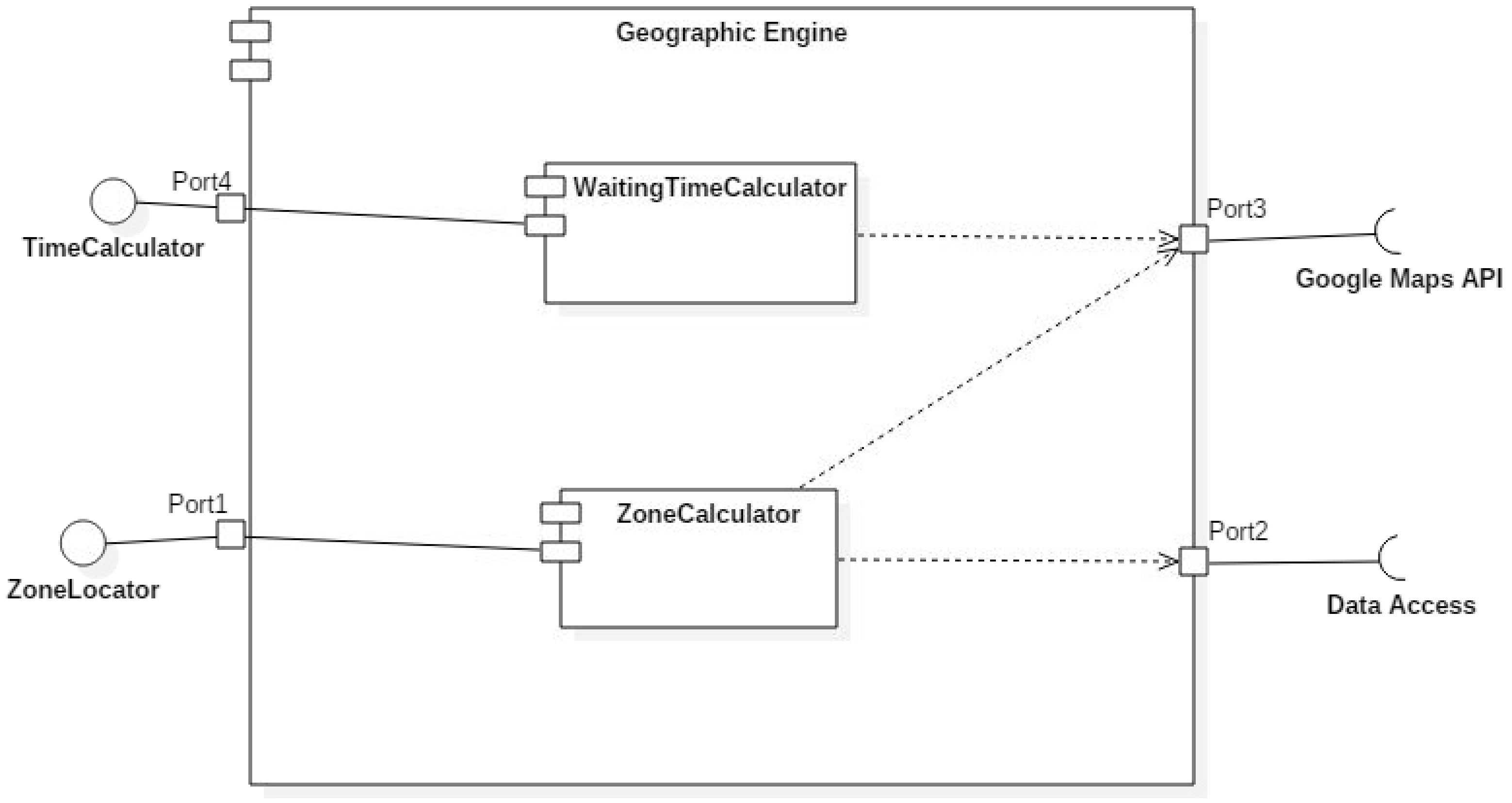
Geographic Engine

ZoneLocator

- Zone getTaxiZone(Location location)
    - It computes the corresponding zone of a given location.
- List<Zone> getTaxiZones()
    - It returns all the zones.
- boolean isValidLocation(Location location)
    - Given a location, it checks whether it is valid or not.

