POLITECNICO
MILANO 1863

A.Y. 2020/2021

Computer Science and Engineering

Software Engineering 2 Project

CLup - Customers Line-up

Design Document

-	-
-	-
-	-

Deliverable:	DD
Title:	CLup - Requirement Analysis and Verification Document
Authors:	
Version:	1.0
Date:	2021-01-03
Download page:	https://github.com/rb-sl/SoftEng2Project2020
Copyright:	© 2021 – All rights reserved

Contents

List of Figures

1 Introduction

1.1 Purpose

This document represents the DD (Design Document) of the CLup application; it describes CLup's architecture and integrates with the RASD. Furthermore, it describes its components, how they interact, and what is the plan for implementation, integration and testing.

1.2 Scope

1.2.1 Problem description

CLup is a software-to-be that will help stores and customers preventing dangerous crowds during shopping due to the current Covid-19 Pandemic. This system should offer the possibility to line up from home for stores while still allowing to physically line up for stores. Moreover, it should offer the possibility to book a visit for a chosen store and get notifications about stores occupancy during the week. The system should also allow store managers to add their stores to the system and monitor entrances.

The application has to be easy-to-use, scalable and reliable.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Available store**: a store that can host other customers, based on closing time
- **Book a visit**: action of requesting entrance at a specific date and time (that will be guaranteed)
- **Delay window**: number of minutes a customer is allowed to enter after their turn is called (if using a ticket) or their entrance time is reached (if after a visit)
- **Get a ticket**: action of requesting the entrance as soon as possible
- **Historical data**: information about precedent store flows stored by date/time
- **Web mapping service**: a service external to the application (such as Google Maps)
- **Long term user**: for a store, an user who visited it for at least a number of times defined by the Store Managers
- Parameters:
  - **M**: the dimension (in minutes) of the delay window specified by the Store Manager
  - **P%**: parameter specified by the Store Manager that indicates the minimum percentage of places that is reserved to tickets
- **Reasonably available store**: a store for which waiting time is less than a user defined parameter
- **Spatial locality**: distance from the user inferior to a user defined parameter
- **Store information**: refers to store id, address, customer capacity, opening times, allowed delay of a customer, items/categories
- **User**: an user of the application. As per the user diagram, the term encompasses both e-Customers and Store Managers

1.3.2 Acronyms and Abbreviations

- **eC**: e-Customer
- **SM**: Store Manager
- **DD**: Design Document
- **RASD**: Requirement Analysis and Specification Document
- **[Gn]**: n-th goal

- **[Rn]**: n-th requirement
- **[Dn]**: n-th domain assumption
- Internet Protocols:
  - **HTTPS**: Hypertext Transfer Protocol Secure
  - **TLS**: Transport Layer Security
  - **REST**: REpresentational State Transfer

1.4 Revision history

Version	Date	Comment
1.0	2021-01-03	First release

1.5 Reference documents

R&DD Assignment AY 2020-2021, *Software Engineering 2's BeeP page*

1.6 Document structure

This document is composed of the following parts:

- **Section 1** is an introduction to the document and offers an overview of the document
- **Section 2** describes the application design, showing its components and explaining the design decisions
- **Section 3** shows the user interface diagrams of the application
- **Section 4** maps the requirements identified in the RASD to the components described in section 2
- **Section 5** suggests an implementation and test strategy
- **Section 6** reports the effort spent and the tools used to realize this document

2 Architectural design

2.1 Overview

2.1.1 High level architecture

Figure 1 shows what machines are going to be used for the system: most components are replicated to both increase performance (through the use of load balancers) and reliability, as one machine can take over the work of the other in case of fault. The logical layers are:

- Presentation: realized by the clients, either the application or the web browser
- Logic: covered by the web servers and the application servers
- Data: consists in the database servers and their DBs

The tiers of the system are illustrated more precisely in Section 2.3.

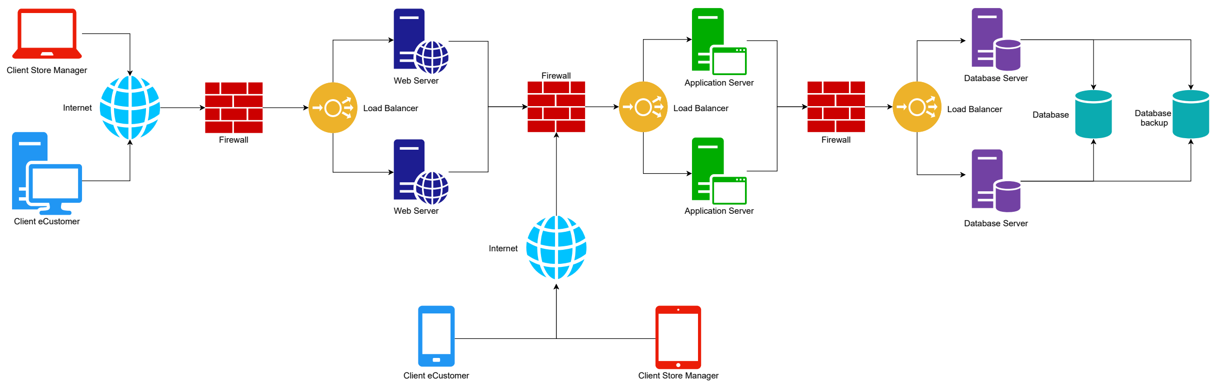


Figure 1: High level architecture of the system

2.1.2 Class diagram

A class diagram of the model of the application is visible in Figure 2; it is more detailed and presents only classes that will be developed with respect to the higher-level one shown in the RASD; it corresponds to the core system residing on the application server. Getter and setter methods have been omitted for space reasons, but they are present for every attribute except user passwords.

Classes relative to users represent the way they interact with the application; the e-Customer can therefore take all the actions described in the previous document, while the Store Manager (also through the Store class) can control every aspect of entrances and of their activity.

2.1.3 ER diagram

Figure 3 shows the ER model for the application's database. The tables are mapped to some classes in the class diagram (that adds some generalizations).

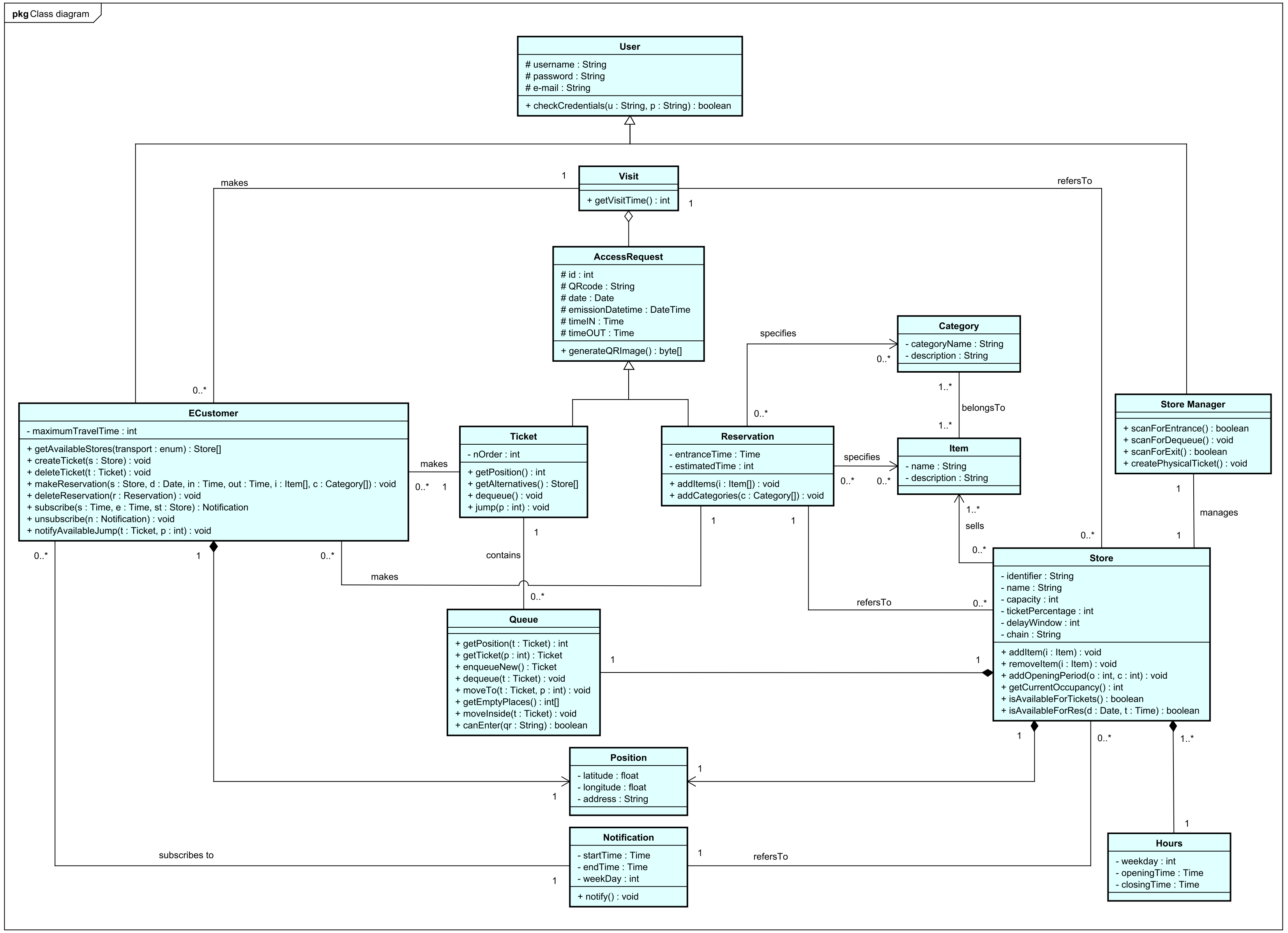

pkg Class diagram
User
# username : String
# password : String
# e-mail : String
+ checkCredentials(u : String, p : String) : boolean
Visit
+ getVisitTime() : int
makes
refersTo
AccessRequest
# id : int
# QRcode : String
# date : Date
# emissionDatetime : DateTime
# timeIN : Time
# timeOUT : Time
+ generateQRImage() : byte[]
Category
- categoryName : String
- description : String
specifies
belongsTo
Store Manager
+ scanForEntrance() : boolean
+ scanForDequeue() : void
+ scanForExit() : boolean
+ createPhysicalTicket() : void
ECustomer
- maximumTravelTime : int
+ getAvailableStores(transport : enum) : Store[]
+ createTicket(s : Store) : void
+ deleteTicket(t : Ticket) : void
+ makeReservation(s : Store, d : Date, in : Time, out : Time, i : Item[], c : Category[]) : void
+ deleteReservation(r : Reservation) : void
+ subscribe(s : Time, e : Time, st : Store) : Notification
+ unsubscribe(n : Notification) : void
+ notifyAvailableJump(t : Ticket, p : int) : void
Ticket
- nOrder : int
+ getPosition() : int
+ getAlternatives() : Store[]
+ dequeue() : void
+ jump(p : int) : void
Reservation
- entranceTime : Time
- estimatedTime : int
+ addItems(i : Item[]) : void
+ addCategories(c : Category[]) : void
Item
- name : String
- description : String
sells
manages
contains
Store
- identifier : String
- name : String
- capacity : int
- ticketPercentage : int
- delayWindow : int
- chain : String
+ addItem(i : Item) : void
+ removeItem(i : Item) : void
+ addOpeningPeriod(o : int, c : int) : void
+ getCurrentOccupancy() : int
+ isAvailableForTickets() : boolean
+ isAvailableForRes(d : Date, t : Time) : boolean
Queue
+ getPosition(t : Ticket) : int
+ getTicket(p : int) : Ticket
+ enqueueNew() : Ticket
+ dequeue(t : Ticket) : void
+ moveTo(t : Ticket, p : int) : void
+ getEmptyPlaces() : int[]
+ moveInside(t : Ticket) : void
+ canEnter(qr : String) : boolean
Position
- latitude : float
- longitude : float
- address : String
Notification
- startTime : Time
- endTime : Time
- weekDay : int
+ notify() : void
Hours
- weekday : int
- openingTime : Time
- closingTime : Time
subscribes to
refersTo
1
0..*
1..*

