POLITECNICO MILANO 1863

Design Document

Authors: Mattia Calabresi,
Olexandr Shchukhlyi,
Alessandro Brigandì
Professor: Luciano Baresi

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

The purpose of this document is to clarify and describe different aspects of software development. The goal of Design Document is to achieve precise representation of software parts and its architecture. This document will provide the description of design characteristics required for the implementation of application:

- The high-level architecture design;
- The main components, their interfaces and deployment;
-The runtime view;
-The design patterns;
-The component view design;
-The deployment view.

1.2 Scope

The service ‘‘Clup’’ (stands for Customers line up) gives the possibility to plan your visit to a shop with coordination with the management of a store (call them the third side). The application creates a virtual queue instead of a real one, helping to avoid crowds. Also, the system enables the personnel of the management to monitor a situation in a store.

Clup allows customer two options: to book a visit for the time/day chosen by user, and to accept a token from a machine right outside a store (of course it is not recommended, but in case of not knowing about the application it is the only way).

The location of a customer is getting by GPS, the location of the store is provided by the manager of the shop.

After coming to a shop customer scans the QR code and if the time for a visit is not expired (came not more than some minutes late) user could go in and start to buy what he/she needs. After exiting from the store user is popping out from the virtual queue.

The first option uses the notification feature on the user device, so, a customer gets the message when the planned visit time is coming.

If a person is using a mobile device to stand in queue/book a visit he can see a load of stores to choose the one with the minimum people flow.

The second option presumes that a customer does not know about mandatory using the application to get in a store. So, when a person comes, he got a ticket with the time of a visit. This request is added to the queue such as it will be a request from the first option. To visit/leave the store a customer has to scan his ticket before arrival/departure.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- Booking – recording to visit a store
- Queue – virtual queue where bookings are inserted
- Common User – mobile app user
- Prioritized User – machine user
- Privilege User – manager of the store
- Machine – public terminal near stores where customers can book a visit

1.3.2 Acronyms

- DD – Design Documents
- GPS - Global Positioning System
- UML – Unified Modelling Language
- QR-code – Quick Response Code
- DB – database
- API - application programming interface
- DBMS – database management system
- IOS – iPhone OS
- RMI - Remote Method Invocation
- SMS – Short Message Service
- GUI – Graphical User Interface
- HTTP - HyperText Transfer Protocol
- JSON – JavaScript Object Notation
- XML - eXtensible Markup Language
- REST - Representational State Transfer
- MVC – Model View Controller
- CLup - Customers Line Up

1.4 Revision history

Date	Modifications
29/12	First Version
9/1	Second version. Correction of the document

1.5 Reference Documents

• Specification Document: “SafeStreets Mandatory Project Assignment.pdf”

• Slides of the lectures

1.6 Document Structure

• Chapter 1 describes the scope and purpose of the DD, including the structure of the document and the set of definitions, acronyms and abbreviations used.

• Chapter 2 contains the architectural design choice, it includes all the components, the interfaces, the technologies (both hardware and software) used for the development of the application. It also includes the main functions of the interfaces and the processes in which they are utilised (Runtime view and component interfaces). Finally, there is the explanation of the architectural patterns chosen with the other design decisions.

• Chapter 3 shows how the user interface should be on the mobile and web application.

• Chapter 4 describes the matching between the goals and requirements with the elements which compose the architecture of the application.

• Chapter 5 traces a plan for the development of components to maximize the efficiency of the developer team and the quality controls team. It is divided in two sections: implementation and integration. It also includes the testing strategy.

• Chapter 6 shows the effort spent for each member of the group.

• Chapter 7 includes the reference documents.

2 Architectural design

2.1 High-level components and their interaction

The application presents the classical three-tier architecture, which consists of the next components presenting logic layers: Client Application installed on end-devices and represents the Presentation Layer, Server Application providing the main business logic and representing Application layer, Database Server handling necessary data operations and representing Data access level. The first level (Client Application) provides to the user of the software the possibility to send requests and obtain responses of booking status. The second level (Server Application) handling requests from all users providing concurrency and processing with respect to business logic. The third level (Database server) provides the necessary functionality to work with data.

The graphical representation of the three-tier architecture of the system:

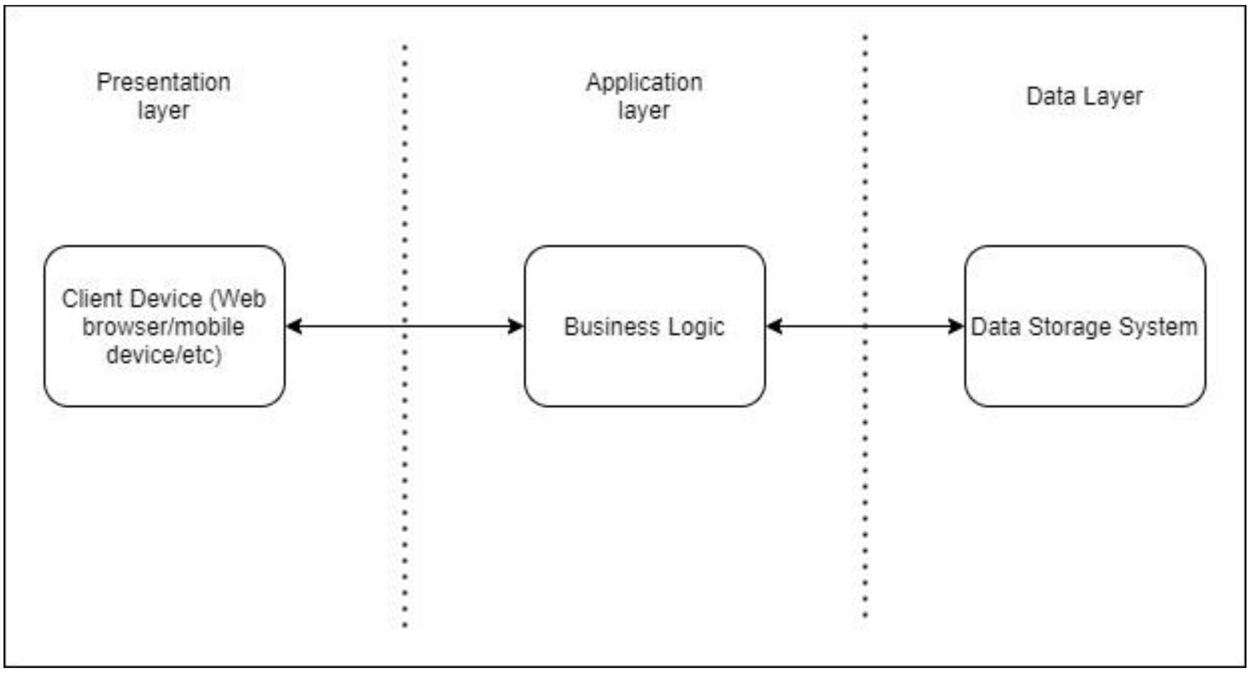


Figure 1 - Three-tier architecture

After describing three-tier architecture, we can start to build system architecture. However, the system should allow to process a big number of concurrent requests, so it is necessary to embed tools for parallel work of several servers. One of this tools is “least connection algorithm”. It takes into account the number of connections supported by the servers at the current time. Each next question is sent to the server with the fewest active connections. Load balancers are used for all types of servers. They will allow redirecting data between replicated nodes, besides the duplication of facilities helps to increase reliability. More than that our systems exchanging data with outside API.

The web server is used to work with managers, processes requests, generates dynamic pages, communicates with the server application layer. The application server is responsible for most of the business logic (that’s why only this element is connected to DBMS), within it are the components that allow the system to process queues, booking requests, registration and authentication. The application server accesses the Google Maps when you need to put a mark on the map.

The mobile device exchanges data with Google Maps in order to indicate the person's geolocation on the map.

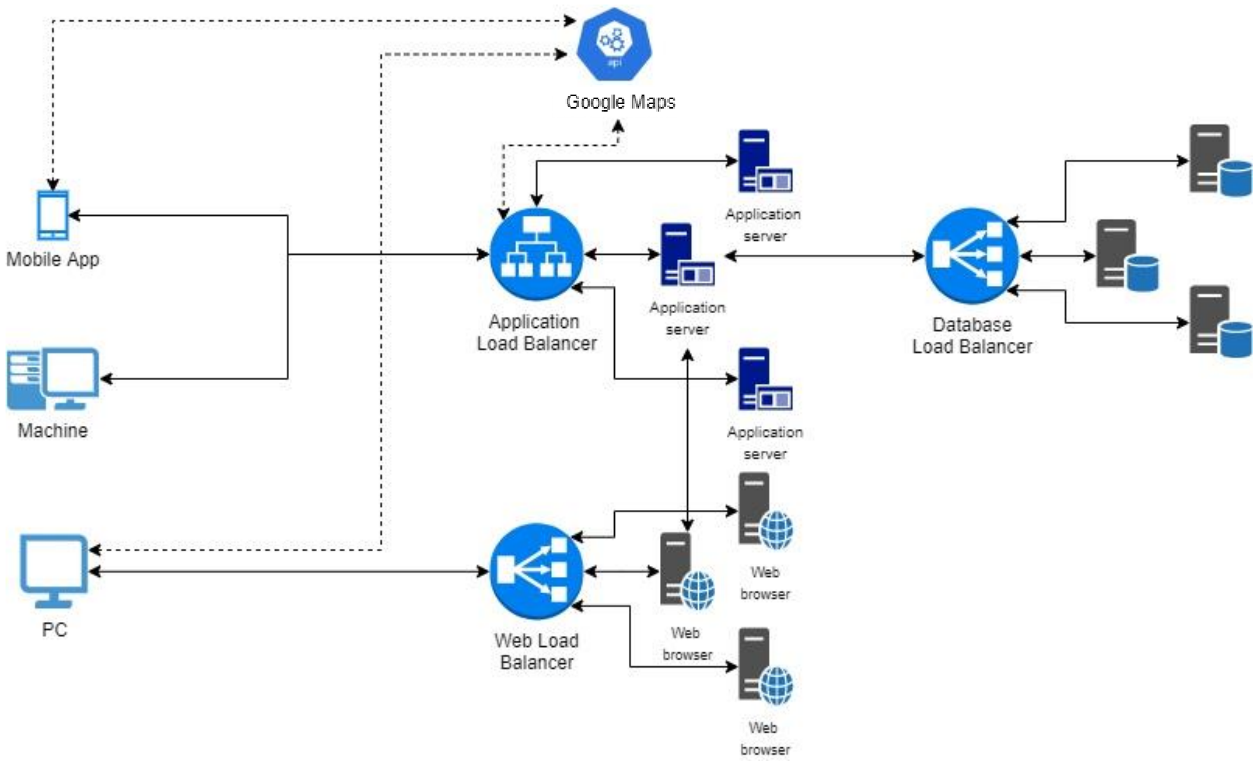


Figure 2 – System architecture

2.2 Component view

The component view includes all the components which representing different executors of business logic of the system and interactions between them. The ports that represent the external interface exposed by components are shown only among different subsystems for the sake of simplicity. In the diagram only the application server subsystem is analyzed in detail because it is the core component of the system: it contains the business logic (Application layer). The other components of the Presentation layer, the Data access layer and the web server are represented (through their software components only) just to represent their interactions with the application server.