TimeCalculator

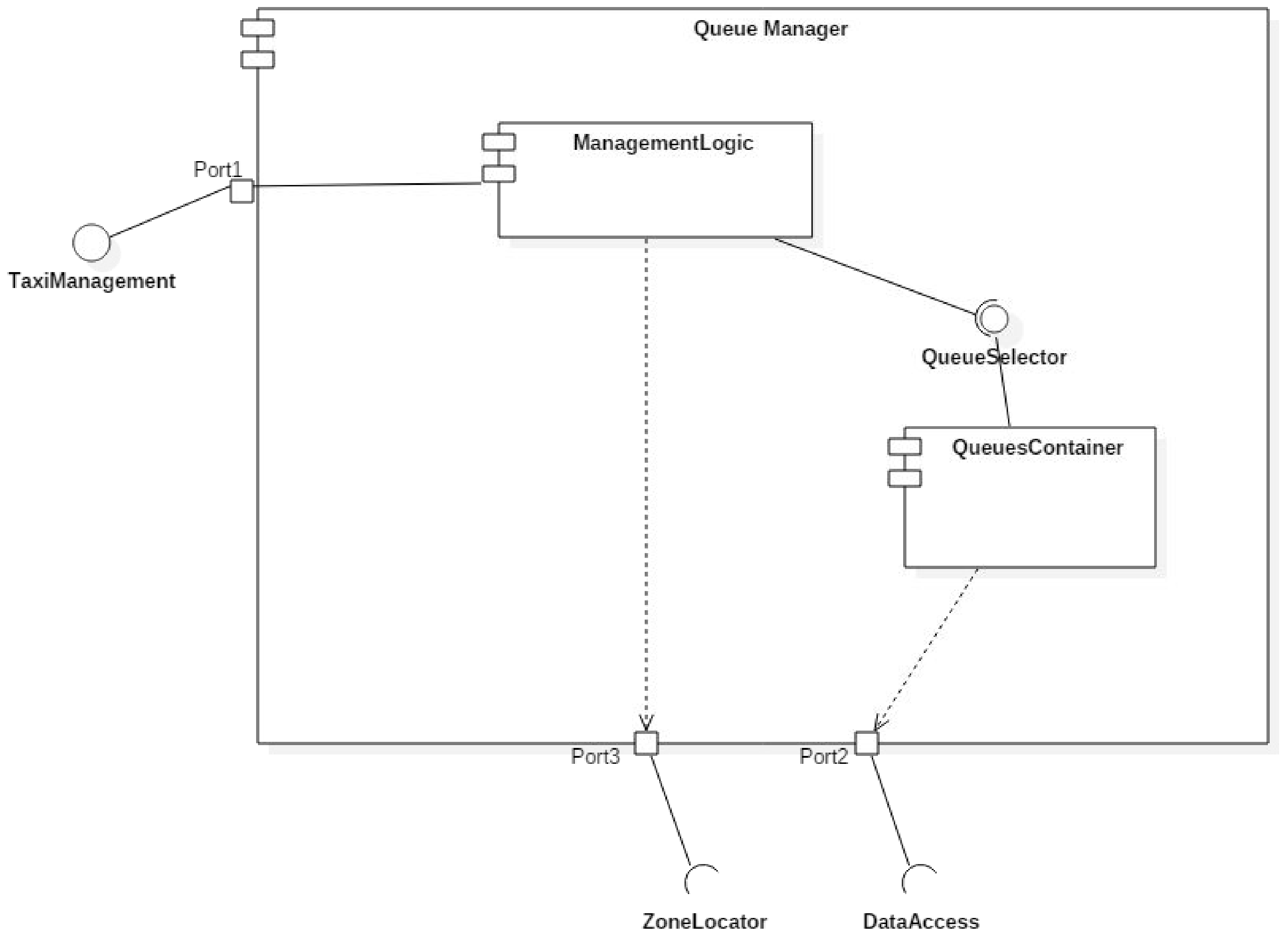
- Time getTimeEstimation(Location source, Location, destination)
    - Given valid source and valid location, it returns an approximative traveling time.


Geographic Engine
Port4
TimeCalculator
WaitingTimeCalculator
Port3
Google Maps API
Port1
ZoneLocator
ZoneCalculator
Port2
Data Access

Queue Manager

TaxiManagement

- void addTaxi(TaxiDriver taxi, Zone zone)
    - Given a taxi driver and a zone, it adds the taxi driver to the queue corresponding to the zone.
- TaxiDriver getTaxi(Zone zone)
    - It returns the rst available taxi from the queue corresponding of the given zone.
- void removeTaxi(TaxiDriver taxi)
    - Given a taxi driver, it removes it from the queue he is in.
- void changeZone(TaxiDriver taxi, Zone zone)
    - Given a taxi driver and a zone, it remove the taxi driver from the queue he is in, and add it in the queue corresponding to the zone.