Figure 2: Class Diagram

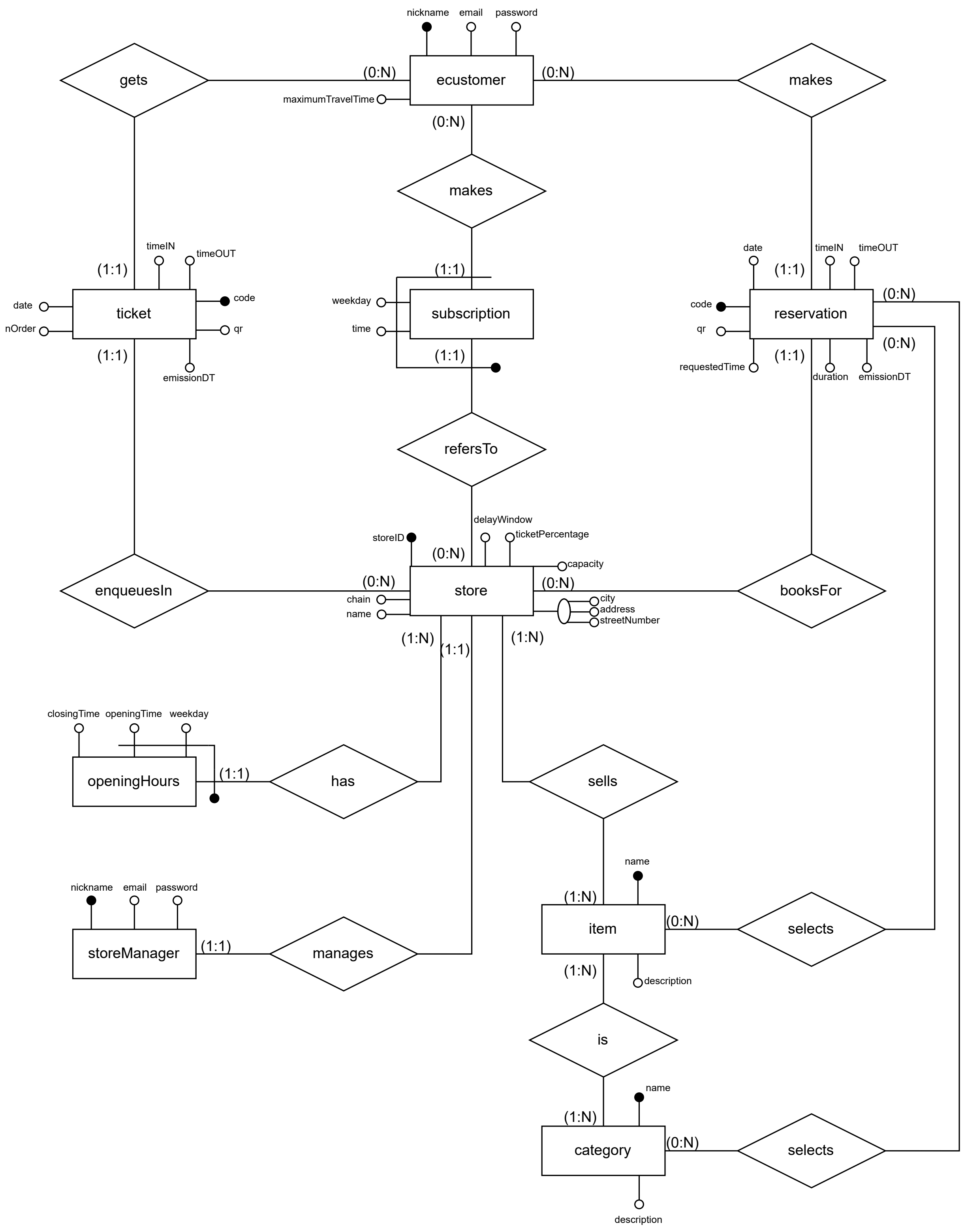


Figure 3: ER diagram of the database

2.2 Component view

This section illustrates the components of the application. Figure 4 offers an high-level representation of the web application's modules; to keep the scheme as clear as possible and avoid modules duplication (which can result misleading), the mobile application module has been omitted, as it connects exactly like the webapp's one. The next sections offer a greater detail on the subsystems, while the components shown here are:

- Maps Adapter: component used to interface the application with the mapping service's API
- Data Access Module: represents the system used to query the relational database

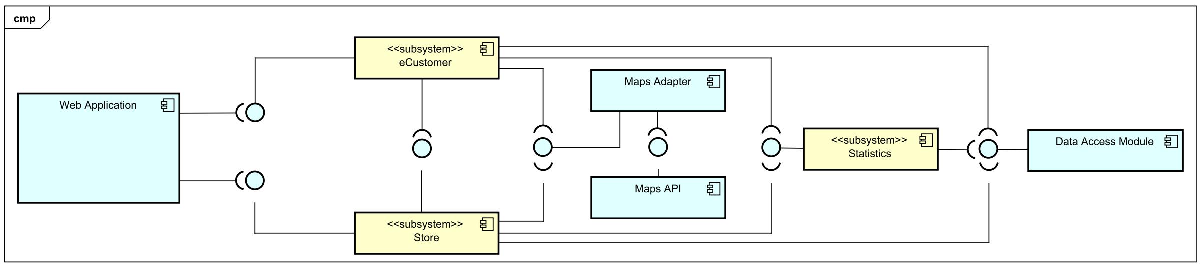


Figure 4: High-level component view

2.2.1 Store subsystem

The store subsystem modules, as shown in Figure 5, are the following:

- SM Account Manager Module: allows to modify information related to the Store Manager's account
- Store Info Manager Module: enables the insert and update of data related to the store (capacity, delay window, items...)
- Access Creation Module: controls the creation of reservations and tickets (including the printing of physical ones)
- Queue Manager Module: manages a queue of tickets, allowing all operations such as enqueue and dequeue
- Visit Manager Module: is responsible for creating visits, scheduling entrances, handle delays, scan codes and manage the customers inside

The connections from components to the Account Manager to check if the Store Manager is logged in are omitted to avoid cluttering, as many modules need to check if the user can access their functions.

2.2.2 e-Customer subsystem

The components that realize the e-Customer subsystem are shown in Figure 6. They are:

- eC Account Manager Module: allows the modification of the e-Customer's profile
- Store Recommender Module: manages the retrieval and proposal of stores to the e-Customer; considers location, travel time and store state to create its lists
- Access Request Module: is the module responsible for requesting the creation of tickets and reservations and handles the user interaction
- Requests Manager Module: controls the visualization and deletion of active tickets and reservations
- Notification Module: sends all notifications to the e-Customer when requested by other modules
- Subscription Module: handles the user's requests for slot notifications

Internal connections to the Account Manager used to check if the e-Customer is logged in are omitted.

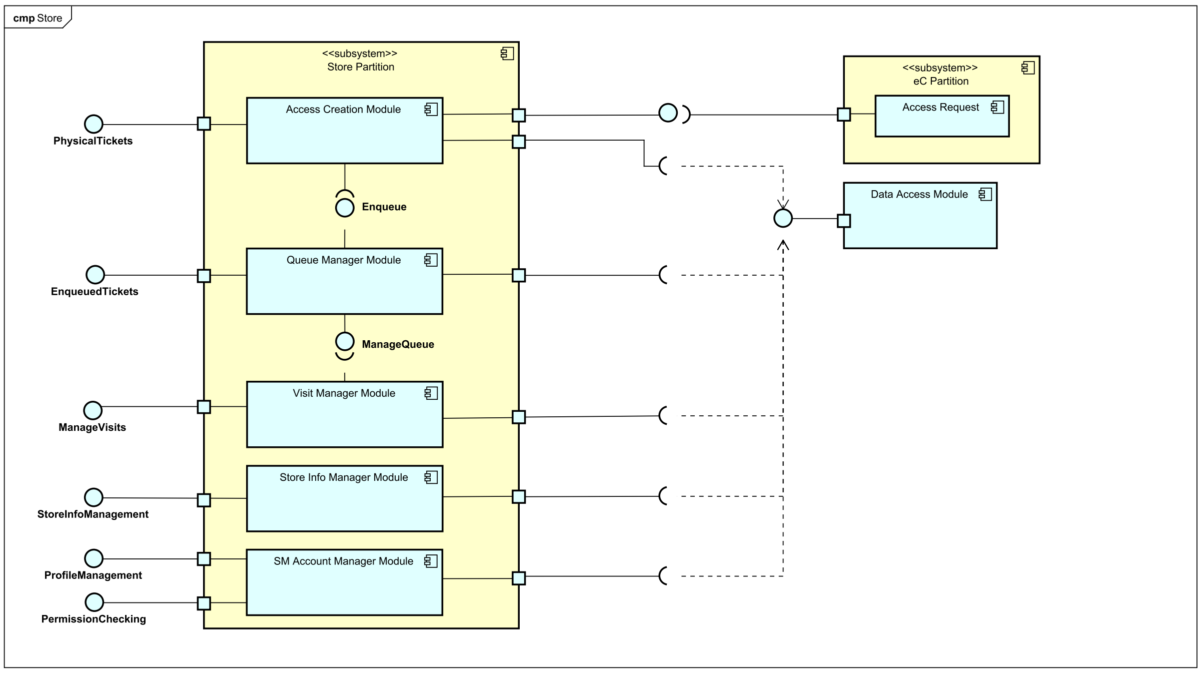


Figure 5: Component view of the Store Manager subsystem

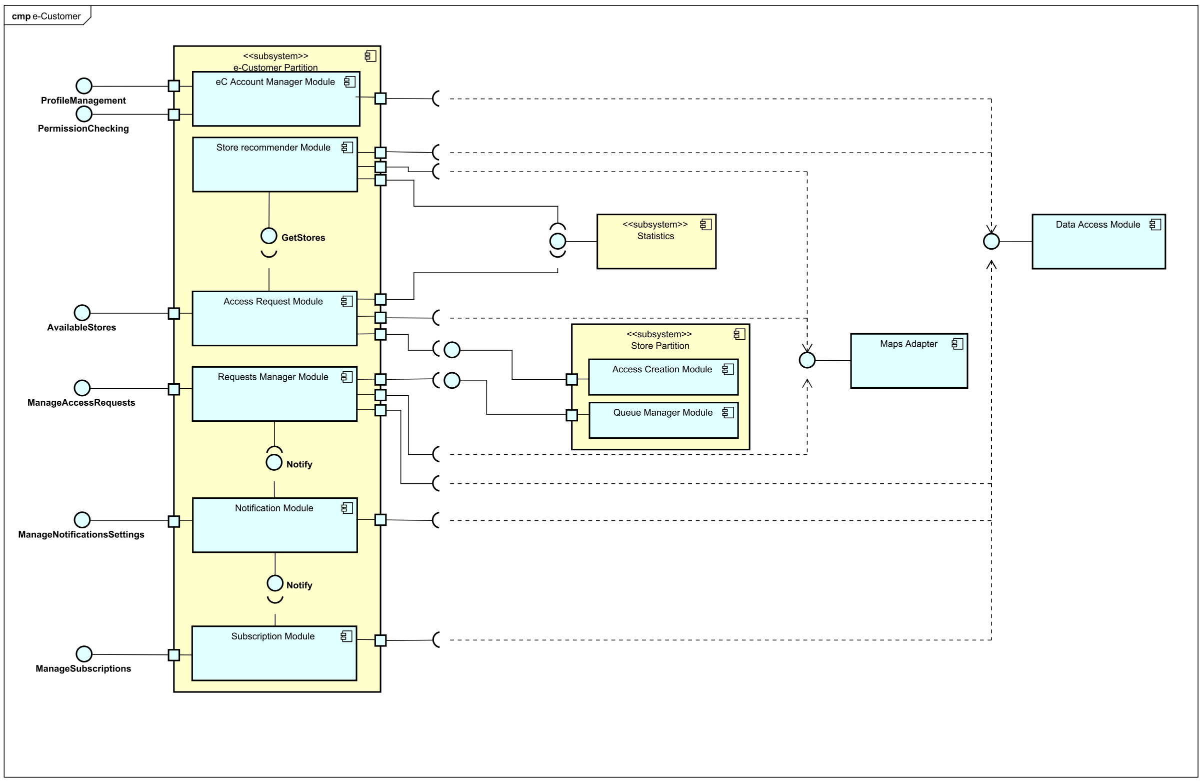


Figure 6: Component view of the e-Customer subsystem

2.2.3 Statistics subsystem

The Statistics subsystem is shown in Figure 7. It is composed of:

- Customer statistics: is responsible for elaborating data such as visit time or preferred store
- Store statistics: computes and aggregates data like average time of visit or mean time of queue

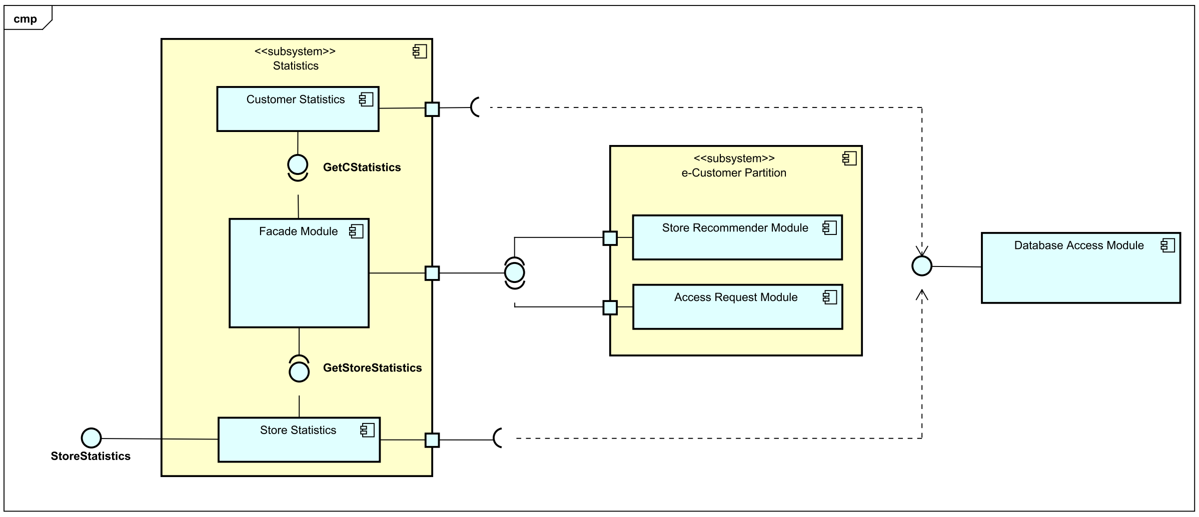


Figure 7: Component view of the Statistic subsystem

2.3 Deployment view

The deployment view of the application can be seen in Figure 8. It uses a 5-tier architecture to separate the web aspects from the application itself and from the data. This way, the code can be better decomposed to increase reusability and for security reasons, exposing only the web tier to browsers. The safety of the system is ensured by the use of two DMZs, one isolating the web tier from the rest and one to better protect the data layer.