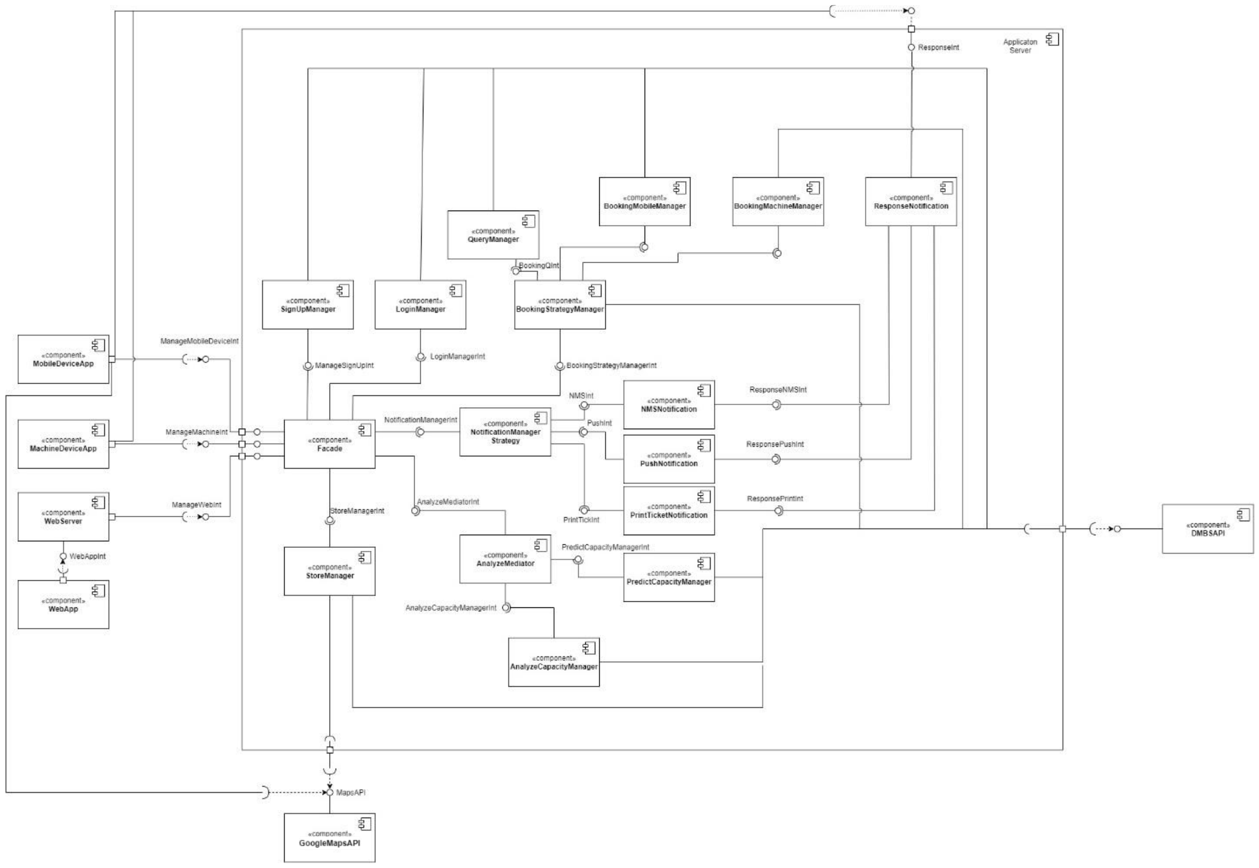

Applicaton Server
ResponseInt
«component» BookingMobileManager
«component» BookingMachineManager
«component» ResponseNotification
«component» QueryManager
BookingQInt
«component» SignUpManager
«component» LoginManager
«component» BookingStrategyManager
«component» MobileDeviceApp
ManageMobileDeviceInt
ManageSignUpInt
LoginManagerInt
BookingStrategyManagerInt
NMSInt
«component» NMSNotification
ResponseNMSInt
«component» MachineDeviceApp
ManageMachineInt
«component» Facade
NotificationManagerInt
«component» NotificationManager Strategy
PushInt
«component» PushNotification
ResponsePushInt
«component» WebServer
ManageWebInt
AnalyzeMediatorInt
StoreManagerInt
PrintTickInt
«component» PrintTicketNotification
ResponsePrintInt
«component» DMBSAPI
WebAppInt
«component» WebApp
«component» StoreManager
«component» AnalyzeMediator
PredictCapacityManagerInt
«component» PredictCapacityManager
AnalyzeCapacityManagerInt
«component» AnalyzeCapacityManager
MapsAPI
«component» GoogleMapsAPI

Figure 3 – component diagram

The components' functions contained in the 'ApplicationServer' are described in the following:

Facade: implementation of the facade pattern. The facade pattern is a structural design pattern that allows you to hide the complexity of a system by reducing all possible external calls to a single object that delegates them to the corresponding objects in the system. The facade is partitioned according to the type of component it has to interact with because the functionalities offered are quite different among them. The MobileAppFacade handles the interaction with the users' mobile app: it allows them to sign up, log in, book their visits to selected shops, and cancel them. The MachineFacade is indeed concerned with messages coming from the users who use machines near stores. Finally, the WebAppFacade offers the interface for the interaction with third parties' web application: it has additional functionalities, which allow privileged users to manage the store and analyze its attendance in different periods of time. A more detailed view of the Facade component to show the described partitioning is provided below.

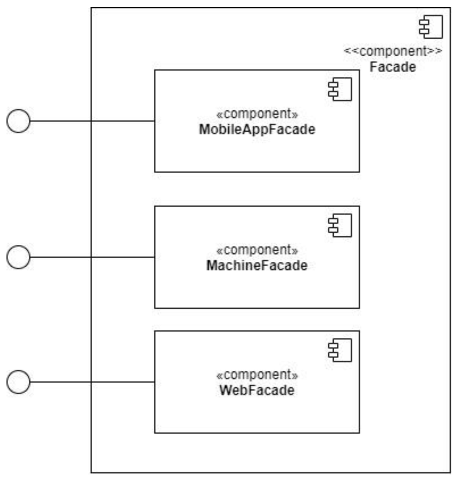


Figure 4 – Detailed view of the Facade component and its provided interfaces

The component MobileAppFacade includes interfaces for next components: ManageSignUpInt, LoginManagerInt, BookingStategyManagerInt, NotificationManagerInt.

The component MachineFacade includes interfaces for next components: ManageSignUpInt, LoginManagerInt, BookingStategyManagerInt, NotificationManagerInt.

The component WebFacade gives the manager of the shop access to next components: ManageSignUpInt, LoginManagerInt, BookingStategyManagerInt, NotificationManagerInt, AnalyzeMediatorInt, StoreManagerInt.

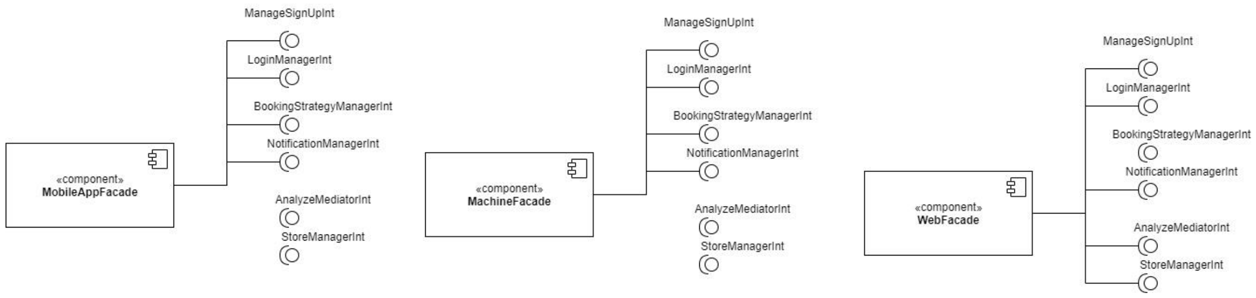


Figure 5 – Detailed view of the interfaces exploited by the Facade components

- **SignUpManager:** provides all functionalities for users to register to CLup. It has to interact with DBMS to store data about the registration and performing controls about the chosen username and password.
- **BookingStrategyManager:** it is the implementation of pattern strategy, in our case instantiates two components **BookingMachineManager** and **BookingMobileManager**, each of which is used for handling requests from its type of devices. Also, **BookingStrategyManager** is connected with **QueryManager** to insert different types of messages into queues.
- **BookingMachineManager**: includes functionalities to process incoming requests from the machine.
- **BookingMobileManager:** includes functionalities to process incoming requests from mobile devices.
- **QueryManager**: obtaining messages for structuring them in a queue with respect to their priority.
- **StoreManager**: services for adding new shop with its parameters (address, area, departments, etc) to database or editing existing. Only privileged user (staff of shop) has access to this component.
- **AnalyzeMediator**: pattern mediator implemented to the system for providing interaction between **AnalyzeCapacityManager** and **PredictCapacityManager**.
- **PredictCapacityManager**: the main part of this component is a linear model for predicting the number of customers in the shop for certain periods of time.
- **AnalyzeCapacityManager**: collecting and analyzing data of visits, to provide **PredictCapacityManager** with precise data for prediction.
- **NotificationStrategyManager**: the system has several types of notification and pattern Strategy binds all of them for convenient control.
- **NMSNotification**: this component serves for handling SMS.
- **PushNotification**: this component serves for handling push notifications.
- **PrintTicketNotification**: these component responses for notification, which are delivered to Machines near shops to print tickets.
- **DBMSAPI**: the classical tools to communicate with databases. In this case, the component represents a connection with the non-relational database MongoDB.
- **ResponseNotification**: is responsible for delivering all types of notifications to all end devices.
- **GoogleMap**: collection of API from Google company, among which there are tools for obtaining geolocation, displaying icons from the application database, calculating the path between point.

2.3 Deployment View

In the following image, the Clup deployment diagram is represented: it shows the execution architecture of the system and represents the distribution (deployment) of software artifacts to deployment targets (nodes). A deployment diagram is a type of UML diagram that shows the execution architecture of a system, including nodes such as hardware or software runtimes, and their middleware. Connection in the representation layer between smartphone and client application is provided by any modern operating system (IOS, Android, etc.). The server part is presented as cloud communications platform Twilio, which allows software developers to programmatically make phone verification, and perform other communication functions using its web service APIs. Twilio intended to process requests from mobile devices and machines installed near shops. The second version works on NGINX also installed on Red Hat Enterprise Linux 8 performing handling requests from personal computers. Two versions of the server application are connected by Java Remote Method Invocation(RMI). The technical implementation of the Data layer is provided by Airtable cloud-based service which is responsible for the grid view of the data.