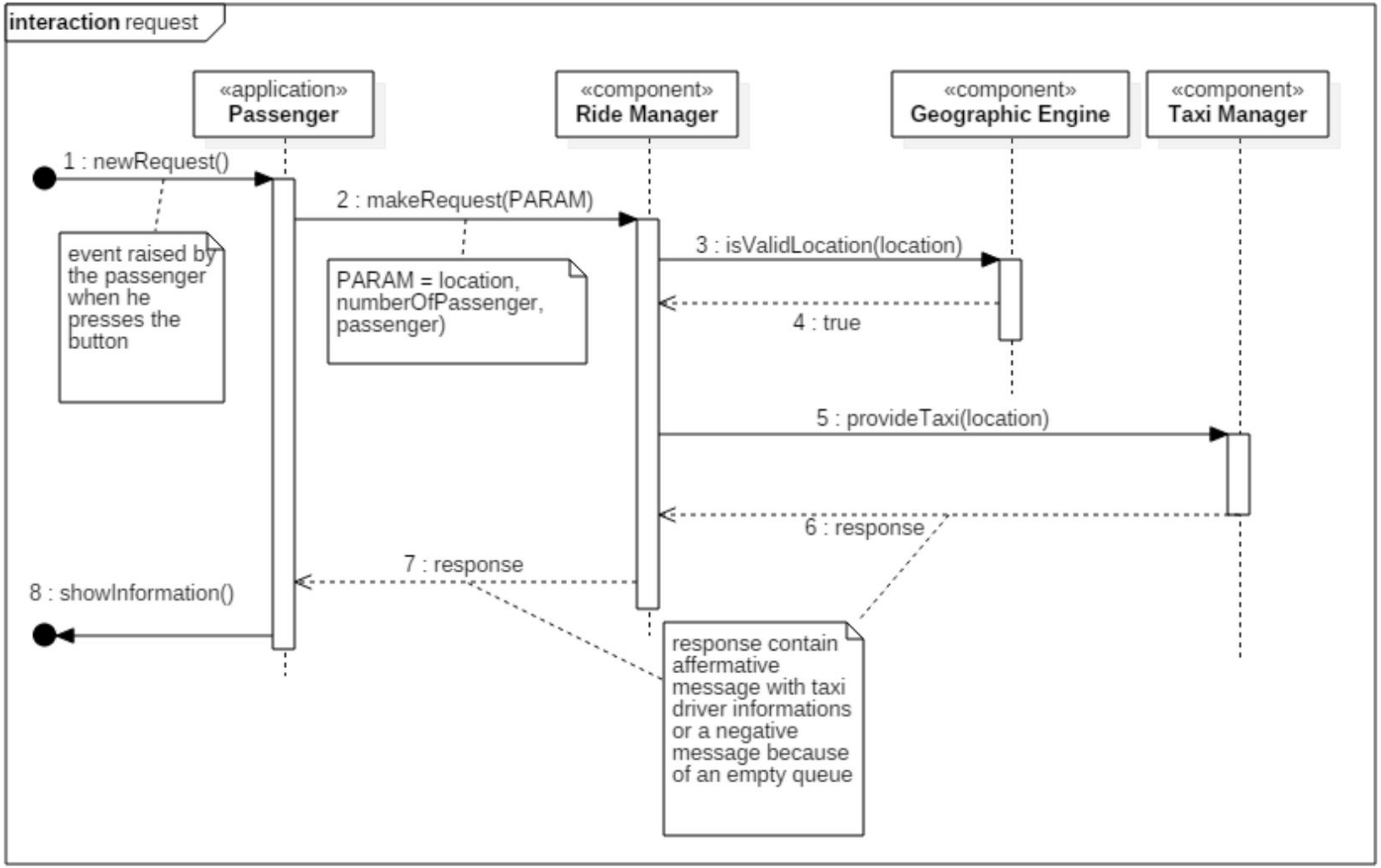

Queue Manager
ManagementLogic
Port1
TaxiManagement
QueueSelector
QueuesContainer
Port3
Port2
ZoneLocator
DataAccess

Runtime

NB: we consider as *requests* the calls for an immediate service and *reservations* the booking of a taxi