With respect to the theoretical layer subdivision, layers 2 and 3 have been united through the use of Apache Tomcat, that covers both the web server's and servlet container's functions.

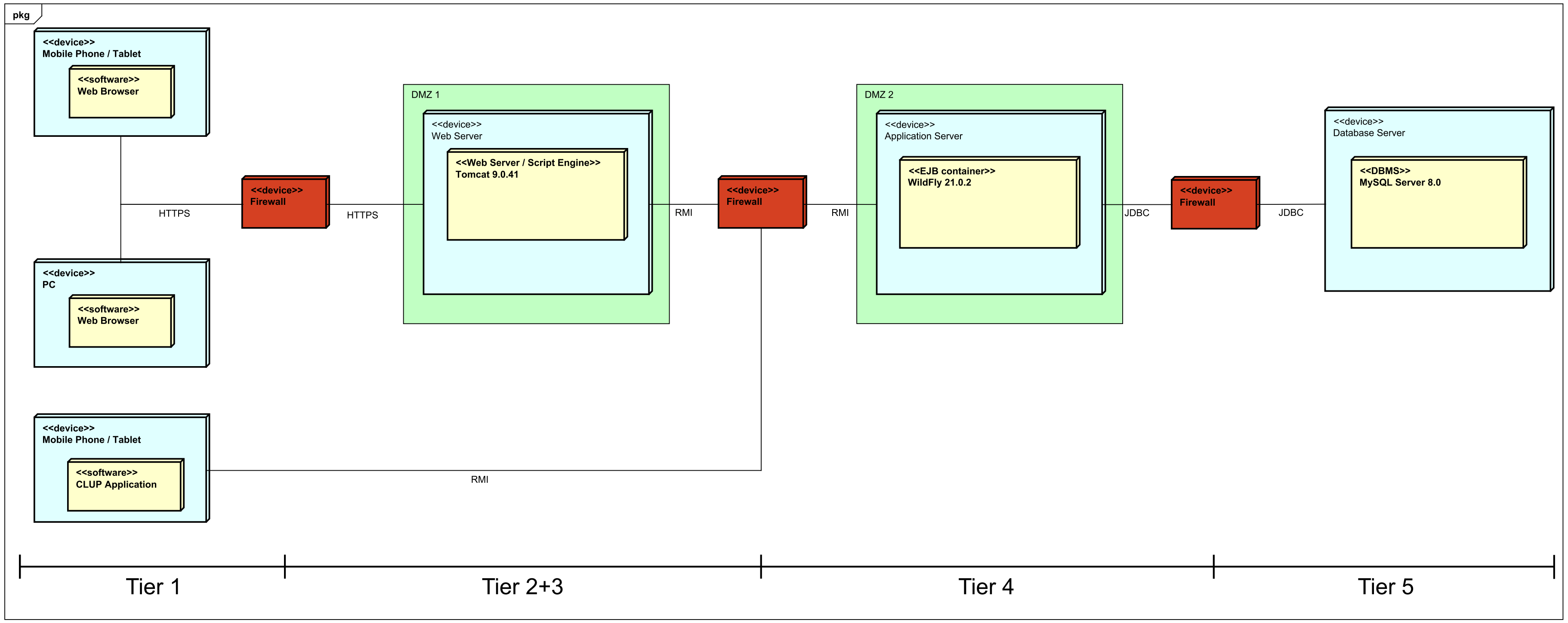


Figure 8: Deployment diagram of the application

2.4 Runtime view

This section covers some of the interactions among the system's components.

Figure 9 shows the interaction between modules to achieve the e-Customer login function.

Diagrams in Figures 10 and 11 represent the different steps needed for getting a ticket or booking a visit for an e-Customer. The core of these processes is the retrieval of stores, done through the following steps:

- The Access Request module asks for the localization of the user
- The Store Recommender requests stores close to the user based on the parameters defined by the user (as mean of transport, maximum travel time...)
- The Recommender then queries the database to know which of the candidate stores use CLup
- The returned stores are checked for availability to host more tickets based on the queues and time of visits
- The finally filtered list is then returned to the user

This order has been defined to offer an incremental filter, allowing to access the maps API only once and limit the accesses to the DB to requests only for previously identified store. Furthermore, the request for stores on a geographic base would be limited anyway even for large cities[5], as opposed to extracting all stores in the database.

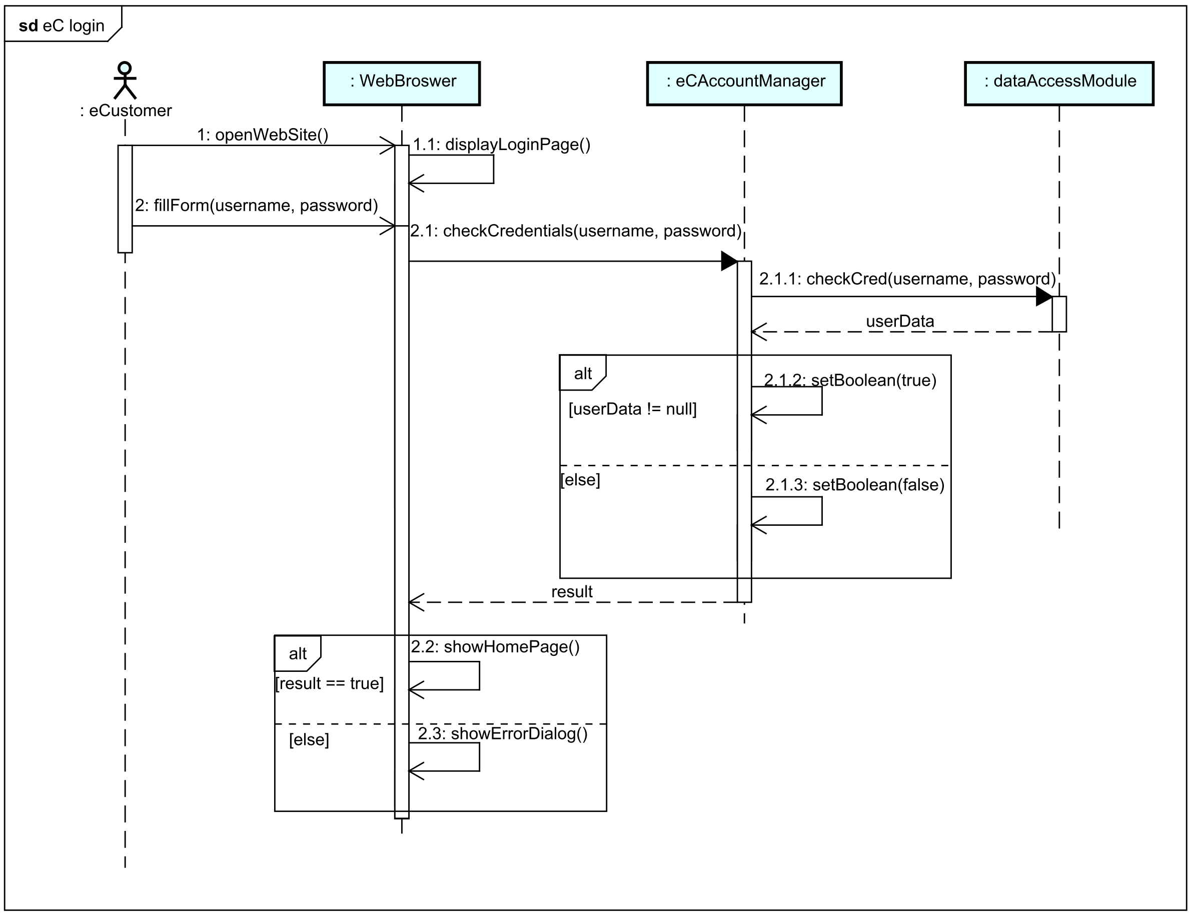


Figure 9: Runtime view of an e-Customer login

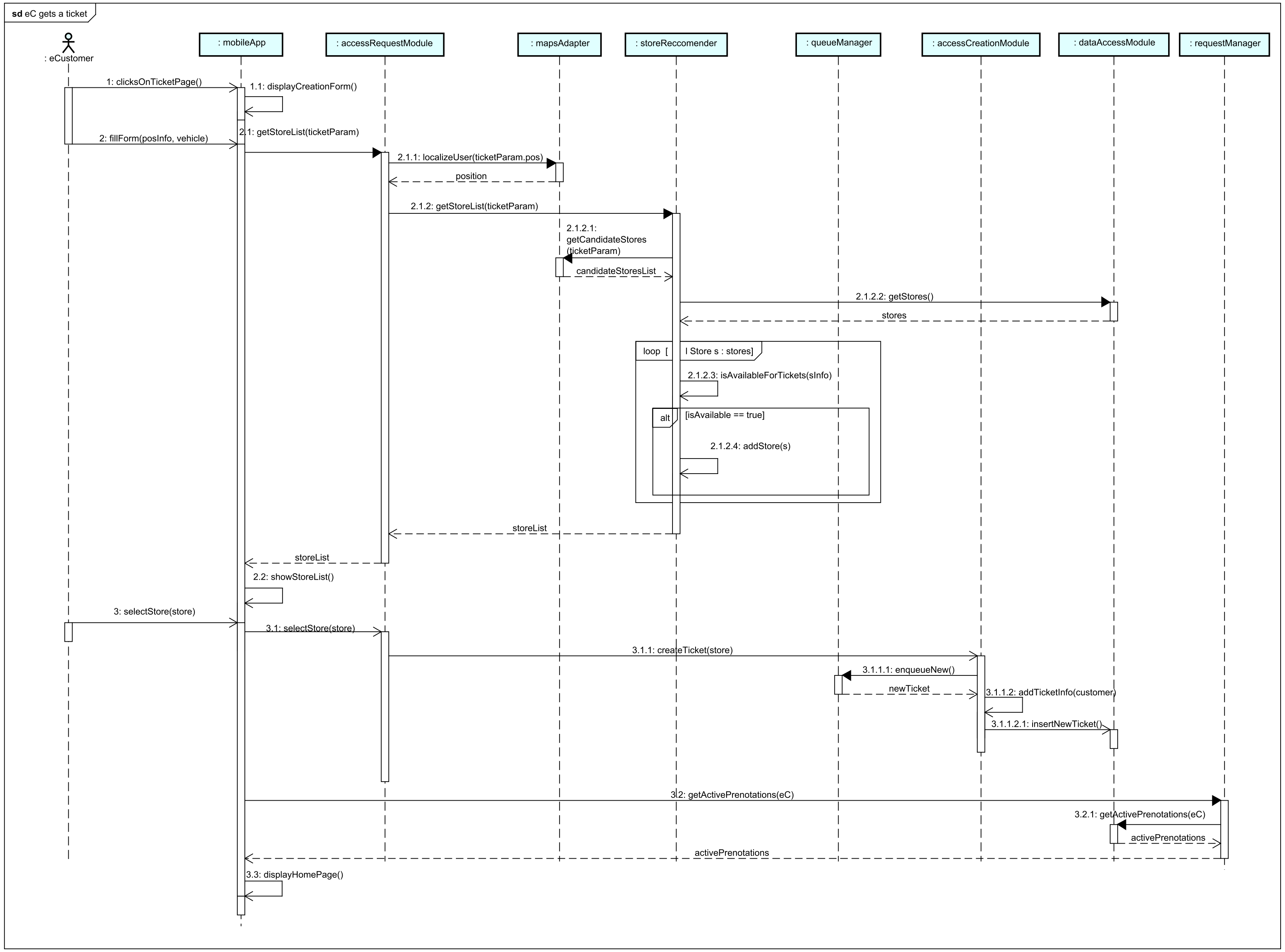

sd eC gets a ticket
: eCustomer
: mobileApp
: accessRequestModule
: mapsAdapter
: storeReccomender
: queueManager
: accessCreationModule
: dataAccessModule
: requestManager
1: clicksOnTicketPage()
1.1: displayCreationForm()
2: fillForm(posInfo, vehicle)
2.1: getStoreList(ticketParam)
2.1.1: localizeUser(ticketParam.pos)
position
2.1.2: getStoreList(ticketParam)
2.1.2.1: getCandidateStores (ticketParam)
candidateStoresList
2.1.2.2: getStores()
stores
loop [l Store s : stores]
2.1.2.3: isAvailableForTickets(sInfo)
alt [isAvailable == true]
2.1.2.4: addStore(s)
storeList
storeList
2.2: showStoreList()
3: selectStore(store)
3.1: selectStore(store)
3.1.1: createTicket(store)
3.1.1.1: enqueueNew()
newTicket
3.1.1.2: addTicketInfo(customer)
3.1.1.2.1: insertNewTicket()
3.2: getActivePrenotations(eC)
3.2.1: getActivePrenotations(eC)
activePrenotations
activePrenotations
3.3: displayHomePage()