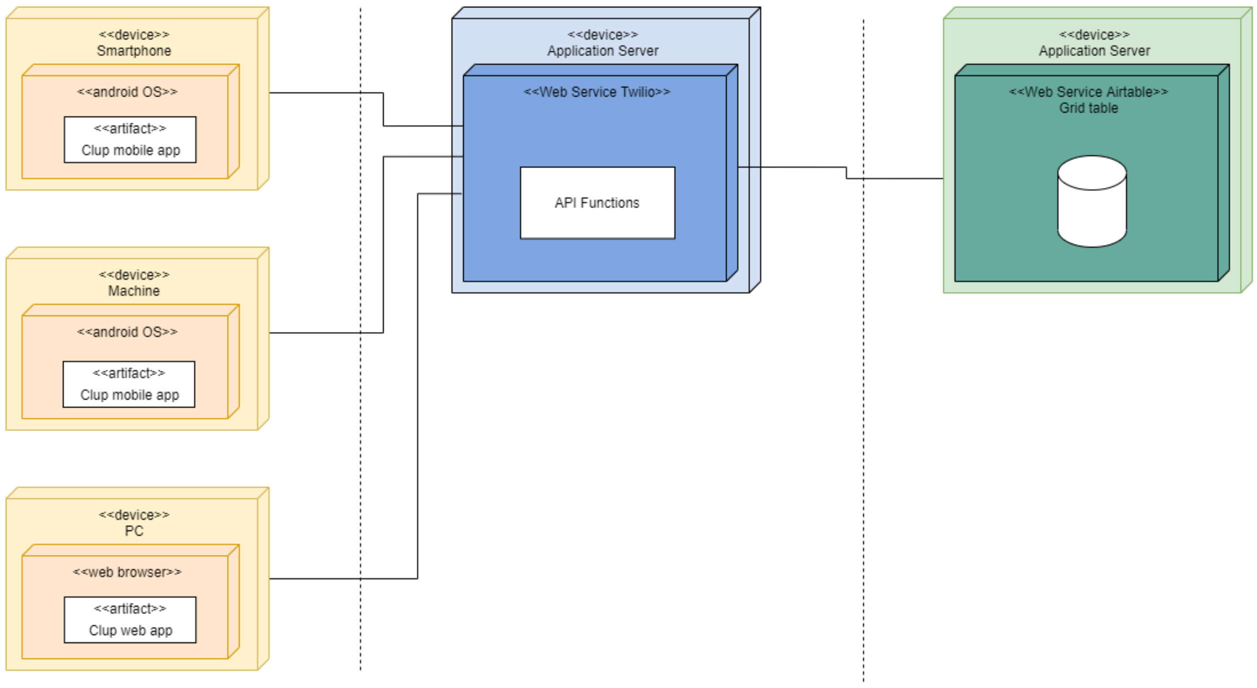


Figure 6 – Deployment diagram

In the diagram, external systems, as well as some other components are not represented to focus only on the components that host the core functionalities of the application or on the components for which the deployment is effectively executed The three tiers respectively contain:

Presentation tier: here the presentation logic must be deployed. Users must be provided with a mobile application on their smartphone and third parties with a web application accessible from their web browsers: the mobile application is the most comfortable way for a user to have access to the services and the web application has been chosen instead of a website because the services are most concerned in the interaction with the third party. The mobile application must be available for both Android and iOS to make it available on most of the devices. Users ask to communicate with the application server to book a visit to a shop or to cancel it. Third parties ask to communicate with the webserver to retrieve data of store attendance in different time periods and edit information about the store.

Business tier: here is deployed the application logic. The application server implements all the business logic, handles all the requests, and provides the appropriate answers for all the offered services. It is directly addressed by the mobile application and handles also some requests that are forwarded by the web server and sent by the web application in all the cases in which the webserver can’t provide information either because it doesn’t have them in its local disk or because some dynamic content has been requested.

Data-tier: here the data access must be deployed. The Airtable cloud-based service is responsible for the grid view of the data, which helps to retrieve/load information without special queries and interact with other components of the system in an easier way using Vue js for example.

2.4 Runtime View

2.4.1 Choosing the store

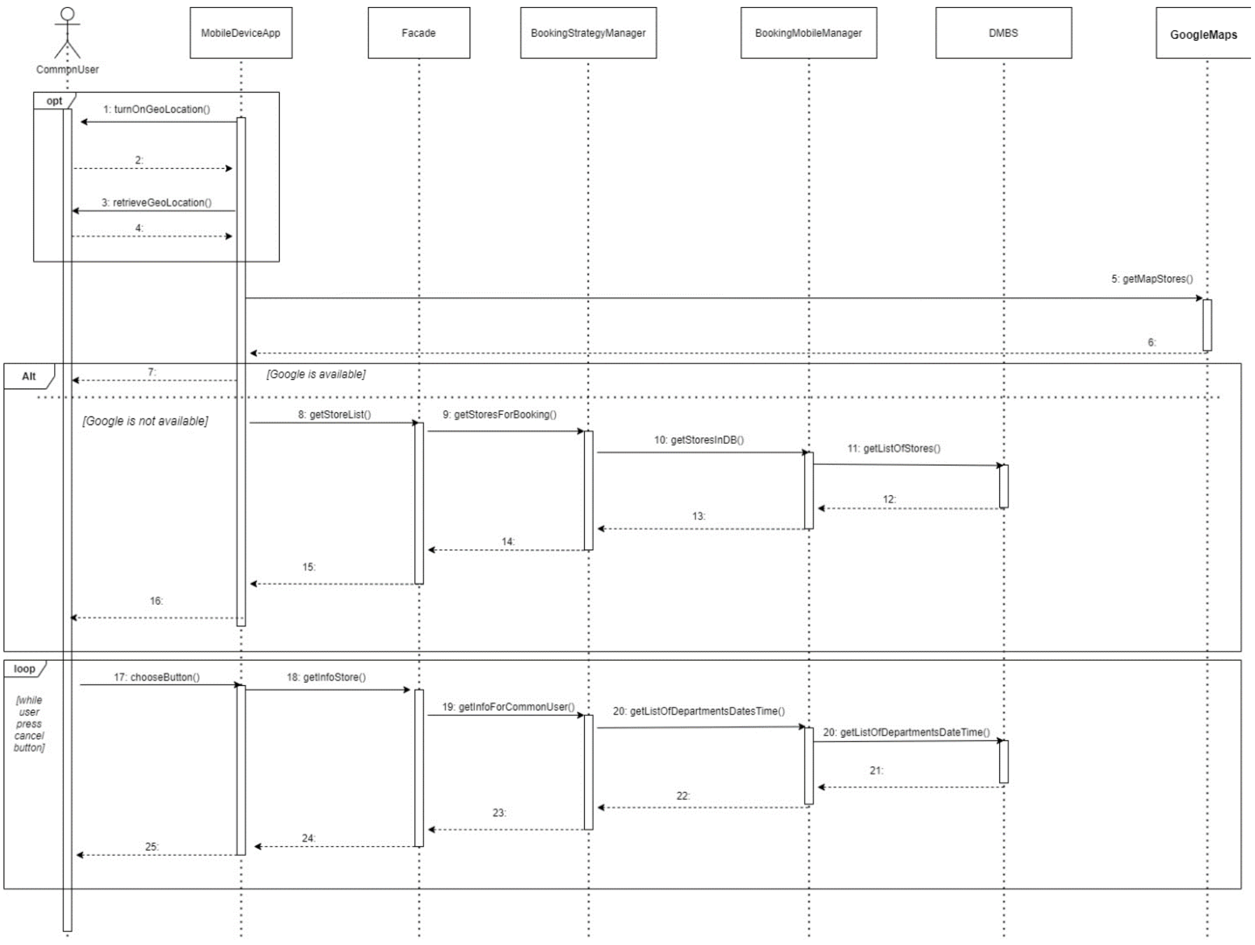


Figure 7 – Choosing the store sequence diagram

In this sequence diagram the process through which a customer chooses the store to visit. If the user gives permission to retrieve his or her location through using a geolocation system installed in the device, then requests are sent to **GoogleMap** API. After that, the user's icon is displayed on the map. This helps a customer to choose the optimal (nearest and possible for booking) store. The next step is to get the list of stores, for that purpose message uses the next components: Facade, **BookingStrategyManager**, **BookingMobileManager**, **DBMS**. In the last, all information about registered in the system shops is stored. After retrieving the data and displaying it on a mobile device. A user has the option to choose a store to visit. For providing this operation system invokes such methods as: **chooseButton**() in the interface and **getInfoStore**() in a **MobileDeviceApp**, **getInfoForCommonUser** in Facade component, **getListOfDepartmentsDateTime**() in **BookingStrategyManager** and his child component **BookingMobileManager**, finally **DBMS** sends data. After executing this pipeline of operations user can watch the information about the picked store.

2.4.2 Booking a visit by common user diagram

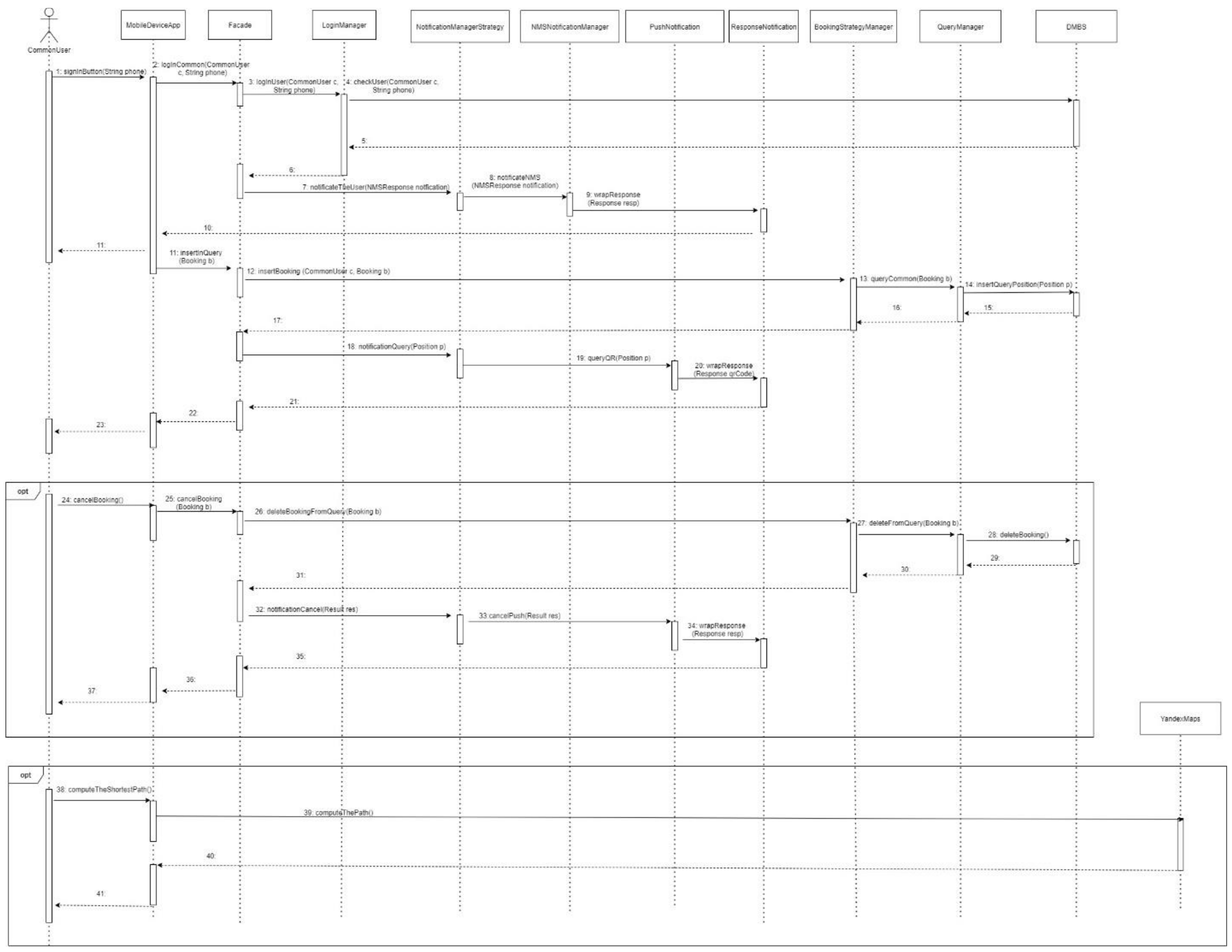

CommonUser
MobileDeviceApp
Facade
LoginManager
NotificationManagerStrategy
NMSNotificationManager
PushNotification
ResponseNotification
BookingStrategyManager
QueryManager
DMBS
1: signInButton(String phone)
2: loginCommon(CommonUser c, String phone)
3: loginUser(CommonUser c, String phone)
4: checkUser(CommonUser c, String phone)
5:
6:
7: notificateTheUser(NMSResponse notification)
8: notificateNMS (NMSResponse notification)
9: wrapResponse (Response resp)
10:
11:
11: insertInQuery (Booking b)
12: insertBooking (CommonUser c, Booking b)
13: queryCommon(Booking b)
14: insertQueryPosition(Position p)
15:
16:
17:
18: notificationQuery(Position p)
19: queryQR(Position p)
20: wrapResponse (Response qrCode)
21:
22:
23:
opt
24: cancelBooking()
25: cancelBooking (Booking b)
26: deleteBookingFromQuery(Booking b)
27: deleteFromQuery(Booking b)
28: deleteBooking()
29:
30:
31:
32: notificationCancel(Result res)
33 cancelPush(Result res)
34: wrapResponse (Response resp)
35:
36:
37:
YandexMaps
opt
38: computeTheShortestPath()
39: computeThePath()
40:
41:

Figure 8 – Booking a visit by common user sequence diagram

The functionality of booking a visit is the main goal of the application installed on a mobile device. Firstly, to get access to this possibility user should log in. For that option the system uses **LoginManager** which checks the data in the database, passing the control to, which invokes class **NMSNotificationManager** to send an SMS to the customer to authorize him in the application. After a successful login, the user can make a booking and receive the time of visit. The mobile application invokes a method **insertQuery**(Booking b), where Booking b is information about booking. The booking manager interacts with **QueryManager** to organize a queue properly. After inserting into the database user receives a message (generated by **NotificationManager**) containing a QR-code and time of a visit.