Passenger makes a *request*

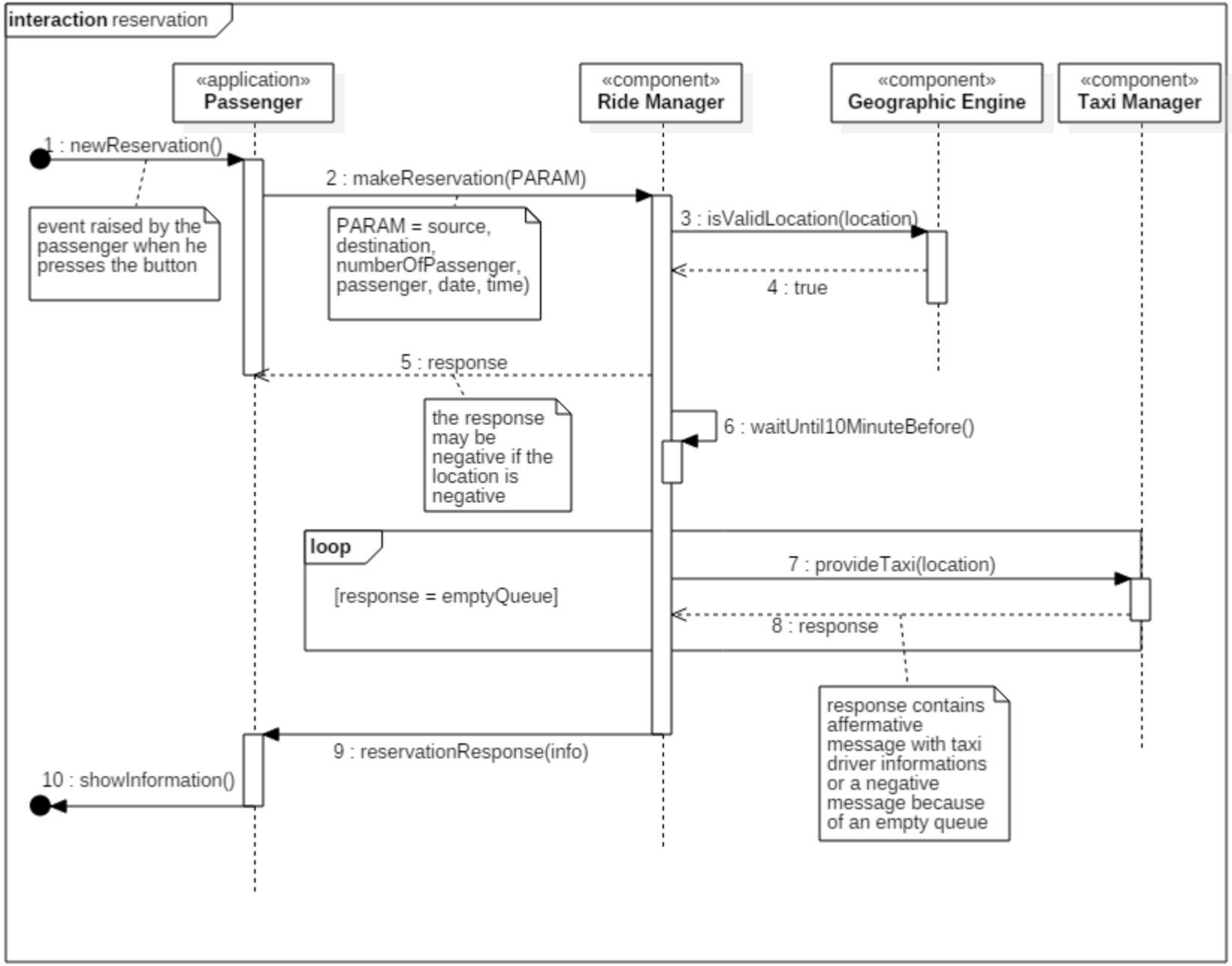
- Passenger (application):
  - Submits the request through the relative form
- Ride Handler
  - Checks if the provided location is valid and calls the taxi manager for a taxi
- Geographic engine
  - Performs the mathematical operations for checking the correctness of the location
- Taxi Manager
  - It provides the taxi for the specified location

interaction request
«application»
Passenger
«component»
Ride Manager
«component»
Geographic Engine
«component»
Taxi Manager
1 : newRequest()
event raised by the passenger when he presses the button
2 : makeRequest(PARAM)
PARAM = location, numberOfPassenger, passenger)
3 : isValidLocation(location)
4 : true
5 : provideTaxi(location)
6 : response
7 : response
8 : showInformation()
response contain affermative message with taxi driver informations or a negative message because of an empty queue