Figure 10: Runtime view of an e-Customer getting a ticket

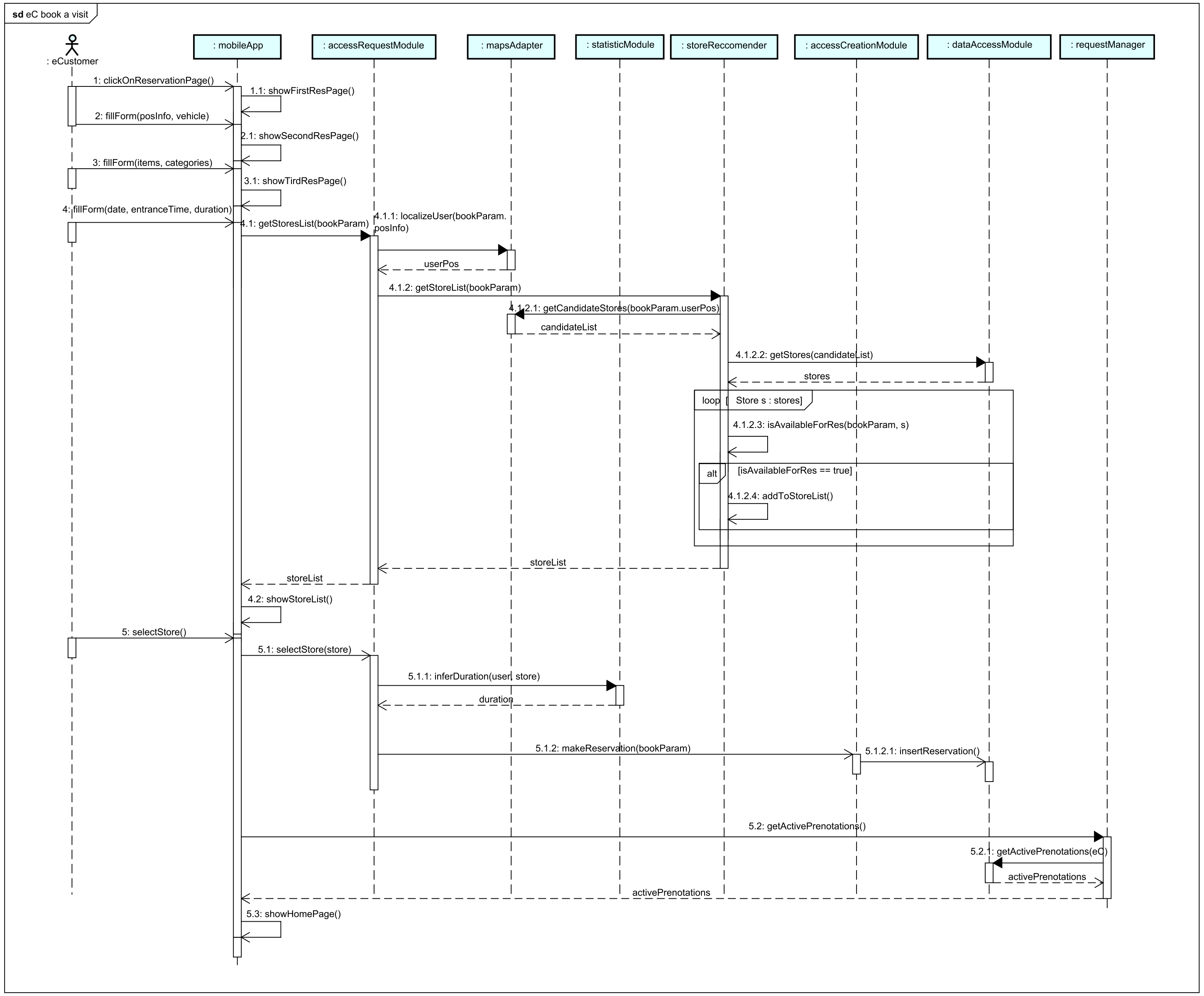

sd eC book a visit
: eCustomer
: mobileApp
: accessRequestModule
: mapsAdapter
: statisticModule
: storeReccomender
: accessCreationModule
: dataAccessModule
: requestManager
1: clickOnReservationPage()
1.1: showFirstResPage()
2: fillForm(posInfo, vehicle)
2.1: showSecondResPage()
3: fillForm(items, categories)
3.1: showTirdResPage()
4: fillForm(date, entranceTime, duration)
4.1: getStoresList(bookParam)
4.1.1: localizeUser(bookParam.posInfo)
userPos
4.1.2: getStoreList(bookParam)
4.1.2.1: getCandidateStores(bookParam.userPos)
candidateList
4.1.2.2: getStores(candidateList)
stores
loop [Store s : stores]
4.1.2.3: isAvailableForRes(bookParam, s)
alt [isAvailableForRes == true]
4.1.2.4: addToStoreList()
storeList
storeList
4.2: showStoreList()
5: selectStore()
5.1: selectStore(store)
5.1.1: inferDuration(user, store)
duration
5.1.2: makeReservation(bookParam)
5.1.2.1: insertReservation()
5.2: getActivePrenotations()
5.2.1: getActivePrenotations(eC)
activePrenotations
activePrenotations
5.3: showHomePage()

Figure 11: Runtime view of an e-Customer booking a visit

Figure 12 illustrates how the request of ticket deletion by an e-Customer is carried out, while Figures 13 and 14 concern Store Manager activities, namely modifying their store's information and scanning entering tickets or reservations.

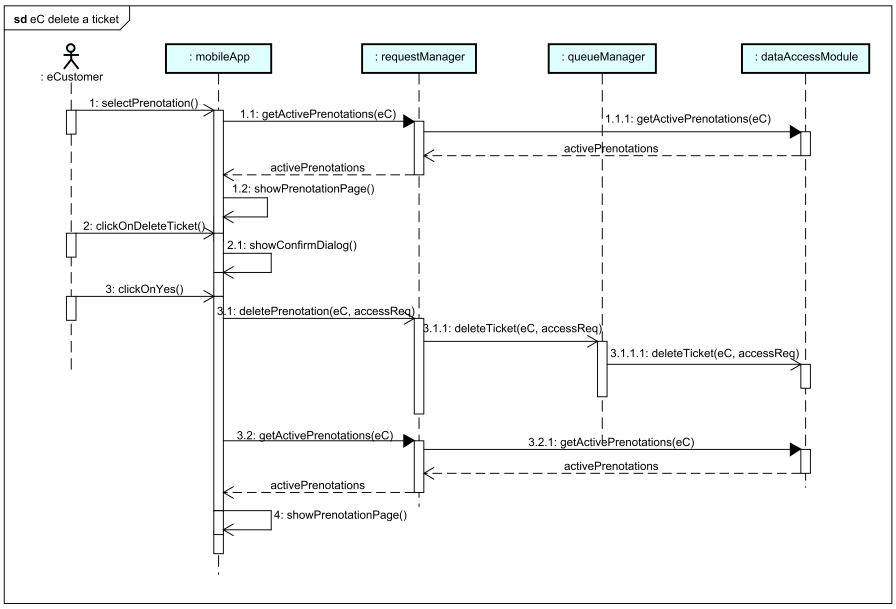


Figure 12: Runtime view of an e-Customer deleting a ticket

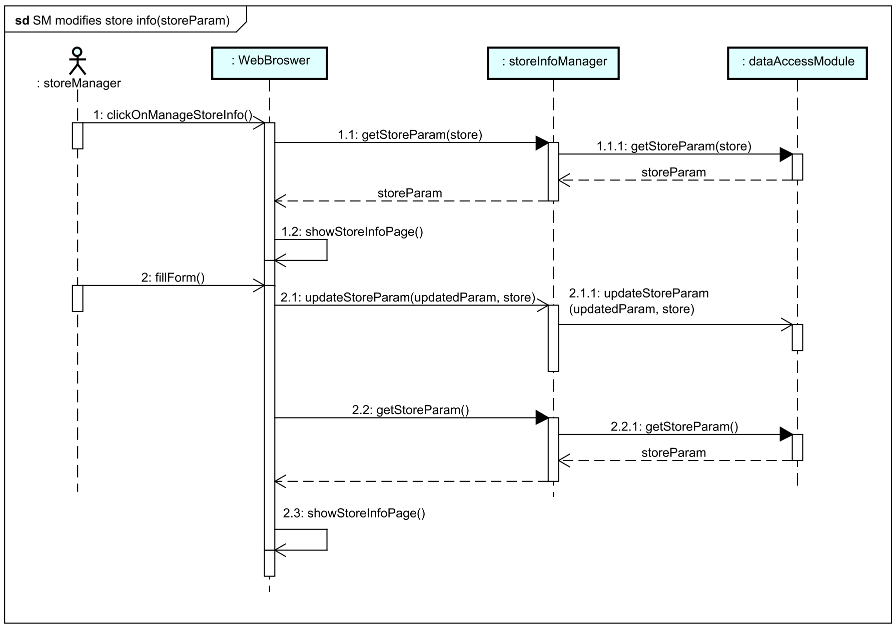

sd SM modifies store info(storeParam)
: storeManager
: WebBroswer
: storeInfoManager
: dataAccessModule
1: clickOnManageStoreInfo()
1.1: getStoreParam(store)
1.1.1: getStoreParam(store)
storeParam
storeParam
1.2: showStoreInfoPage()
2: fillForm()
2.1: updateStoreParam(updatedParam, store)
2.1.1: updateStoreParam (updatedParam, store)
2.2: getStoreParam()
2.2.1: getStoreParam()
storeParam
2.3: showStoreInfoPage()

Figure 13: Runtime view of a Store Manager modifying their store's information

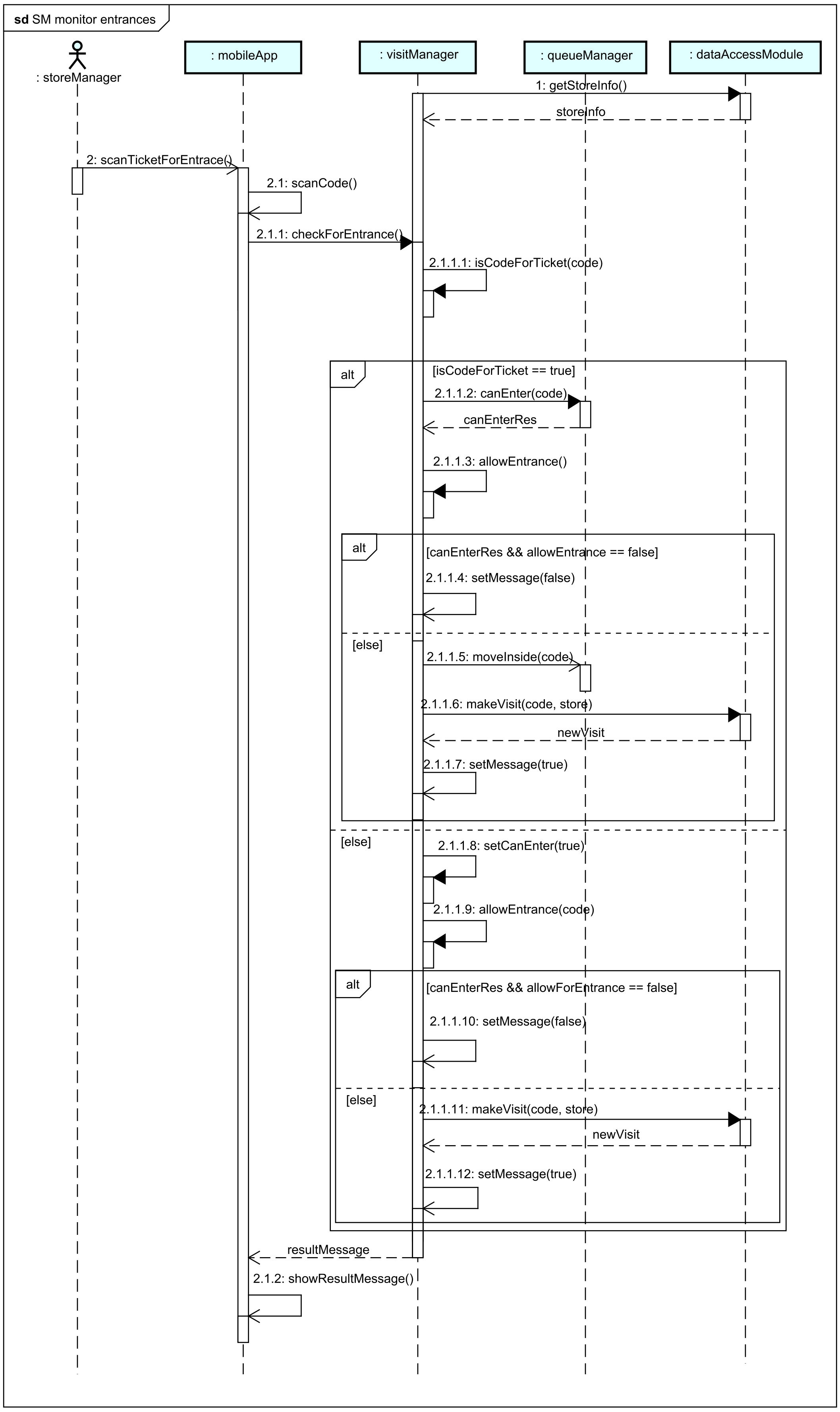

sd SM monitor entrances
: storeManager
: mobileApp
: visitManager
: queueManager
: dataAccessModule
1: getStoreInfo()
storeInfo
2: scanTicketForEntrace()
2.1: scanCode()
2.1.1: checkForEntrance()
2.1.1.1: isCodeForTicket(code)
alt
[isCodeForTicket == true]
2.1.1.2: canEnter(code)
canEnterRes
2.1.1.3: allowEntrance()
alt
[canEnterRes && allowEntrance == false]
2.1.1.4: setMessage(false)
[else]
2.1.1.5: moveInside(code)
2.1.1.6: makeVisit(code, store)
newVisit
2.1.1.7: setMessage(true)
[else]
2.1.1.8: setCanEnter(true)
2.1.1.9: allowEntrance(code)
alt
[canEnterRes && allowForEntrance == false]
2.1.1.10: setMessage(false)
[else]
2.1.1.11: makeVisit(code, store)
newVisit
2.1.1.12: setMessage(true)
resultMessage
2.1.2: showResultMessage()

Figure 14: Runtime view of a Store Manager monitoring accesses

2.5 Component interfaces

Figure 15, Figure 16 and Figure 17 show the interfaces exposed by subsystems to interact with other components.