The user also has the possibility to cancel his or her booking (if plans have changed). A person presses the button for canceling the visits. It launches next operations: mobile application invokes method **cancelBooking**(Booking b), the facade forwards it to **QueryManager** where method **deleteFromQuery** erasing the recording from the queue. After a successful deletion, a user receives a notification.

A customer needs to know the time of his path to a shop to not be late. For that reason, the case of computing the shortest path is implemented. When method **computeThePath**() is called, the system sends data to **GoogleMaps API**, where the functionality of computing path is executing, the response goes back to the mobile application.

2.4.3 Booking a visit by prioritized user

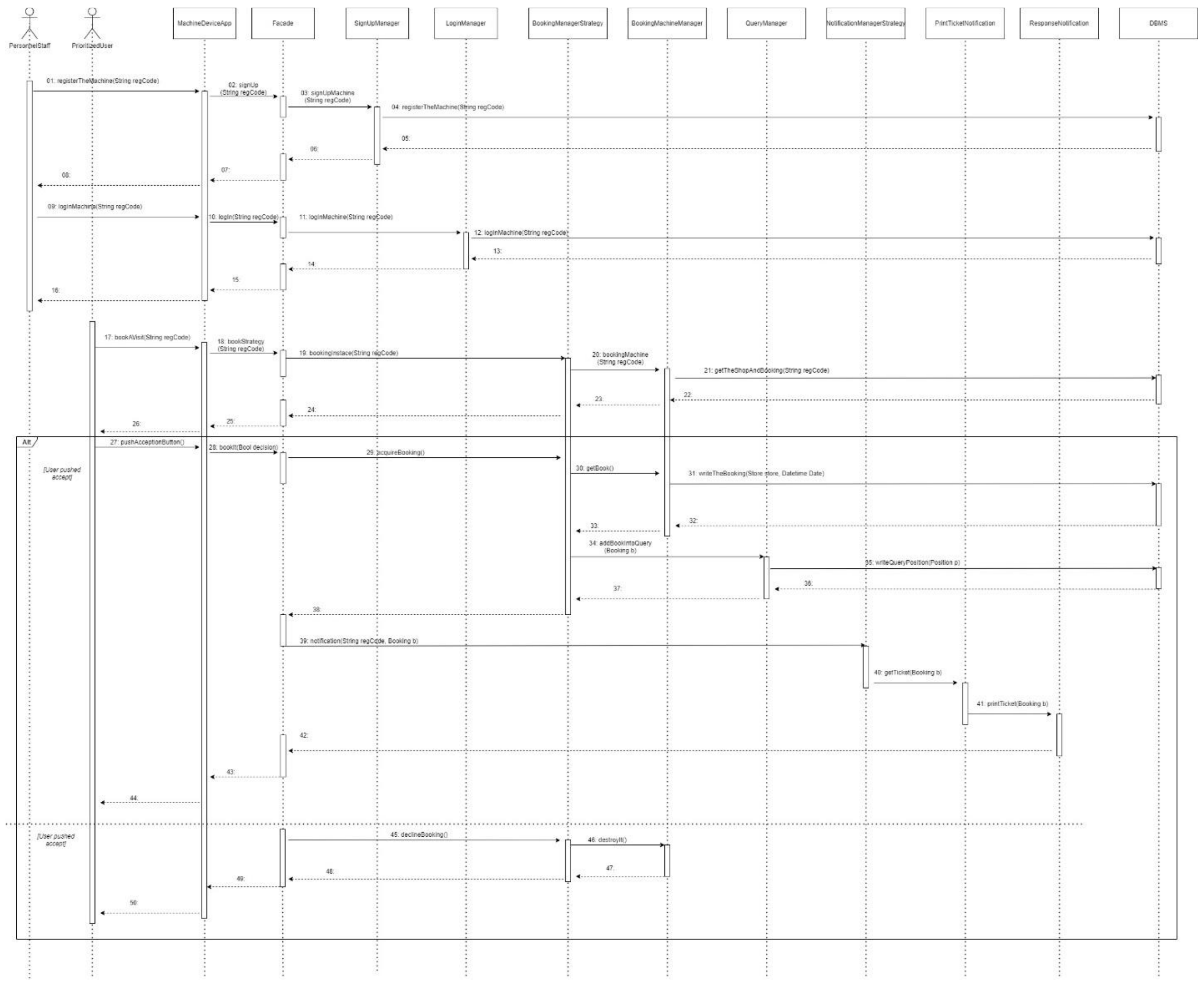

PersonnelStaff
PrioritizedUser
MachineDeviceApp
Facade
SignUpManager
LoginManager
BookingManagerStrategy
BookingMachineManager
QueryManager
NotificationManagerStrategy
PrintTicketNotification
ResponseNotification
DBMS
01: registerTheMachine(String regCode)
02: signUp (String regCode)
03: signUpMachine (String regCode)
04: registerTheMachine(String regCode)
05:
06:
07:
08:
09: loginMachine(String regCode)
10: login(String regCode)
11: loginMachine(String regCode)
12: loginMachine(String regCode)
13:
14:
15:
16:
17: bookAVisit(String regCode)
18: bookStrategy (String regCode)
19: bookingInstace(String regCode)
20: bookingMachine (String regCode)
21: getTheShopAndBooking(String regCode)
22:
23:
24:
25:
26:
Alt
27: pushAcceptionButton()
28: bookIt(Bool decision)
29: acquireBooking()
[User pushed accept]
30: getBook()
31: writeTheBooking(Store store, Datetime Date)
32:
33:
34: addBookIntoQuery (Booking b)
35: writeQueryPosition(Position p)
36:
37:
38:
39: notification(String regCode, Booking b)
40: getTicket(Booking b)
41: printTicket(Booking b)
42:
43:
44:
[User pushed accept]
45: declineBooking()
46: destroyIt()
47:
48:
49:
50:

Figure 9 – Booking a visit by prioritized user diagram sequence diagram

The booking by prioritized user differs from the common one in several points. First, the prioritized user can not choose the time, a person gets the closest available time. Second, the prioritized user makes the booking from a machine, which is located near stores. Third, instead of the message the user of the machine gets a ticket with a QR-code and visiting time.

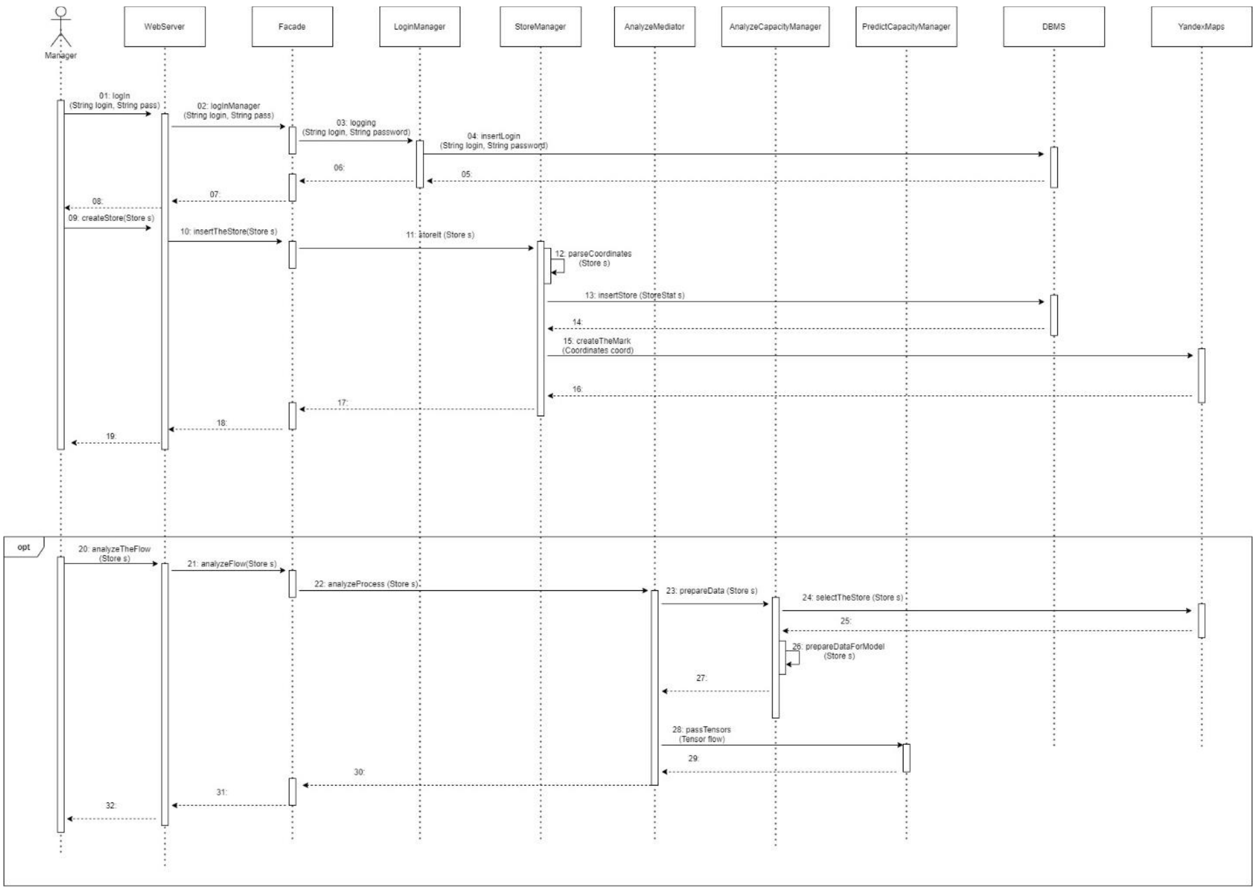
For usage, the machine should be registered in the system by store stuff (agent **PersonnelStaff** in the scheme). Registration code which is sent by them goes into **SingUpManager**. It performs the necessary operations and writes in database information about the machine. If options for any reason are erasing from the machine, a person of a store can log in to the machine with a registration code.

When a prioritized user is intended to book a visit with the help of a machine he or she comes to display and chooses the option “book a visit”. The process is going like in the previous case with the common users, only priority for entering a queue is higher. After defining a place in the queue, the system estimates the time of waiting and shows it to the user. There are two scenarios:

If a customer consents with this time he pushes the button accept and receives a ticket with QR-code and time, information is writing in the database.

If a customer declines the booking, nothing is inserted into a queue and database.