Runtime

Passenger makes a *reservation*

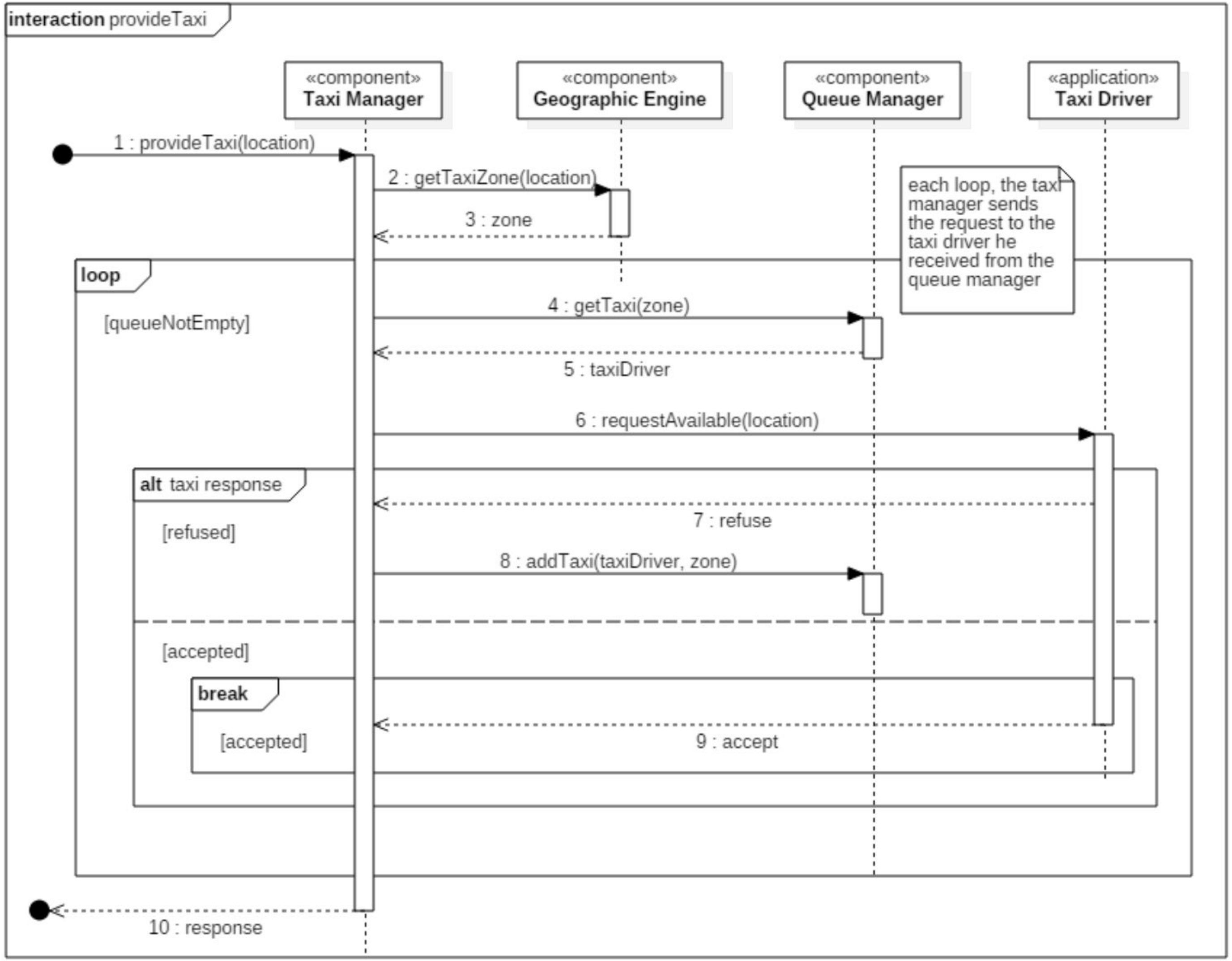
- Passenger (application):
    - Submits the reservation through a form
- Ride Handler:
    - Checks the correctness of the location and communicate to the passenger’s application the outcome of the process. It then waits until 10 minutes before the specified meeting time. When it is time calls the taxi manager *until* there is at least a taxi available for that reservation.
- Geographic engine:
    - Checks the correctness of the provided location
- Taxi Manager:
    - if available, provides a taxi for the specified location


interaction reservation
«application» Passenger
«component» Ride Manager
«component» Geographic Engine
«component» Taxi Manager
1 : newReservation()
event raised by the passenger when he presses the button
2 : makeReservation(PARAM)
PARAM = source, destination, numberOfPassenger, passenger, date, time)
3 : isValidLocation(location)
4 : true
5 : response
the response may be negative if the location is negative
6 : waitUntil10MinuteBefore()
loop
[response = emptyQueue]
7 : provideTaxi(location)
8 : response
response contains affermative message with taxi driver informations or a negative message because of an empty queue
9 : reservationResponse(info)
10 : showInformation()