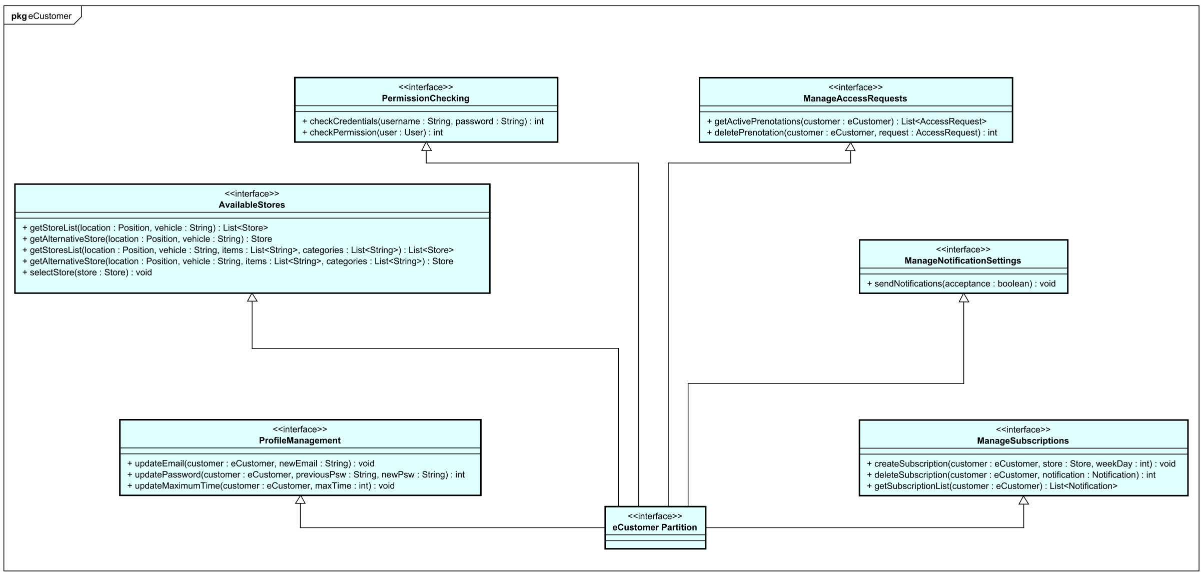

pkg eCustomer
<<interface>>
PermissionChecking
+ checkCredentials(username : String, password : String) : int
+ checkPermission(user : User) : int
<<interface>>
ManageAccessRequests
+ getActivePrenotations(customer : eCustomer) : List<AccessRequest>
+ deletePrenotation(customer : eCustomer, request : AccessRequest) : int
<<interface>>
AvailableStores
+ getStoreList(location : Position, vehicle : String) : List<Store>
+ getAlternativeStore(location : Position, vehicle : String) : Store
+ getStoresList(location : Position, vehicle : String, items : List<String>, categories : List<String>) : List<Store>
+ getAlternativeStore(location : Position, vehicle : String, items : List<String>, categories : List<String>) : Store
+ selectStore(store : Store) : void
<<interface>>
ManageNotificationSettings
+ sendNotifications(acceptance : boolean) : void
<<interface>>
ProfileManagement
+ updateEmail(customer : eCustomer, newEmail : String) : void
+ updatePassword(customer : eCustomer, previousPsw : String, newPsw : String) : int
+ updateMaximumTime(customer : eCustomer, maxTime : int) : void
<<interface>>
ManageSubscriptions
+ createSubscription(customer : eCustomer, store : Store, weekDay : int) : void
+ deleteSubscription(customer : eCustomer, notification : Notification) : int
+ getSubscriptionList(customer : eCustomer) : List<Notification>
<<interface>>
eCustomer Partition

Figure 15: Component interfaces for the e-Customer subsystem

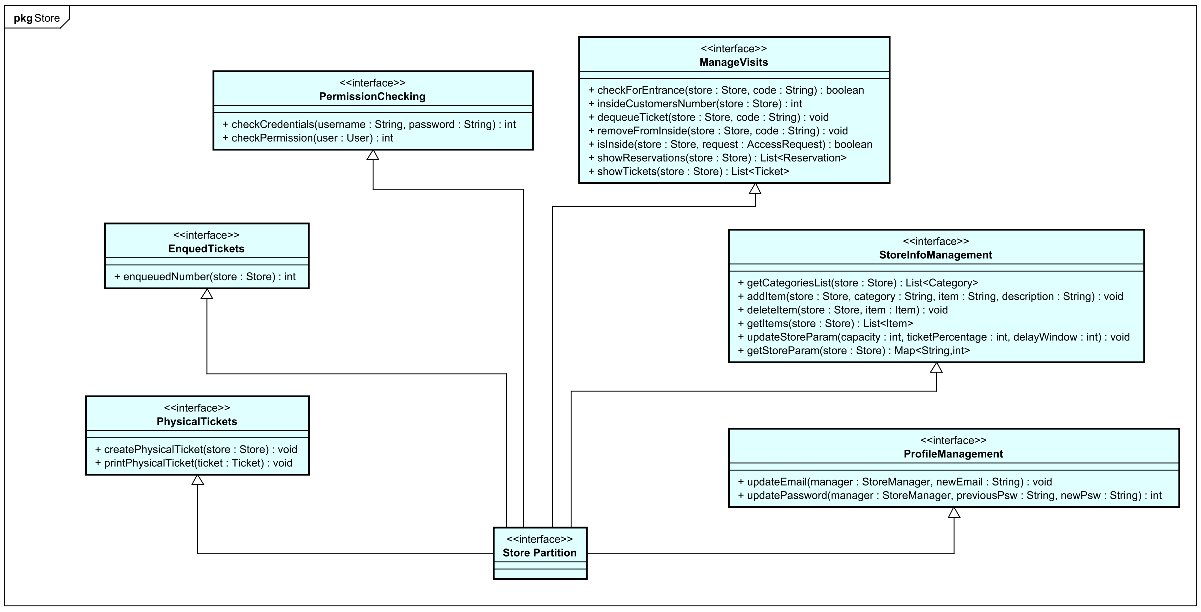

pkg Store
<<interface>>
ManageVisits
+ checkForEntrance(store : Store, code : String) : boolean
+ insideCustomersNumber(store : Store) : int
+ dequeueTicket(store : Store, code : String) : void
+ removeFromInside(store : Store, code : String) : void
+ isInside(store : Store, request : AccessRequest) : boolean
+ showReservations(store : Store) : List<Reservation>
+ showTickets(store : Store) : List<Ticket>
<<interface>>
PermissionChecking
+ checkCredentials(username : String, password : String) : int
+ checkPermission(user : User) : int
<<interface>>
EnquedTickets
+ enqueuedNumber(store : Store) : int
<<interface>>
StoreInfoManagement
+ getCategoriesList(store : Store) : List<Category>
+ addItem(store : Store, category : String, item : String, description : String) : void
+ deleteItem(store : Store, item : Item) : void
+ getItems(store : Store) : List<Item>
+ updateStoreParam(capacity : int, ticketPercentage : int, delayWindow : int) : void
+ getStoreParam(store : Store) : Map<String,int>
<<interface>>
PhysicalTickets
+ createPhysicalTicket(store : Store) : void
+ printPhysicalTicket(ticket : Ticket) : void
<<interface>>
ProfileManagement
+ updateEmail(manager : StoreManager, newEmail : String) : void
+ updatePassword(manager : StoreManager, previousPsw : String, newPsw : String) : int
<<interface>>
Store Partition

Figure 16: Component interfaces for the Store Manager subsystem

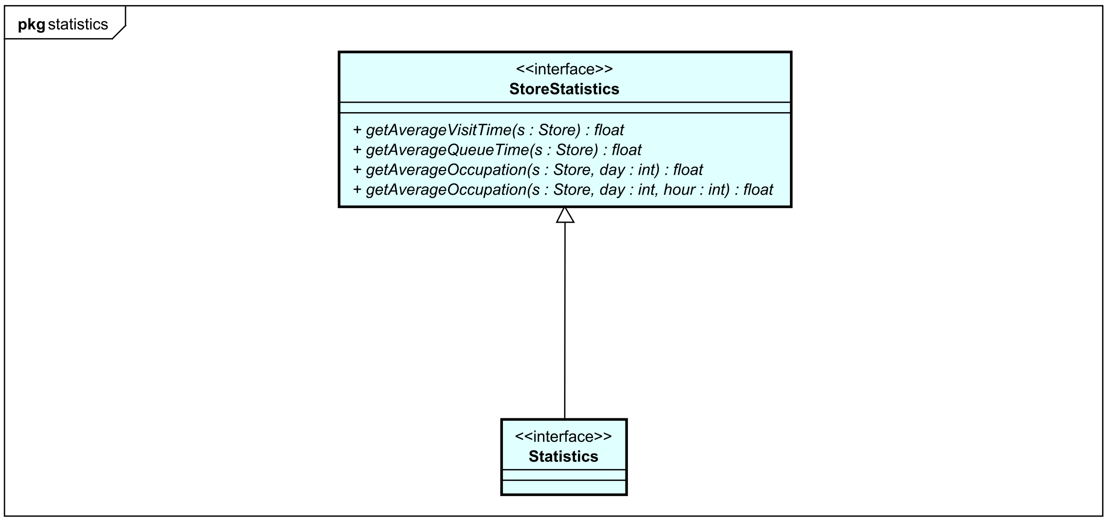

pkg statistics
<<interface>>
StoreStatistics
+ getAverageVisitTime(s : Store) : float
+ getAverageQueueTime(s : Store) : float
+ getAverageOccupation(s : Store, day : int) : float
+ getAverageOccupation(s : Store, day : int, hour : int) : float
<<interface>>
Statistics

Figure 17: Component interfaces for the Statistics subsystem

2.6 Selected architectural styles and patterns

This section reports some directions on the use of design patterns during the implementation of the CLup system. As previously stated, the system is based on a modular design, but the following patterns are suggested:

- The base architecture pattern is MVC (Model-View-Controller), as it allows to decouple the application logic from the user interaction
- To support MVC, the Observer Pattern[7] should be used
- To have an extended interface to the database the DAO[4] (Data Access Object) architectural pattern is used to separate the business logic from the data tier, to moreover improve testing simplicity, to increase code readability and to make the independent from the data access technology
- The application's and the online mapping system is done through an Adapter, in order to create a layer between the application and the API, and have a unified interface[2] for all the system's components.
- For what concerns the Statistic Module, a façade pattern[6] is used to offer a unique and easy-to-use interface to its components.
- The frontend should use a Command pattern[3] to hide the difference among functionalities offered from the UI from a code perspective
- The notification module should work as a Servant to the modules using it

2.7 Other design decisions

2.7.1 Algorithm for queue rearrangement

In order to maximize a store occupancy, the system adopts an algorithm to let customers jump forward if a place is freed because someone dequeued; it is shown here through pseudocode.

What is important to note is that this behaviour is based on empty places mapped 1:1 to e-Customers through notifications (physical tickets are not allowed to shift), as shown in Figure 18: every e-Customer is allowed to jump only to one place, so that their original order can be respected (if everyone accepts). An e-Customer that rejects the jump or lets the time window expire won't be asked again (in red), while, in an extreme case like the one shown (with multiple drop during one period) a customer might be asked to jump several times.