2.4.4 Manager work



Manager
WebServer
Facade
LoginManager
StoreManager
AnalyzeMediator
AnalyzeCapacityManager
PredictCapacityManager
DBMS
YandexMaps
01: login (String login, String pass)
02: loginManager (String login, String pass)
03: logging (String login, String password)
04: insertLogin (String login, String password)
05:
06:
07:
08:
09: createStore(Store s)
10: insertTheStore(Store s)
11: storeIt (Store s)
12: parseCoordinates (Store s)
13: insertStore (StoreStat s)
14:
15: createTheMark (Coordinates coord)
16:
17:
18:
19:
opt
20: analyzeTheFlow (Store s)
21: analyzeFlow(Store s)
22: analyzeProcess (Store s)
23: prepareData (Store s)
24: selectTheStore (Store s)
25:
26: prepareDataForModel (Store s)
27:
28: passTensors (Tensor flow)
29:
30:
31:
32:

Figure 10 – Manager work sequence diagram

A manager logs in to the system with of **LoginManager** component. After successful authentication, he or she can create write information about his store in the database (area, departments, coordinates, etc.). The pipeline of creating the record about the store is next: agent Manager invokes **createStore**(Store s), **WebApplication** invokes **insertTheStore**(), Facade passes the control to **StoreManager**, where coordinates are retrieving from message by parsing it, insertion in DBMS happens and the mark denoting the shop on the map is adding to **GoogleMaps**.

Also, the system provides the functionality of analyzing human flow in the store, to see the “busiest” hours. The pipeline of analyzing the flow in the store is next: agent Manager invokes **analyzeTheStore**(Store s), **WebApplication** invokes **analyzeFlow**(), Facade passes the control to **AnalyzeMediator**, which firstly gives control to **AnalyzeCapacityManager** (responsible for retrieving and preparing data), then transfers data to **PredictCapacityManager**, where the linear model is used for computations.

2.5 Component Interfaces

The diagram below shows the concept of interfaces in the perspective of implementing classes. Each of the interfaces contains a list of methods that implement logic within their component.

It should be clarified that the presented methods in the component interface diagram describe a general concept it may not coincide with what the developers will create. They can either be supplemented or completely excluded. Clarify the main compenents in more detail:

ManageSignUpInt: describes the registration process as a store personnel and machines used.

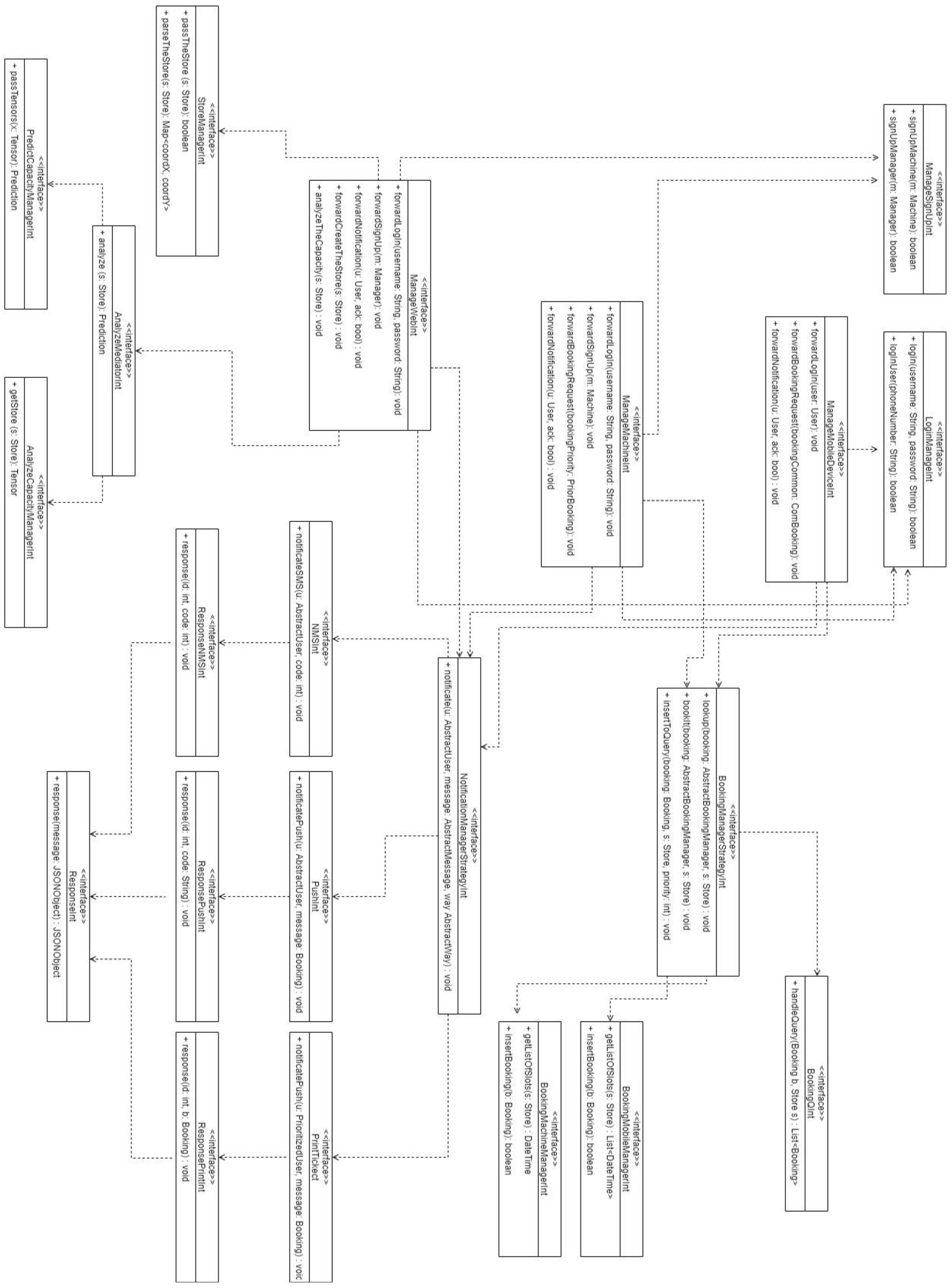
LogInManageInt: takes the role of authorization, both methods return a boolean value that reduces successful completion, *LogInUser* is executed for ordinary users, *logIn* - for store managers.

ManageMobileDeviceInt: responsible for incoming requests from the customizer, including login, booking, notification.

AnalyzeMediatorInt: takes into account the capacity by department, which subsequently affects the calculation of the queue, preventing crowding. *AnalyzeCapacityManagerint* returns the store under investigation, PredictCapacityManager deals with the technical part of the calculation.

NotificationManagerStrategyInt: It is not difficult to guess, like the component itself, like all inherited responsible for the alert, whether it is a physical printed coupon or push notification.

BookingManagerStrategy: performs the role of the central booking link. Getting free slots is carried out through the *BookingMobileManager* or *BookingMachineManager* depending on the booking device, additionally using bookingqing for direct insertion.



<<interface>> ManageSignUpInt
+ signUpMachine(m: Machine): boolean
+ signUpManager(m: Manager): boolean
<<interface>> LoginManageInt
+ login(username: String, password: String): boolean
+ loginUser(phoneNumber: String): boolean
<<interface>> ManageMobileDeviceInt
+ forwardLogIn(user: User): void
+ forwardBookingRequest(bookingCommon: ComBooking): void
+ forwardNotification(u: User, ack: bool) : void
<<interface>> ManageMachineInt
+ forwardLogIn(username: String, password: String): void
+ forwardSignUp(m: Machine): void
+ forwardBookingRequest(bookingPriority: PriorBooking): void
+ forwardNotification(u: User, ack: bool) : void
<<interface>> ManageWebInt
+ forwardLogIn(username: String, password: String): void
+ forwardSignUp(m: Manager): void
+ forwardNotification(u: User, ack: bool) : void
+ forwardCreateTheStore(s: Store) : void
+ analyzeTheCapacity(s: Store) : void
<<interface>> StoreManagerInt
+ passTheStore (s: Store): boolean
+ parseTheStore(s: Store): Map<coordX, coordY>
<<interface>> AnalyzeMediatorInt
+ analyze (s: Store): Prediction
<<interface>> PredictCapacityManagerInt
+ passTensors(x: Tensor): Prediction
<<interface>> AnalyzeCapacityManagerInt
+ getStore (s: Store): Tensor
<<interface>> BookingManagerStrategyInt
+ lookup(booking: AbstractBookingManager, s: Store) : void
+ bookIt(booking: AbstractBookingManager, s: Store): void
+ insertToQuery(booking: Booking, s: Store, priority: int) : void
<<interface>> BookingQInt
+ handleQuery(Booking b, Store s) : List<Booking>
<<interface>> BookingMobileManagerInt
+ getListOfSlots(s: Store) : List<DateTime>
+ insertBooking(b: Booking): boolean
<<interface>> BookingMachineManagerInt
+ getListOfSlots(s: Store) : DateTime
+ insertBooking(b: Booking): boolean
<<interface>> NotificationManagerStrategyInt
+ notificate(u: AbstractUser, message: AbstractMessage, way AbstractWay) : void
<<interface>> NMSInt
+ notificateSMS(u: AbstractUser, code: int) : void
<<interface>> PushInt
+ notificatePush(u: AbstractUser, message: Booking) : void
<<interface>> PrintTicket
+ notificatePush(u: PrioritizedUser, message: Booking) : voic
<<interface>> ResponseNMSInt
+ response(id: int, code: int) : void
<<interface>> ResponsePushInt
+ response(id: int, code: String) : void
<<interface>> ResponsePrintInt
+ response(id: int, b: Booking) : void
<<interface>> ResponseInt
+ response(message: JSONObject) : JSONObject

Figure 11 – CLup: Component Interfaces

The detailed architecture of the program and its interaction with third-party API is described below in the Class diagram.