Runtime

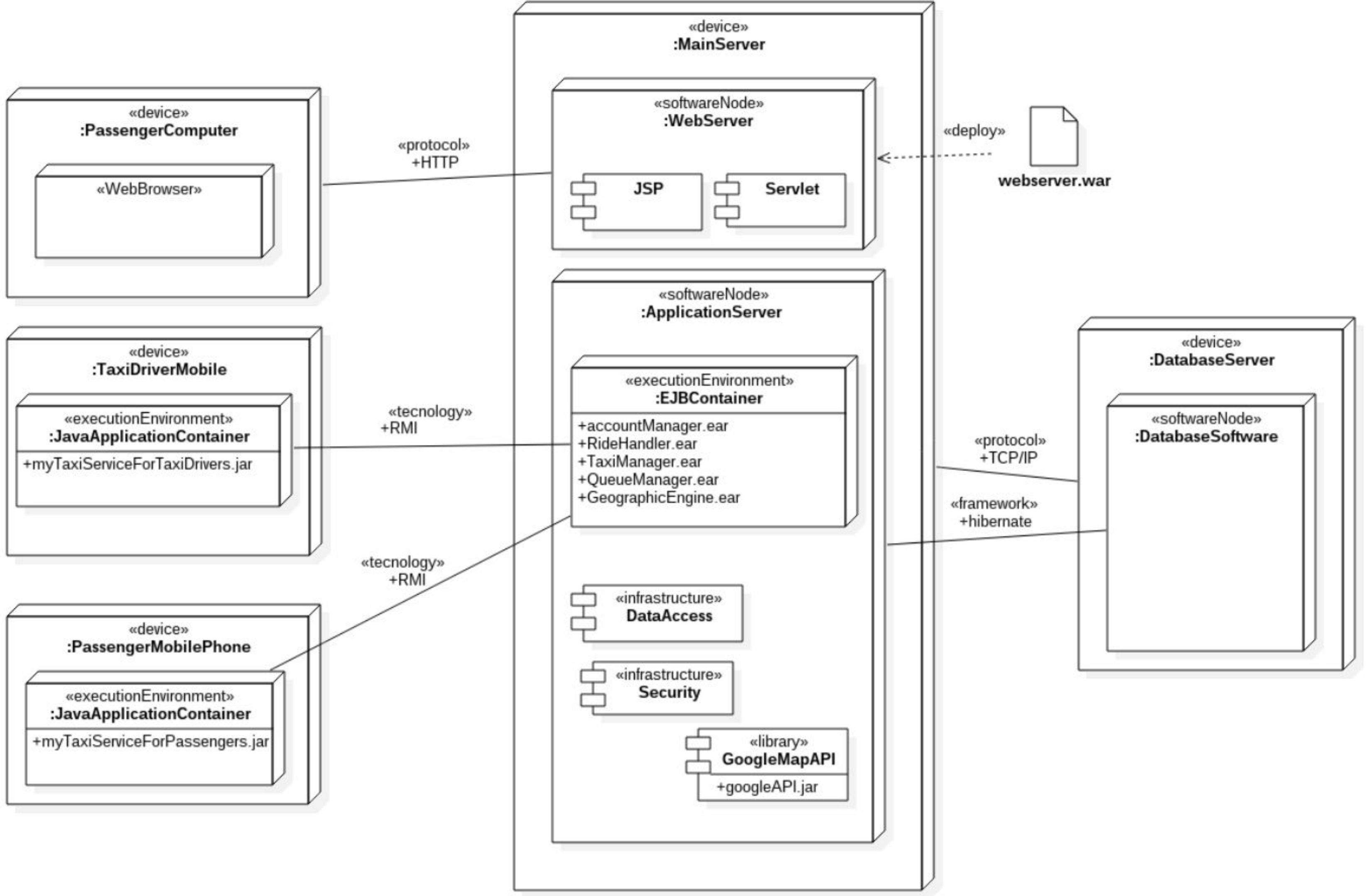
Provision of a taxi

- Taxi Manager:
  - Gets the zone relative to the location. *Until* he finds a taxi driver who accepts the request / reservation, or the queue relative to the zone is empty, it asks to the taxi driver provided by the queue manager.
- Taxi driver (application):
  - The application receives an offer of a request / reservation, it shows up a pop-up and ask the taxi driver to choose what he want to do
- Geographic engine
  - it retrieves the corresponding zone of a provided location
- Queue manager
  - It returns the *first* taxi driver in the queue of the provided zone. If the queue is empty it returns a negative response


interaction provideTaxi
«component» Taxi Manager
«component» Geographic Engine
«component» Queue Manager
«application» Taxi Driver
1 : provideTaxi(location)
2 : getTaxiZone(location)
3 : zone
loop
[queueNotEmpty]
each loop, the taxi manager sends the request to the taxi driver he received from the queue manager
4 : getTaxi(zone)
5 : taxiDriver
6 : requestAvailable(location)
alt taxi response
[refused]
7 : refuse
8 : addTaxi(taxiDriver, zone)
[accepted]
break
[accepted]
9 : accept
10 : response