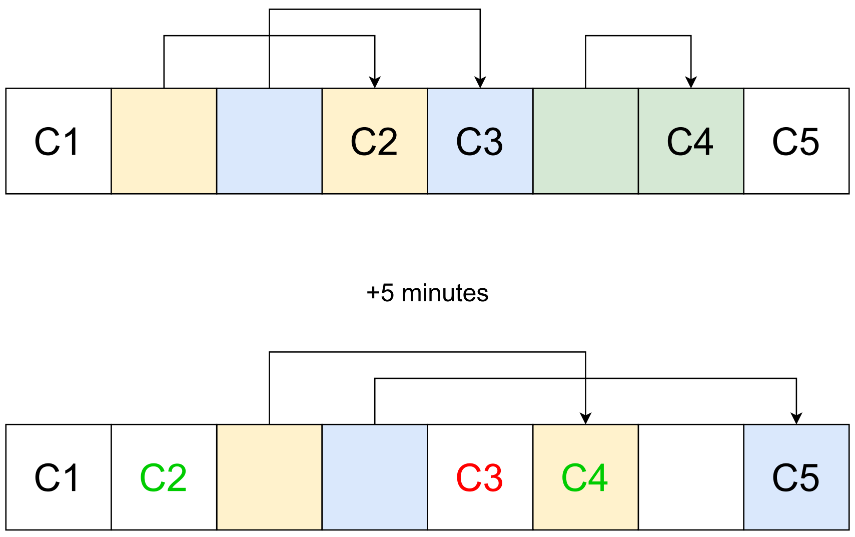


Figure 18: Visualization of the queue algorithm

Variables This pseudocode assumes the existence of a queue of tickets, treated for simplicity as an array.

Notifier Every 5 minutes, a number of customers at most equal to the number of empty places is notified of an available jump, for which they won't be notified again (to avoid both multiple notifications and being notified either after a rejection or an expiration of the given time). The empty places notification order reflects the current e-Customer order.

```
doNotNotify = []
while(true)
        holes = queue.getHoles()
        nextPlace = -1
        foreach(h in holes)
                nextPlace = max(nextPlace + 1, h + 1)
                while(nextplace in holes
                        or queue[nextplace] in doNotNotify)
                        nextplace++
                queue[nextplace].getCustomer().notifyJumpTo(h)
                doNotNotify.add(queue[nextplace], h)
        sleep(300)
```

Jump If a customer accepts the jump, their place in queue is updated:

```
moveTo(ticket, jumpDest)
```

2.7.2 QR code generator

This section shows how QR codes can be generated in order to ensure security and meet other design decisions of the applications. The format of the string changes based on the type of the access request, but the suggested general form consists of

```
control_character + MD5(emissionDT + storeID + accessParam)
```

Where the `control_character` identifies the type of request through a number and the accessParam, exclusive to the type:

- Ticket:
  - `control_character`: 0
  - `accessParam`: nOrder
- Reservation:
  - `control_character`: 1
  - `accessParam`: entranceTime

Through the use of MD5 all codes are guaranteed to be of length 33 characters. Figure 19 shows an example of generated QR code for a ticket with parameters:

- emissionDT: 2020-12-31 10:53:31 (without spaces and special characters)
- storeID: 154
- nOrder: 51

That corresponds to the string

```
0dd8e72b69b8e4fac5c3f3bef07e09a4a
```

Figure 19: Generated QR code

3 User interface design

This section illustrates the UX diagrams, i.e. the transitions between the user interface's pages. The application's mock-ups can be found in the RASD.

3.1 e-Customer UX diagram

The e-Customer application is composed of four sections reachable through tabs:

- Home page: displays some instructions to better explain the application's interfaces and functions, and how to navigate through them. Moreover, it shows the active access requests (if present)
- Ticket creation: allows the user to line up by filling the creation form and then choosing a store among the suggested ones
- Book a visit: allows the e-Customer to create a new reservation by passing through the three booking form steps and then by choosing the store they like from the list of the ones meeting the specified parameters
- Settings: allows e-Customers to manage notifications, subscriptions and their profile

These sections can be reached after the login or signup process Figure 20 shows the transitions.

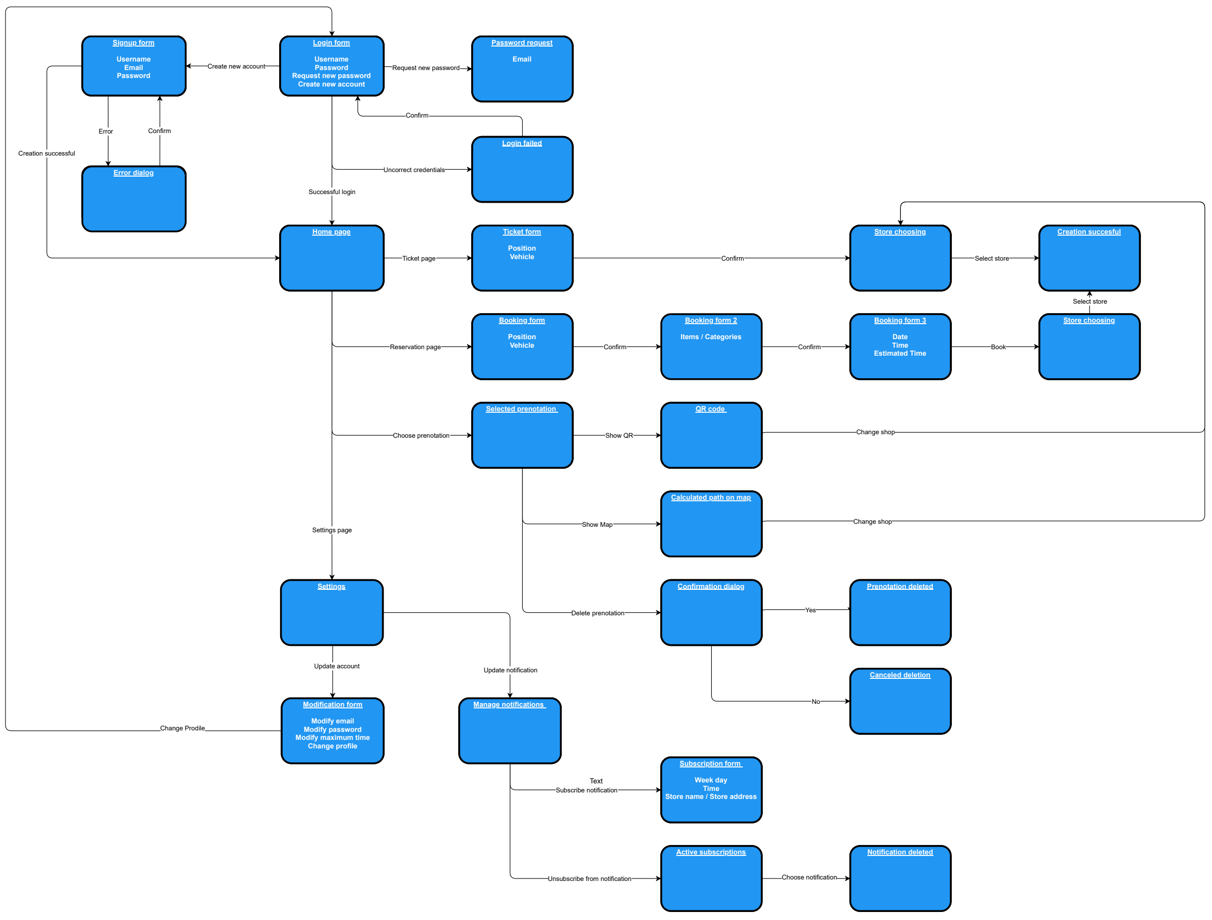


Figure 20: UX diagram of the e-Customer application

3.2 Store Manager UX diagram

The Store Manager application is composed of four sections as well:

- Home page: displays instructions about the application's functionalities and gives the possibility to access other functions such as viewing store statistics or modifying the store's parameters
- Ticket creation: allows the Store Manager to see the enqueued tickets and to create and print physical ones
- Reservations: allows the Store Manager to visualize active reservations by date
- Settings: allows Store Manager to change information related to their profile

These sections can be reached after a login or signup process ended up successfully; furthermore, every screen can activate the QR code scanning function. Figure 21 shows the transitions among the interfaces.

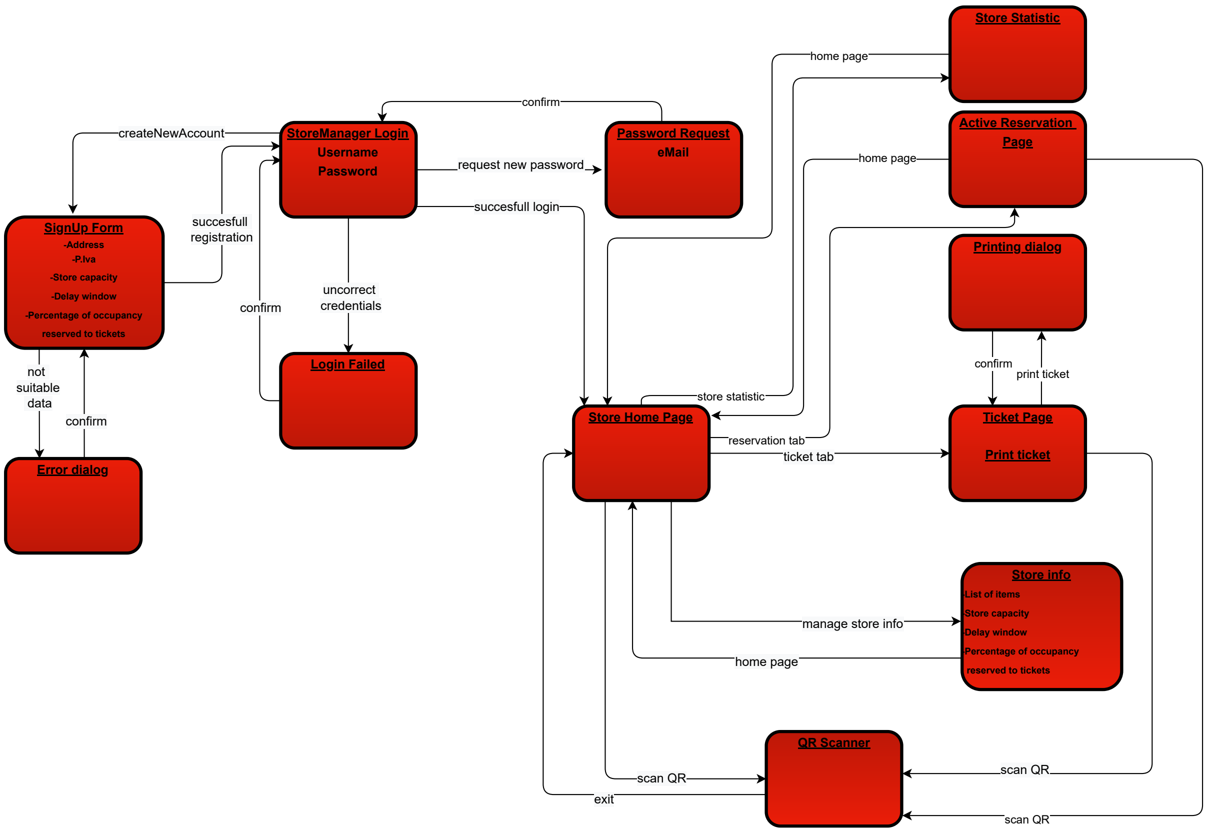


Figure 21: UX diagram of the Store Manager application

4 Requirements traceability

This section illustrates which components realize each requirement previously specified in the RASD.

[R0] Must allow users to provide credentials
- eC Account manager module
- SM Account manager module

[R1] Username must be unique
- eC Account manager module
- SM Account manager module

[R2] Store id must be unique
- Store info manager module

[R3] Must allow store managers to add or modify their store's information
- Store info manager module

[R4] Users must be able to login
- eC Account manager module
- SM Account manager module

[R5] Must be able to provide the list of available stores in the user's proximity
- Access Request module
- Store recommender

[R6] Must know the e-Customer's position
- Access Request module
- Maps adapter

[R6.1] Must localize the e-Customer or let them provide manually a position
- Access Request module
- Maps adapter

[R7] Must assign a unique and sequential reservation id
- Access creation

[R8] Must generate a QR code to be scanned at entrance and exit
- Access creation

[R9] Must let e-Customers choose their mean of transport
- Access Request module

[R10] Must allow the entrance when it is the customer's turn
- Visit manager module
- Queue manager module

[R11] Must allow a delay of at most M minutes on a customer's turn
- Visit manager module
- Queue manager module

[R12] Must show e-Customers the number of enqueued customers in front of them
- Queue manager module
- Store recommender

[R13] Must show users historical data on given weekdays
- Access Request module
- Customer visit statistics module

[R14] At least P% places must be reserved for tickets
- Access Request module
- Store info manager module
- Store recommender module

[R15] Must notify e-Customers within a suitable time from entrance
- Customer visit statistics module
- Notification module

[R15.1] Notification time must be based on current store occupation
- Access creation module
- Customer visit statistics module
- Notification module

[R15.2] Notification time must be based on estimated travel time
- Maps Adapter module
- Notification module

[R16] Must let e-Customers advance if a preceding customer dequeued
- Queue manager module
- Notification module

[R17] Must keep track of people inside the stores
- Visit manager module

[R17.1] Must allow scanning QR codes on entrance and exit
- Visit manager module

[R18] Must not allow the entrance of more customers than prescribed
- Visit manager module

[R20] Physical tickets must be placed on the same queue as virtual ones
- Access creation module
- Queue manager module

[R21] Physical tickets must be printed
- Access Creation module

[R22] Must allow Physical Customers to scan their code in order to dequeue
- Visit manager module

[R23] e-Customers who book a visit must be allowed to enter at their chosen time
- Visit manager module

[R24] Must let e-Customers insert the expected duration of the visit
- Access Request module

[R24.1] Must be able to infer and suggest the duration for long term customers
- Customer visit statistics module

[R25] Must let e-Customers insert a list of items/categories they intend to buy
- Access Request module

[R26] Alternatives must be based on information provided by the e-Customer and store occupation (both real time and historical)
- Requests manager module
- Queue manager module
- Customer visit statistics module
- Store occupation stats module
- Store recommender module
- Maps adapter module

[R27] e-Customers must be able to subscribe to the notification service
- Subscription module

[R28] Must let e-Customers unsubscribe from the notification service
- Subscription module

[R29] The system must notify the e-Customers based on their choices
- Subscription module
- Store occupation stats module
- Notification module

5 Implementation, integration and test plan

This section reports the guidelines to implement and test the CLup application. The chosen approach consists in a bottom-up development based on the dependency graph identified in Figure 22.