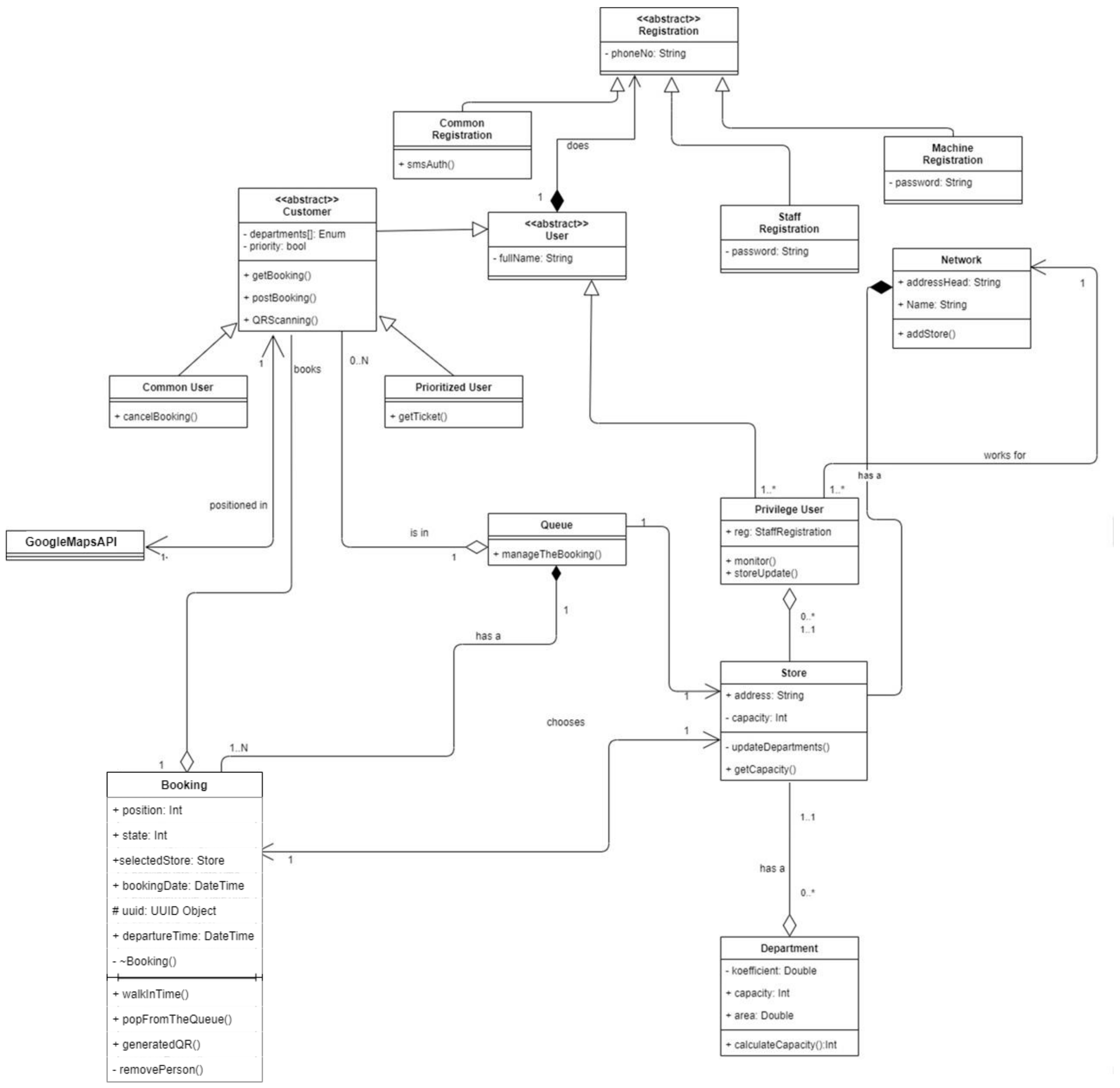


Figure 12 – Class diagram

As can be seen in the screenshot, the entry point of the program can be considered as the *Registration*, which is divided into two branches: the end-user registration and staff registration. Class *Customer*, after inheriting the qualities of the User Class, also can be divided into two parts: the one group of people who will use the application - *CommonUser*, with the consequent possibility of canceling reservations online, and the second group *PrioritizedUser*, which will receive its ticket to the queue upon arrival at the store. The *Customer* class, which could be from 0 to N *Queue* class in general, also uses the *GeoController* class, which provides functionality for tracking the user position and building a route to the selected store. The *GoogleAPI* library is responsible for the rest of the map functionality. The *Customer* class contains a key *getBooking* method that implements the logic of introducing a new user to the queue, storing more detailed information, such as the selected store and the capacity of its departments, the user's state (going to the store, making purchases, leaving the store), and generating a QR code that is necessary needed for going to the store. User's position data is transmitted via a mobile device; therefore, it is important to grant the appropriate rights to use the app. The *Queue* class stores an array of all bookings for the store. The purpose of the Store and *Department* classes is to prevent congestion and create a more efficient distribution of incoming people, to do this, we need to have additional information about the store itself that the owner provides. The *PriviligeUser* class is the store staff that can access the app and its advanced settings. The *Network* class groups stores of a single registered trademark. It is worth noting that the above functionality is performed on the server-side.

2.6 Selected Architecture Style and Patterns

Due to the constant interaction with the customer on the principle of "request-response", it was decided to choose a three-tier application architecture. It is a modular client-server architecture that consists of a presentation tier, an application tier, and a data tier. The data layer stores information, the application layer handles the logic, and the presentation layer is a graphical user interface (GUI) that interacts with the other two layers. These three levels are logical, not physical, and can run on the same physical server or on different machines.

To carry out "communication" with a user, certain rules are needed, namely protocols that would provide additional encryption. HTTP should be immediately chosen - a widespread data transfer protocol originally intended for the transfer of hypertext documents (that is, documents that may contain links that allow you to organize a transition to other documents). The HTTP protocol assumes the use of a client-server data transfer structure. The client application generates a request and sends it to the server, after which the server software processes this request, generates a response and sends it back to the client. The client application can then continue to send other requests, which will be handled in a similar manner. A task that is traditionally solved using the HTTP protocol is the exchange of data between a user application that accesses web resources (usually a web browser) and a web server. At the moment, it is thanks to the HTTP protocol that the work of the World Wide Web is ensured.

To support user-server interaction, HTTP requires a data exchange format to be specified. The choice fell on JSON.

JSON is a simple text-based way to store and transmit structured data. Once a JSON string is created, it is easy to send it to another application or elsewhere on the web because it is plain text. JSON has the following benefits:

- It is compact.
- Its sentences are easy to read and compose by both humans and computers.

XML - Quite a popular competitor to JSON. But this format is used less and less because of the complexity, redundancy of syntax in this markup language, which leads to a larger size of the XML document.

To build a program, in particular to write code, experienced developers use a design pattern. Each of the patterns have a specific purpose and addresses a typical problem.

In this project, 3 patterns were involved, namely:

- Strategy - proposes to define a family of similar algorithms, which are often changed or extended, and move them into their own classes, called strategies. Instead of the original class executing this or that algorithm itself, it will act as a context, referring to one of the strategies and delegating the work to it. To change the algorithm, you just need to substitute another strategy object into the context.

- Mediator - forces objects to communicate not directly with each other, but through a separate intermediary object that knows who needs to redirect a particular request. Thanks to this, the components of the system will depend only on the intermediary and not on dozens of other components.

- Facade - it is a simple interface for working with a complex subsystem containing many classes. it hides the complexity of a system by providing a simplified interface for interacting with it.

Not without calls to a third-party service such as GoogleMaps. There are certain rules for interaction with such calls. In our case, the REST architecture is used, and the API of GoogleMaps:

- REST (Representational State Transfer) - passing view state. It is an architectural style of interaction between components of a distributed system in a computer network. Simply put, REST defines a style of communication (data exchange) between different system components, each of which can be physically located in different places.

- API - Application programming interface, behind these words is a whole set of tools that allow applications to interact with each other.

2.6.1 Model View Controller (MVC)

The basis of the program was decided to choose Model-View-Controller (MVC). The main reason for making the decision was the time-test of this pattern in the aspect of scalable projects.

In MVC, application components are usually divided into three aspects: Model, View, and Controller.

Each of the aspects is responsible for its own logic, namely:

• Controller: Processes user actions, checks the received data and passes them to the model. It's kind of a layer between the model and the view. All business logic, that is, a set of rules, principles, dependencies of object behavior, is located right here. For example, the customer wants to see the current queue to the store, first a request is sent from the view to the controller, and the latter, in turn, will work through the code to read the current queue and send back a request to the model. The model returns the required data back to the controller, which will display the result requested by the customer.

• Model: Receives data from the controller, performs the necessary operations and transfers them to the view. This component is responsible for data and also defines the structure of the application. For example, if you are building a To-Do application, the model component code will define a list of tasks and individual tasks.

• View: Receives data from the model and displays it for the user. View components are implemented via mobile app for the users and web browser interface for shop managers to access their private personal area.

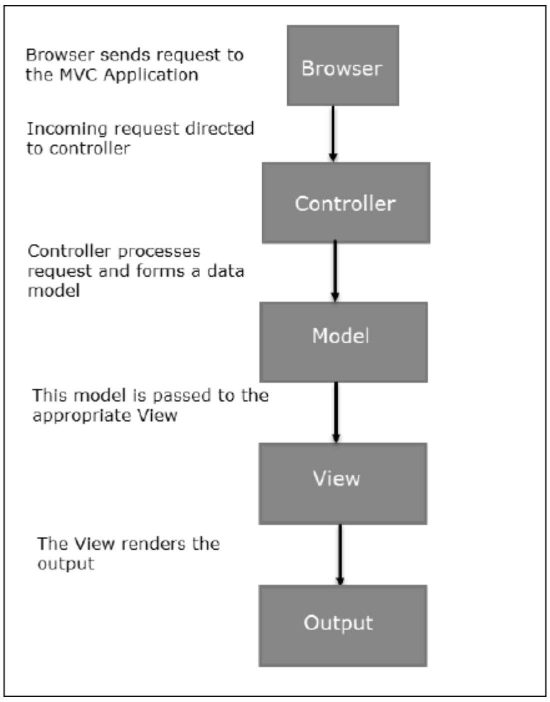

Browser sends request to the MVC Application
Browser
Incoming request directed to controller
Controller
Controller processes request and forms a data model
Model
This model is passed to the appropriate View
View
The View renders the output
Output

Figure 13 – CLup: MVC demostration

2.7 Other Design Decisions

This section is devoted to the conditionality of choosing a database for this project.
After reviewing the main available solutions, Airtable cloud-based tool takes over the role of the database The Airtable features a centralized database that enables a team to work together seamlessly for increased efficiency. It further helps them collaborate through its familiar spreadsheet-like interface. What users see is software that acts like a spreadsheet with its rows and columns, as well as a system that functions as a unified location for records, content, projects, and ideas. Airtable allows users to access data and migrate it to other databases without compromising its integrity. Users can also make use of file attachments, rich media and text, photos, notes, barcodes, and checkboxes. So the service is not limited to one data storage.
Let's make up the main set of advantages:

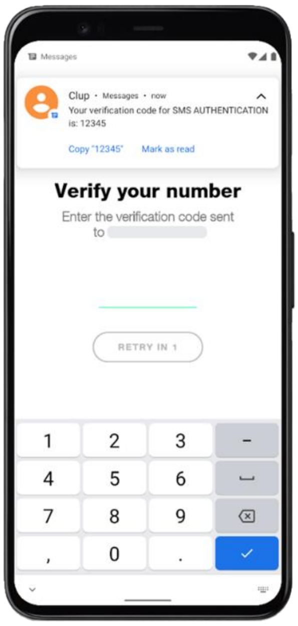
- Complete software. Airtable is basically software that works like a spreadsheet. Though there are cons about this, it is also capable of attaching files. There are drop-down options, as well as checkboxes among others. In fact, it also has various options that Excel does not let its users enjoy.
- Ease of use. Users of Airtable find themselves easily hiding/unhiding fields and even moving them. Using filters is easy and blocks can be used for bulk editing.
- It has an array of templates. One of the strengths of Airtable is its array of templates and layouts to choose from. There is enough to choose from for anyone who prefers to use the built-in ones. However, those who are proficient enough can also opt to create their own from scratch.
- Great view options. Every user has his own view options and preferences, depending on his work. Airtable allows users to go from grid appearance to form appearance. This means that they can view the project as a typical spreadsheet and change that to something that is similar to a database.
- Easily tracks project progress. Even though Airtable acts like a spreadsheet, it is still a project management tool. Therefore, one of its strengths is letting its users track the progress of their projects. It lets them see every department—sales, marketing, and communication, and the tasks they are doing.

3 User Interface Design

3.1 Mockups

To represent the approximate concept and interface of the program, the first mockups are shown below.