Deployment

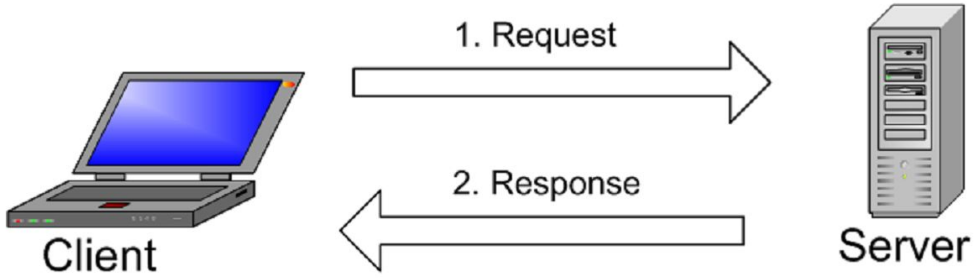
- Thin clients
  - Mobile applications interact directly with the Application server - TCP/IP - RMI calls
  - Browser clients must interact with the web server (Servlet - JSP) - HTTP
- One “machine” with both *Web server* and *Application server*
  - Easy and cheap
- Data layer separated from logic layer: database(s)
  - Accessed by logic layer through infrastructural libraries: *hibernate*
- Clouding possibility
  - Increased reliability
  - Security
  - Division of responsibility


«device»
:MainServer
«softwareNode»
:WebServer
JSP
Servlet
«deploy»
webserver.war
«device»
:PassengerComputer
«WebBrowser»
«protocol»
+HTTP
«softwareNode»
:ApplicationServer
«executionEnvironment»
:EJBContainer
+accountManager.ear
+RideHandler.ear
+TaxiManager.ear
+QueueManager.ear
+GeographicEngine.ear
«device»
:TaxiDriverMobile
«executionEnvironment»
:JavaApplicationContainer
+myTaxiServiceForTaxiDrivers.jar
«tecnology»
+RMI
«tecnology»
+RMI
«device»
:PassengerMobilePhone
«executionEnvironment»
:JavaApplicationContainer
+myTaxiServiceForPassengers.jar
«infrastructure»
DataAccess
«infrastructure»
Security
«library»
GoogleMapAPI
+googleAPI.jar
«device»
:DatabaseServer
«softwareNode»
:DatabaseSoftware
«protocol»
+TCP/IP
«framework»
+hibernate

Architectural styles

Client - server (of course):

- centrality of the server and sparsity of the clients
- absence of business logic client side



- Thin clients
  - Power consumption awareness
- Easily modifiable client applications
- Easy installation of mobile app

Let’s be more precise...

Architectural styles

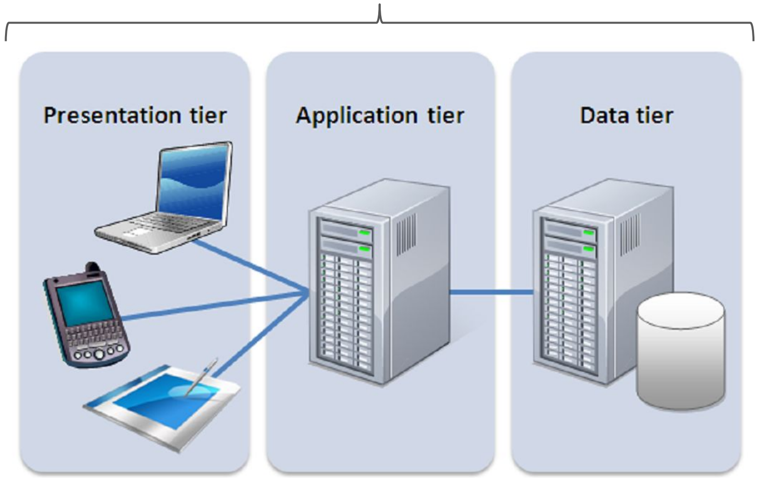
Three-tier-architecture

- Presentation
  - graphic rendering, forms, buttons, ecc...
- Application
  - business logic
- Data
  - storage of informations

Complete *independence* and *modularity*:

- the presentation must interact first with the application in order to access data
- ≠ from MVC pattern

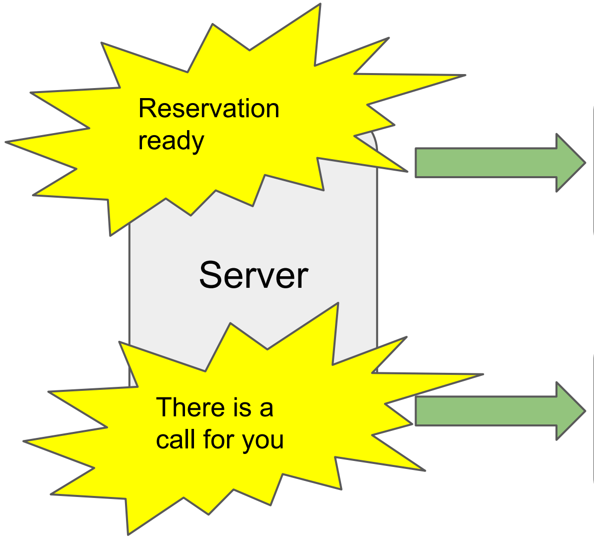
JEE architecture



- Easy scalability
- Well defined interfaces
- Division of responsibility
  - It is easier the implementation of the required *programmatic interface*
- Possibility of replication
  - Reliability