5.1 Implementation

The chosen order of implementation, based on the highlighted dependencies and aimed at obtaining core functionalities as soon as possible is the following:

- Data Access Module
- Maps Adapter
- SM Account Manager Module
- eC Account Manager Module
- Queue Manager Module
- Visit Manager Module
- Access Creation Module
- Store Info Manager Module
- Store Statistics
- Customer Statistics
- Store Recommender Module
- Access Request Module
- Notification Module
- Requests Manager Module
- Subscription Module

If the team is able to handle a good level of parallelization, Figure 23 shows a suggestion of steps aimed at minimizing the overall development time; each module's length is dimensioned based on its expected implementation difficulty, and the critical path (that needs not to be delayed in order to meet potential deadlines) is highlighted in red. The graph is also useful for determining which elements can be delayed without impacting the total development time.

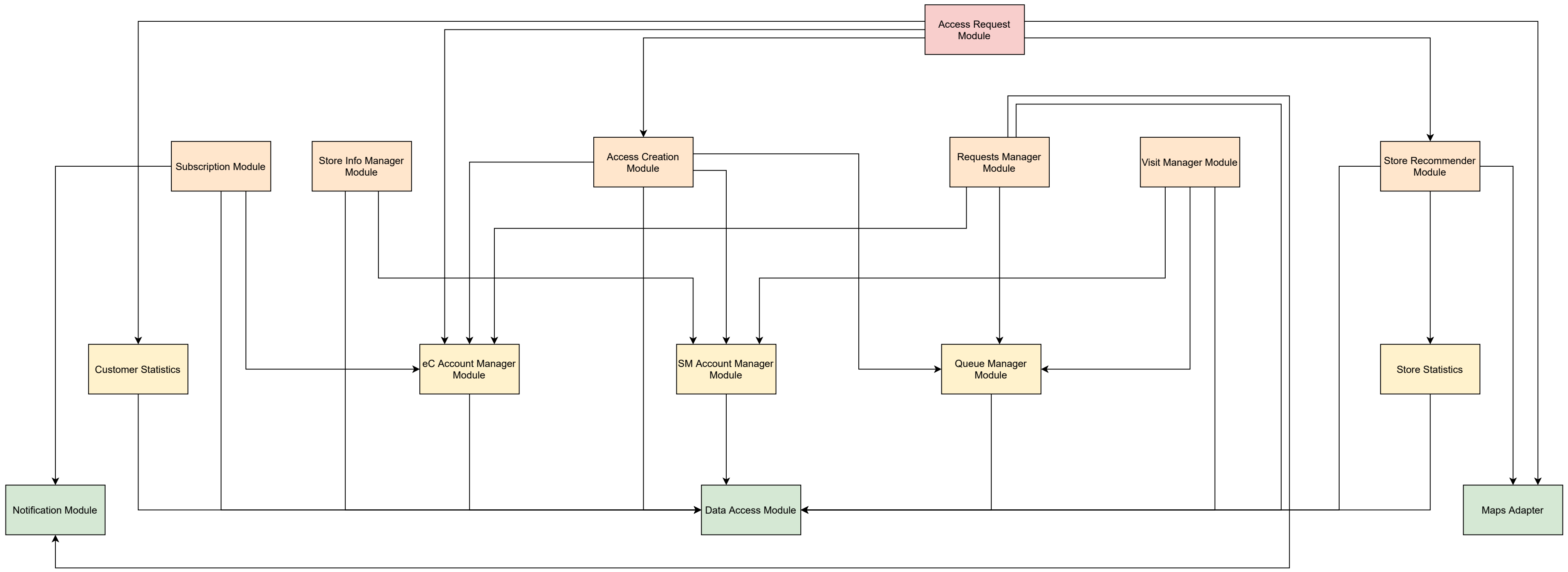


Figure 22: Dependency graph

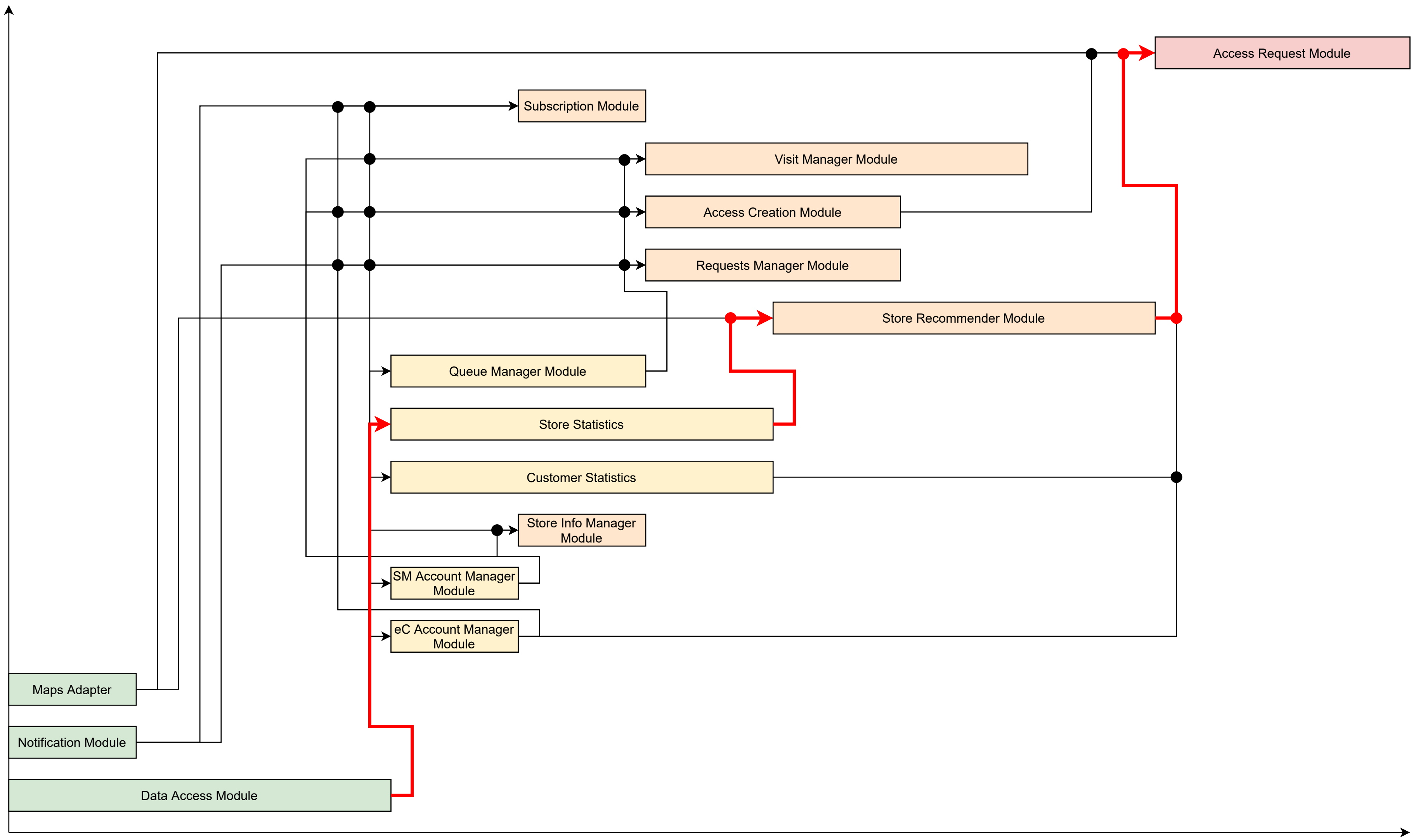


Figure 23: Dependency graph with expected time and critical path highlighted

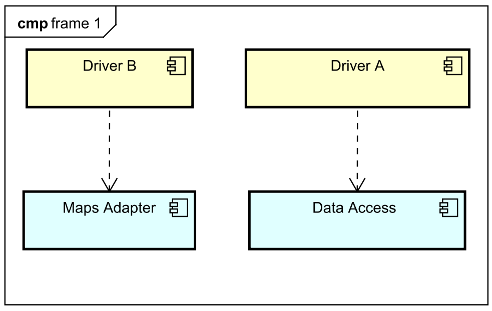
5.2 Integration and Testing

5.2.1 Unit testing

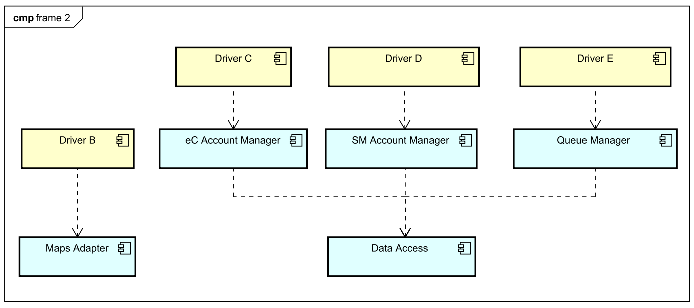
The unit test can be performed as soon as a component has been developed, or even during its realization to better check its conformity to requirements. Having chosen a bottom-up approach, only Drivers are needed in order to correctly perform unit tests.

5.2.2 Integration strategy

The implementation order has been developed considering the possibility to immediately integrate components and test their interactions; this way development time can be optimized even more and errors can be found as early as possible. The integration strategy is shown in the following figures.



(a) First components with their drivers



(b) Adding some modules that use function provided by the Data Access and their drivers

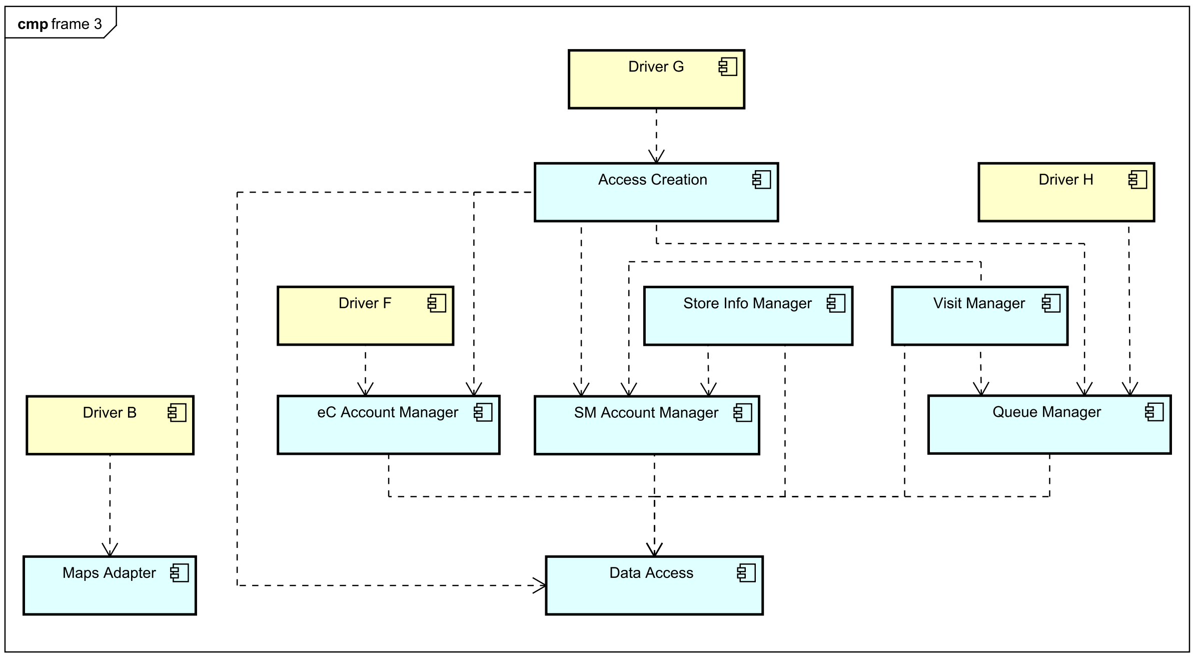


Figure 25: Realization of Access Creation, Store Info Manager, and Visit Modules

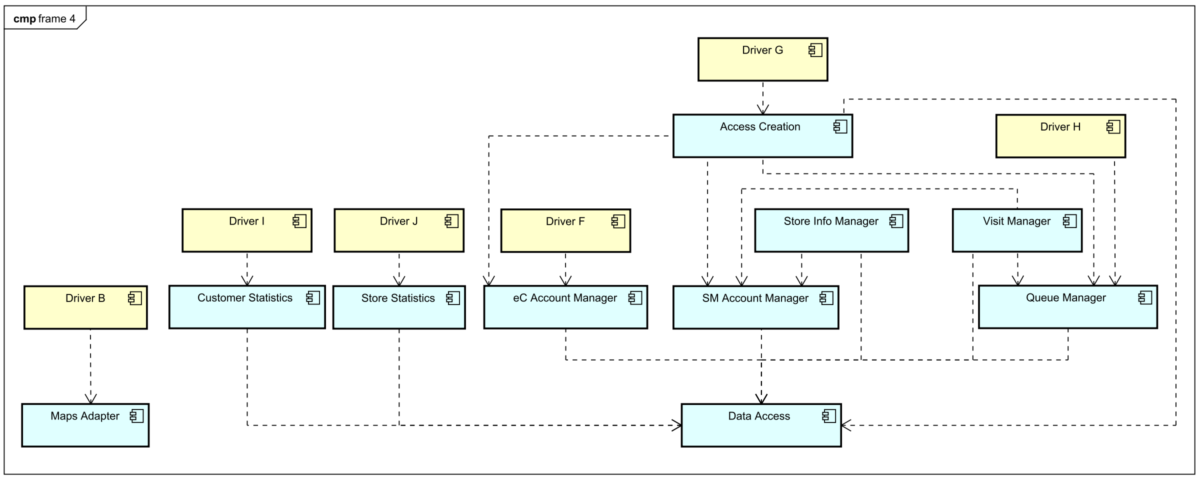


Figure 26: Implementation and integration of Statistics and Store Info Manager Modules