3.1.1 User verification by sending an SMS to the specified number



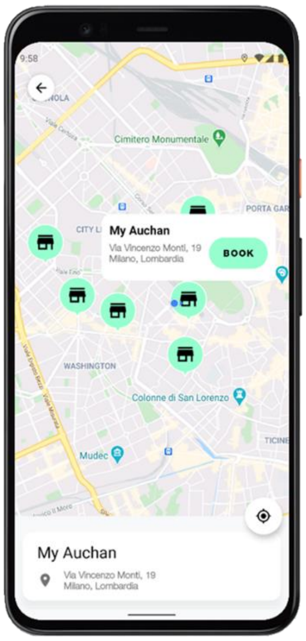
Your verification code for SMS AUTHENTICATION is: 12345

Verify your number

Enter the verification code sent to

Figure 14 – CLup: customer verification

3.1.2 Displaying and selecting the nearst available user stores



My Auchan
Via Vincenzo Monti, 19
Milano, Lombardia

My Auchan
Via Vincenzo Monti, 19
Milano, Lombardia

Figure 15 – CLup: store choosing

3.1.3 Selecting the appropriate store visit dates and time

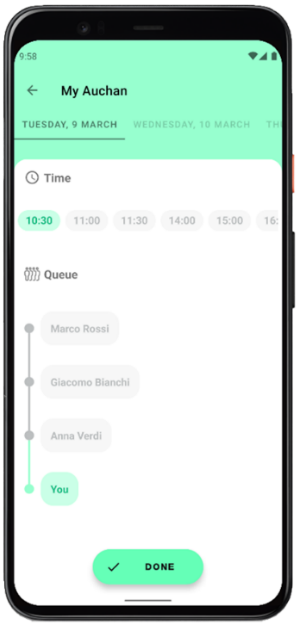
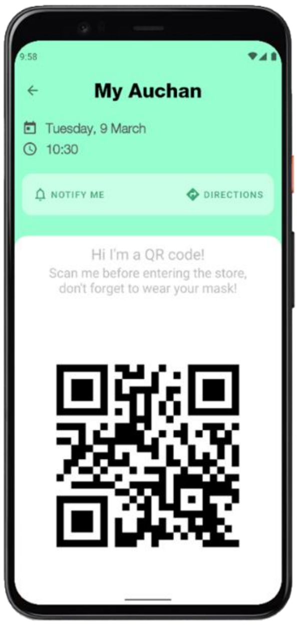


Figure 16 – CLup: date/time choosing

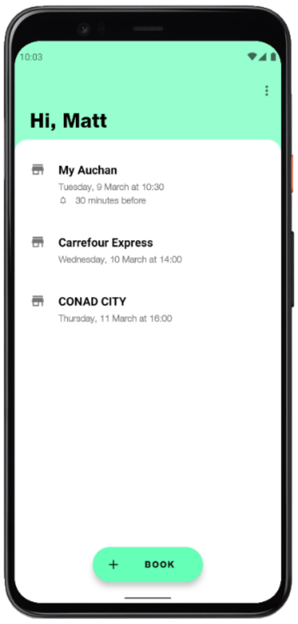
3.1.4 Generate the QR code required for entering



Hi I'm a QR code!
Scan me before entering the store,
don't forget to wear your mask!

Figure 17 – CLup: QR code generation

3.1.5 Displaying actual reservations at a glance


10:03

Hi, Matt

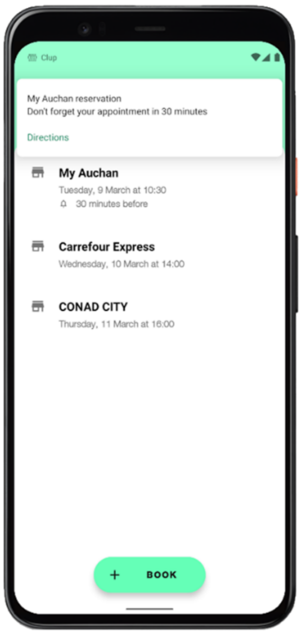
My Auchan
Tuesday, 9 March at 10:30
30 minutes before

Carrefour Express
Wednesday, 10 March at 14:00

CONAD CITY
Thursday, 11 March at 16:00

Figure 18 – CLup: time choosing

3.1.6 Displaying reservation notifications in advance


Clup

My Auchan reservation
Don't forget your appointment in 30 minutes

Directions

My Auchan
Tuesday, 9 March at 10:30
30 minutes before

Carrefour Express
Wednesday, 10 March at 14:00

CONAD CITY
Thursday, 11 March at 16:00

Figure 19 – CLup: Booking notification

3.1.7 Displaying and selecting the nearst available user stores (dark mode demonstration)

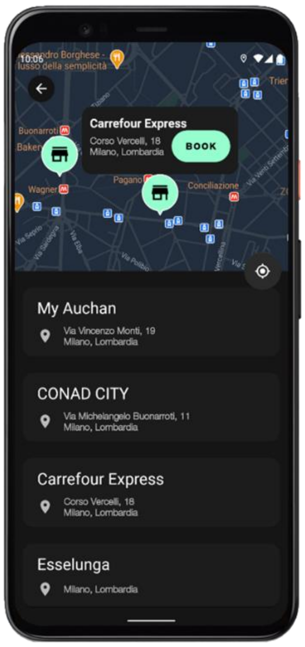


Figure 20 – CLup: Booking notification (dark mode)

3.1.8 Selecting the appropriate store visit dates and time (dark mode demonstration)

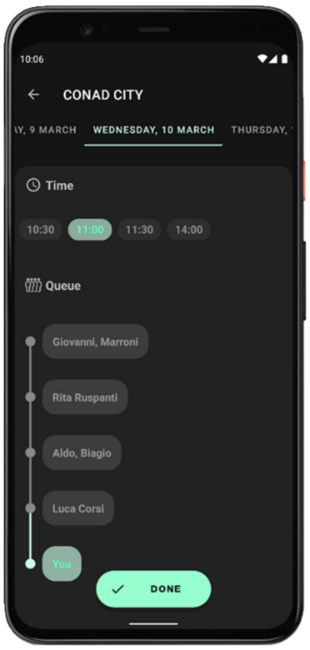


Figure 21 – CLup: date/time choosing (dark mode)

4 Requirements Traceability

This section introduces the matching between the goals and requirements with the elements which compose the architecture of the application.

G1	**The system should alert taking into account the time they need to get to the shop from the place they currently are**	
	R1	The system should allow users to go to the store by shortest path calculated by Google Maps.
	R2	All information about visit (arrival/departure time, the selected departments) should be analyzed by system, to estimate the time of waiting more precisely.
	Components: • **QueryManager** • **BookingStrategyManager** • **NotificationMangetStrategy** • **NMSNotification** • **PushNotification** • **PrintTicketNotification** • **GoogleMapsAPI** • **DBMSAPI**	
G2	**To give people opportunity to stay in queue remotely to avoid crowds in stores and in real queues**	
	R3	The system should send the data of the visit to the server to organize the queue correctly.
	R4	The system should provide concurrency to insert multiple requests in short period of time in correct order.
	R5	The system should balance the human flow in the store by analyzing the departments and store capacity.
	R6	The system generates QR codes by UUID, telephone number/device id
	Components: • **QueryManager** • **BookingStrategyManager** • **BookingMachineManager** • **BookingMobileManager** • **AnalyzeMediator** • **AnalyzeCapacityManager** • **PredictCapacityManager** • **NotificationManagerStrategy** • **PushNotification**	
G3	**Stores should have the possibility to hand out "tickets" on the spot**	
	R7	The system should distinguish users of mobile phones and local devices to prioritize the second ones in the queue.
	R8	QR readers should be installed at the entrance and exit of the store sending data of arrival and departure time to system.
	R9	The machine should be authorized and logged in the system by the manager of the store.
	Components: • **MachineDeviceApp** • **BookingMachineManager** • **PrintTicketNotification** • **SignUpManager** • **LoginManager**	
G4	**The system should allow customers to book a visit to the supermarket**	

	R10	The system automatically extends the queue if the user books a visit for particular date and time.
	R11	CLup service appeals to NMS to provide the authentication of the user.
	Components: • **NSMNotification** • **QueryManager** • **BookingStrategyManager**	
G5	**The system (application and “tickets” on the spot) should include alternative slots (for another day), suggest to the customer the location of the nearest “safe” store based on his location.**	
	R12	The system should provide algorithms to assess the stores human flow.
	R13	The system should process the given location and base on it provide the list of accessible stores for the customer.
	Components: • **BookingStrategyManager** • **GoogleMapsAPI** • **DBMSAPI** • **BookingMobileManager**	
G6	**The system should allow the third party to get the statistical information to perform better management of the store.**	
	R14	The staff of the store have to login into the system to obtain statistic to manage the store in a better way.
	R15	The system provides to the store owner only statistical information with embedded in the system instruments for visualization.
	Components: • **StoreManager** • **SignUpManager** • **AnalyzeCapacityManager** • **PredictCapacityManager** • **BookingStrategyManager** • **AnalyzeMediator** • **DBMSAPI**	

R1	The system should allow users to go to the store by shortest path calculated by Google Maps.
R2	All information about visit (arrival/departure time, the selected departments) should be analyzed by system.
R3	The system should send the data of the visit to the server to organize the queue correctly.
R4	The system should provide concurrency to insert multiple requests in short period of time in correct order.
R5	The system should balance the human flow in the store by analyzing the departments and store capacity.
R6	The system generates QR codes by UUID, telephone number/device id and suggested time of the visit.
R7	The system should distinguish users of mobile phones and local devices to prioritize the second ones in the queue.
R8	QR readers should be installed at the entrance and exit of the store sending data of arrival and departure time to system.
R9	The machine should be authorized and logged in the system by the manager of the store.
R10	The system automatically extends the queue if the user books a visit for particular date and time.

R11	CLup service appeals to NMS to provide the authentication of the user.
R12	The system should provide algorithms to assess the stores human flow.
R13	The system should process the given location and base on it provide the list of accessible stores for the customer.
R14	The staff of the store have to login into the system to obtain statistic to manage the store in a better way.
R15	The system provides to the store owner only statistical information with embedded in the system instruments for visualization.

5 Implementation, Integration and Test Plan

5.1 Overview

Testing is a very important stage in the development of mobile applications. The cost of a mistake in a mobile app release is high. Apps reach Google Play within a few hours, and the Appstore for several weeks. It is not known how long users will be updated. Errors cause strong negative reactions, users leave low ratings and hysterical reviews. New users seeing this do not install the application. Mobile testing is a complex process: dozens of different screen resolutions, hardware differences, several versions of operating systems, different types of Internet connection, sudden disconnections. For the reasons described above, this block is one of the most important.

The diagrams below outline the integration components. The priority component will be directed to the dependent one.

The system is divided in various subsystems:

- MobileDeviceApp
- MachineDeviceApp
- WebServerApp
- WebServerApp
- ApplicationManager
- External system: DBMS, GoogleMaps

The above subsystems will be implemented and tested using a combined approach meaning combination of Top-Down and Bottom-up principles. The choice of both of these principles is the result of the avoidance of implementation of small subsystems because of the used architecture patterns e.g. Strategy pattern and Mediator pattern where we implement the big component at first and then implementing the smaller ones: it's a scenario where top-down approach used. Every other component will be implemented using a bottom-up approach like it was shown in the following Figures

Feature	Importance for the customer	Difficulty of implementation
Sign up and Login	Low	Low
Book a visit	High	Medium
Display user position on the map	Medium	Low
Booking notification	High	Low
Compute the shortest path	Low	Low
Analyze customer flow	High	High
Normalazing customer flow	High	High

Features which are listed above are should be tested before integrating in the product. Each feature is supported by certain managers:

- **Sign up and Login**: an entry point for customers. To be sure of using this feature, you need to test the following responsible components: *Facade* including interfaces for *SignUpManager* and *LoginManager*, which are connected with *DBMS*.

- **Book a visit:** One of the core features that requires the correct operation of the *BookingManagerStrategy* interface using *QueryManager* along with *BookingMobileManager* or *BookingMachineManager* depending on user end-device.
- **Display user position on the map:** uses only *GoogleMapAPI*. Which provides geolocation via the API.
- **Booking notification**: It is good to mention the top-down principle, components i *PushNotification* and *PrintTicketNotification* depend on implementation and testing *NotificationManagerStrategy* It will also need *ResponceNotification* in addition to *BookingMobileManager* or implemented by *ResponceNotification* in addition to *BookingMobileManager* or *BookingMachineManager* depending on user end-device.
- **Compute the shortest path:** employs only *GoogleMapAPI*, which requires to be tested and implemented. The logic of the process is implemented by the maps themselves, it remains only to properly interact with external services and set up a friendly interface with the user.
- **Analyze customer flow** and **Normalazing customer flow**: This component deserves special attention and testing, because the flow of people in the store, bandwidth, moreover, the health of people will be directly dependent on it. Responsible for these features are *PredictCapacityManager* and *AnalyzeCapacityManager* interacting through *AnalyzeMediator*.