Architectural styles

Problem: ***notification to the clients*** (passengers and taxi drivers)



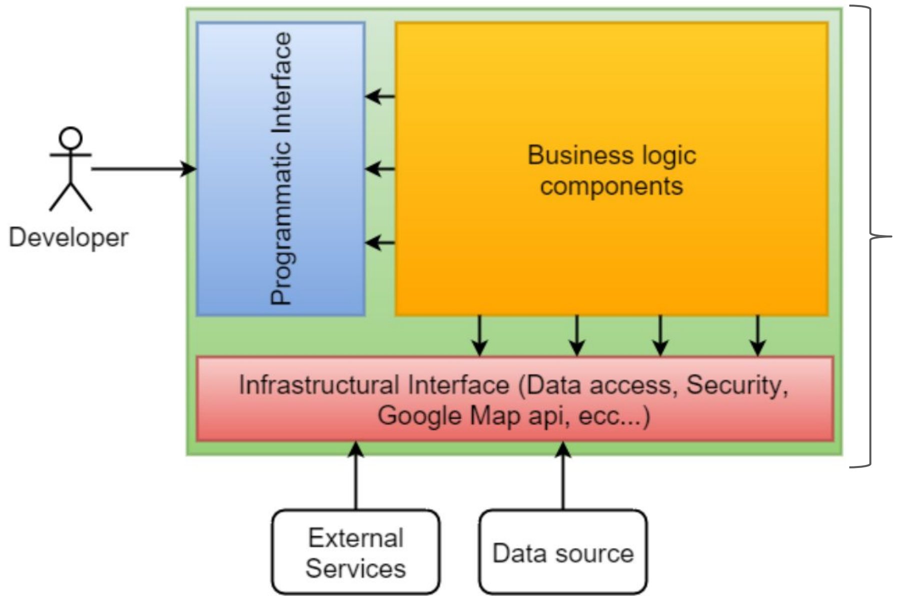
In contrast with the *client - server* paradigm.

It can be resolved *manually* through a **polling** policy client side:

- it must be optimize in order to reduce band consumption and server load
- there are ready-made framework dedicated to this problem: JMS ☺
- variant of the *publish-subscribe* style

Other design choices

- Subdivision into sub-components: D&C
- Depend upon interfaces: DIP
    - In order to implement the *programmatic interface*


Developer
Programmatic Interface
Business logic components
Infrastructural Interface (Data access, Security, Google Map api, ecc...)
External Services
Data source

The programmatic interface is exposed to the developers in order to build new features on the top of the provided business logic.

We use the already defined interfaces

Algorithm Design - Reservation Handler

Problem:

- Reservations come at any time
- Reservations have to be forwarded only 10 minutes before the meeting time

Solution:

- Three threads: Receiver, Waiter, Forwarder
- Keep the reservations ordered by meeting time
- Sleep until 10 minute before the meeting time of the first in the data structure
- When a new reservation comes, insert it in the data structure and check if it is the new first element.

Algorithm Design - Reservation Handler

RECEIVER

```
reservation_received
put(reservation) //ordered list
if index = first
 waitingTime=meetingTime - now
 stopCurrentTimer
 setTimer(waitingTime)
```

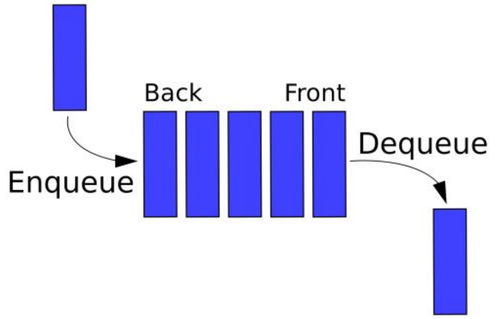
WAITER

```
timer_set
wait(waitingTime)

timer_expires
wakeUpForwarder
```

FORWARDER

```
woken_up
getFirstReservation
getNextReservationMeetingTime
waitingTime = meetingTime-now
stopCurrentTimer
setTimer(waitingTime)
forwardFirstReservation
```


Back
Front
Enqueue
Dequeue