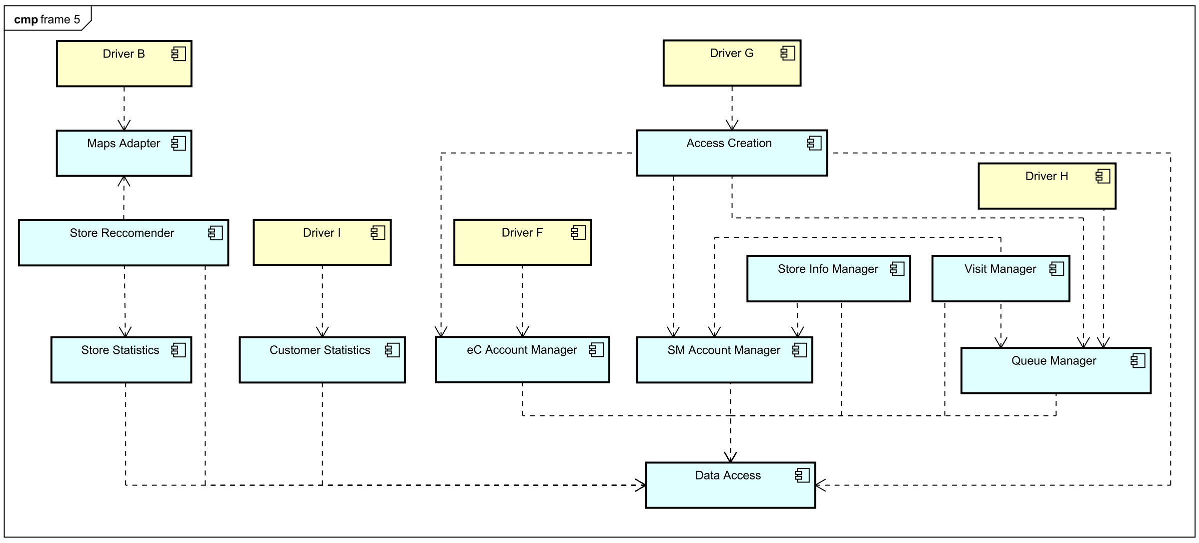


Figure 27: Adding the Store Recommender Module and integrating it with its dependencies

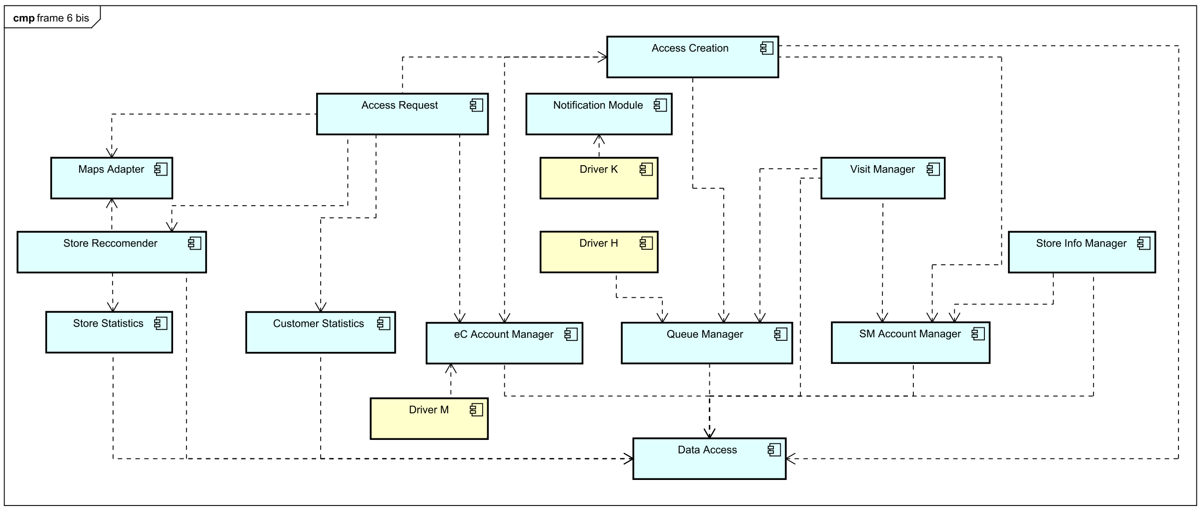


Figure 28: Integration of Access Request and Notification Modules

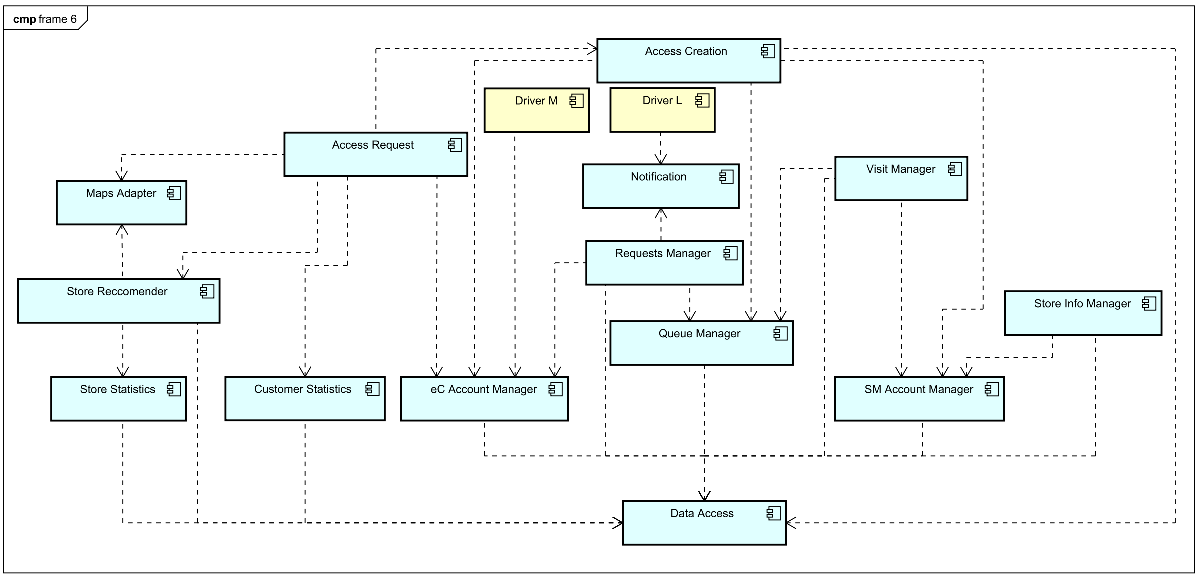


Figure 29: Addition of the Requests Manager Module

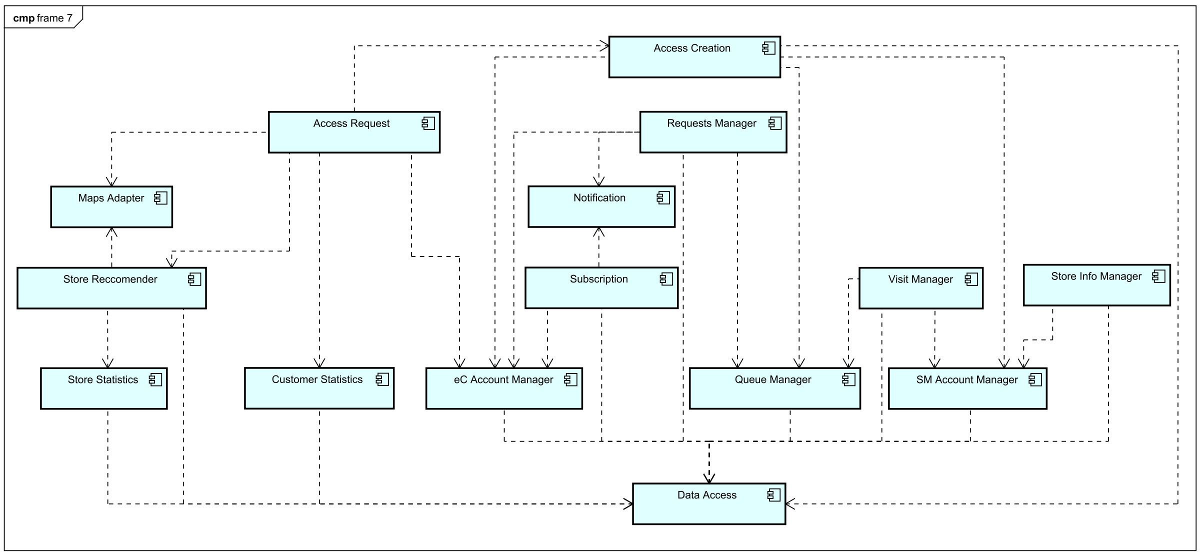


Figure 30: Final module integration with the previously missing Subscription Module

5.3 System Testing

Once all the modules have been integrated, the system as whole should be tested for requirements satisfaction, both functional and non-functional.

Performance Testing Checking if the performance constraints defined in the RASD[1] are met is an important phase when testing the system as a whole. Moreover, it can help with the optimization of algorithms and queries (if helpful) and the identification of bottlenecks.

Load Testing This test should check that the system can handle the load defined in the previous document[1] and is fundamental to expose potential memory occupancy issues.

Stress Testing This final test type should test the ability of the system to handle failures and recover from them.

Other tests After the system testing is successfully completed, an acceptance test can be done in order to assess the usability and the validity of the system. Finally, if the system is going to be used in an environment which is different from the developing one, and installation test can be carried out

6 Effort and tools

6.1 Effort spent breakdown

				TOTAL
Previous DD analysis	3.5	4	3	4.5
Hardware architecture	1	1	1	1
Component view	3.5	6.5	11.5	14.5
ER model	4	-	2	4
Document writing	14.5	4.5	5.5	14.5
UX design	-	3	3	5
Requirements traceability	1.5	0.5	1.5	2.5
Deployment diagrams	1	4.5	4.5	4.5
Class diagram	2.5	2	-	2.5
IIT	4	6.5	6.5	6.5
Algorithm design	1	1	1	1
Sequence diagrams	2	5.5	1	5.5
Interface diagram	0.5	0.5	1.5	1.5
TOTAL	39	39.5	42	67.5

6.2 Software tools

This is the list of tools used during the development of this document:

- **TeXstudio**: Text editor
- **MikTeX**: LaTeX compiler
- **Astah UML**: UML modeling (class diagram, sequence diagrams, component diagrams, deployment diagram)
- **Draw.io**: Other figures (ER model, hardware architecture, algorithms)
- **Pageloot**: QR code generator for the algorithm

References

[1] . *CLup RASD*, chapter 3.6.1.

[2] Adapter pattern. URL: http://w3sdesign.com/?gr=s01&ugr=proble.

[3] Command pattern. URL: http://w3sdesign.com/?gr=b02&ugr=proble.

[4] Data Access Object. URL: https://www.oracle.com/java/technologies/dataaccessobject.html.

[5] Associazione Economia e Sostenibilità. Le 10 questioni della food policy di milano. URL: http://mediagallery.comune.milano.it/cdm/objects/changeme:58783/datastreams/dataStream2398282397540390/content?1468850549528.

[6] Facade pattern. URL: http://w3sdesign.com/?gr=s05&ugr=proble.

[7] Observer pattern. URL: http://w3sdesign.com/?gr=b07&ugr=proble.