5.2 Component integration

The diagrams below outline the integration components. The priority component will be directed to the dependent one.

Integration of the internal components of the Application Server

The following components are successfully integrated.

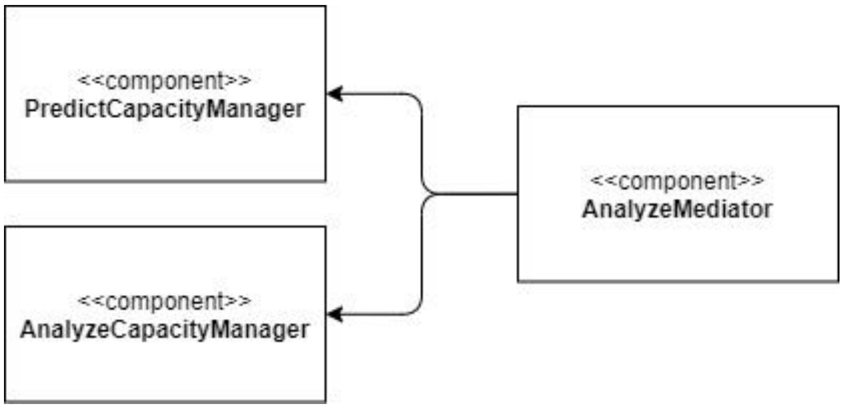


Figure 21 – AnalyzeMediator integration

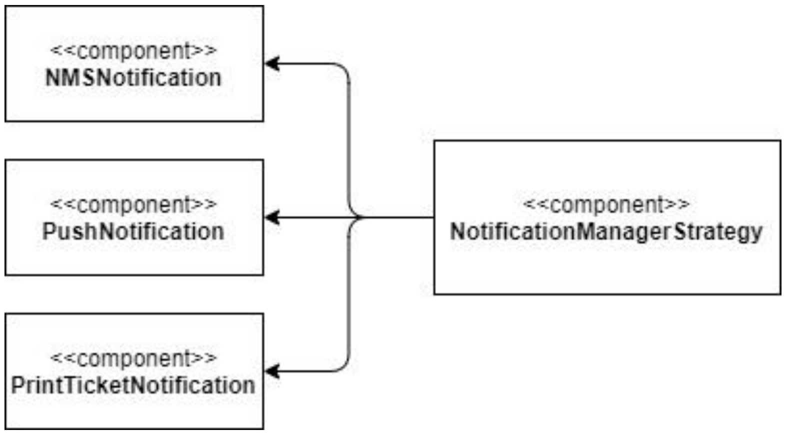


Figure 22 – NotificationManagerStrategy integration

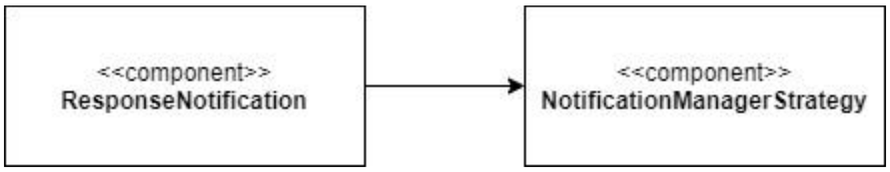


Figure 23 – ResponseNotification - NotificationManagerStrategy integration

Integration of the frontend with the backend

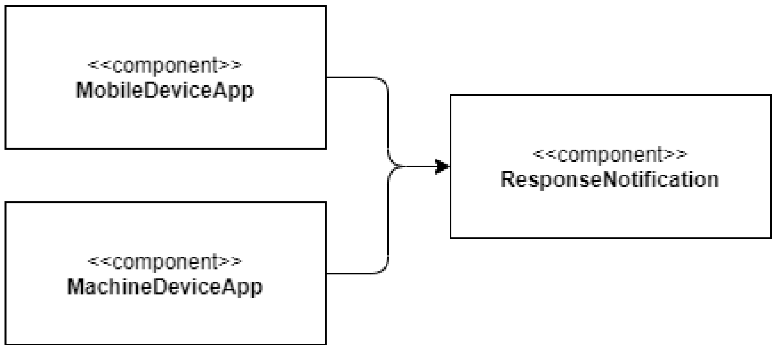


Figure 24 – Mobile/MachineDeviceApp - ResponseNotification integration

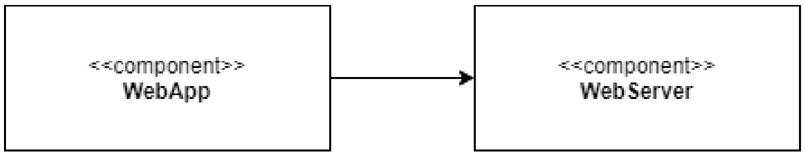


Figure 25 – WebApp - Webserver integration

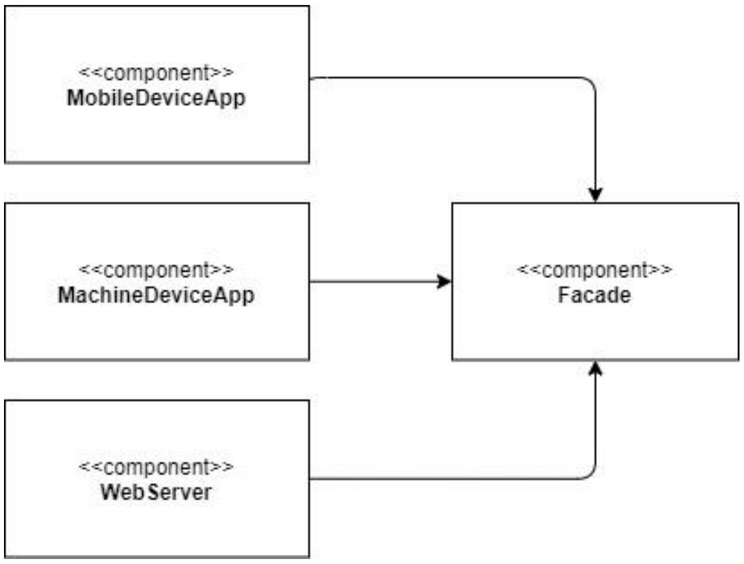


Figure 26 – Facade Integration

Integration with the external services

They are final in integration, after the main components are successfully tested

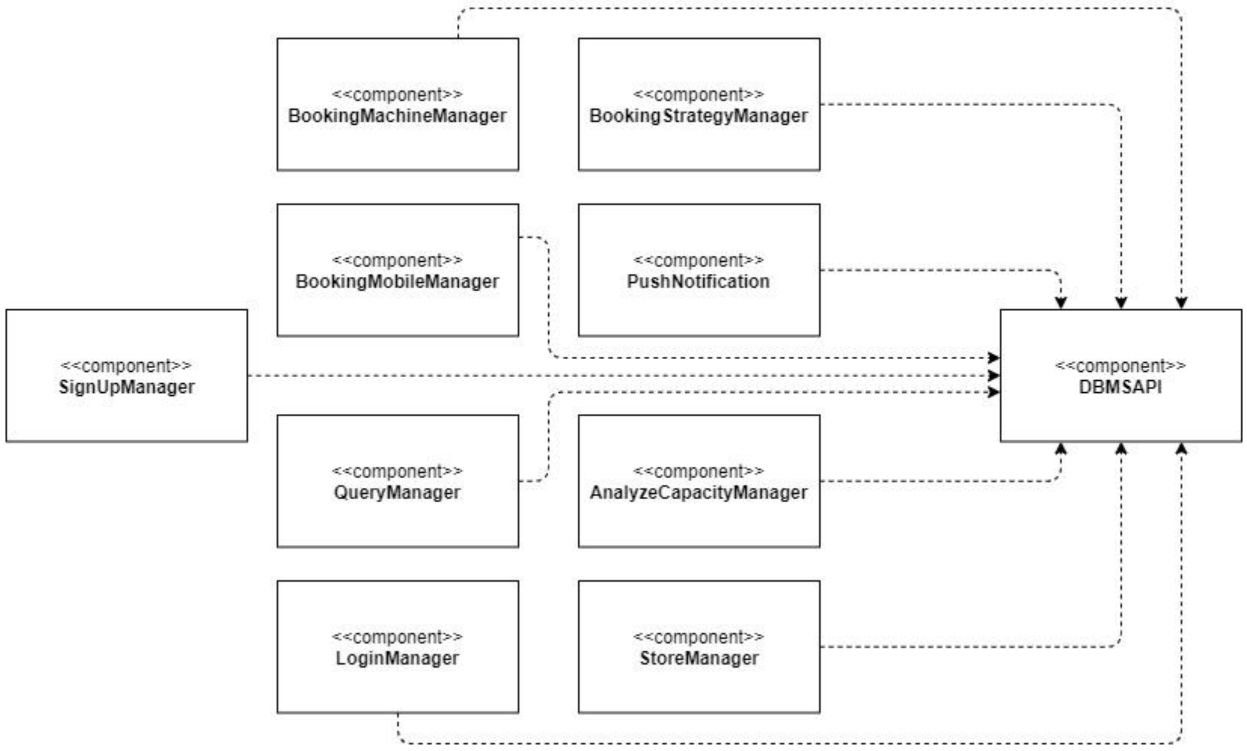


Figure 27 – DBMS API integration

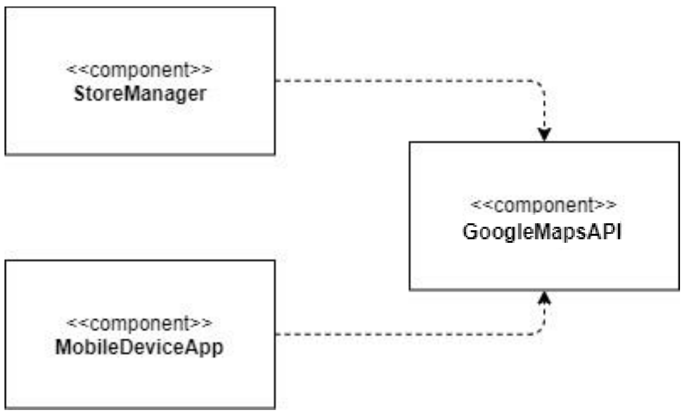


Figure 28 – Google Maps API integration

5.3 System Testing

After completing the main aspects and integration, it is necessary to test the system as a whole, taking into account all the requirements. An important point is to check all modules, making the conditions as close as possible to the real ones.

The system testing process can be divided into categories:

Functional testing: verification of compliance with the requirements.

Performance testing: identification of performance problems, if necessary, optimization of algorithms.

Stress testing: system response to failures and return to the original operating state.

Load testing: search for vulnerabilities, such as a memory leak, garbage collector problem, buffer capacity.

6 Effort Spent

6.1 Mattia Calabresi

Task	Hours
Introduction	3
Architectural Design	4
Runtime view	2
Component View	4
Requirements Traceability	3
Implementation, integration and testing	7
Documents Revision	4

6.2 Oleksandr Shchukhlyi

Task	Hours
Discussion on second part	0.5
Component diagram	3
Component interface	4
Mockups	5
Architectural styles and patterns	7
Implementation, integration and testing	3
Documents Revision	3

6.3 Alessandro Brigandì

Deployment View	4
Detailed View Facade	2
Detailed View Facade Interfaces	2
Component View	5
Runtime View	6
Implementation, integration and testing	5
Documents Revision	4

7 References

- Present diagrams were created on: https://app.diagrams.net/
- Design Patterns: Elements of Reusable Object-Oriented Software (c) Erich Gamma Ralph Johnson Richard Helm John Vlissides
- Google.Maps API https://developers.google.com/maps/documentation