POLITECNICO
MILANO 1863

Requirement analysis and specification document

Davide Li Calsi	*10613807*
Andrea Alberto Marchesi	*10577090*
Marco Petri	*10569751*

Professor: Matteo Giovanni Rossi
Academic year: 2020/2021

Contacts:

Davide Li Calsi	davide.li@mail.polimi.it
Andrea Alberto Marchesi	andreaalberto.marchesi@mail.polimi.it
Marco Petri	marco.petri@mail.polimi.it

Version 1.1 23 December 2020

Table of contents

1 Introduction

Virus SARS-Cov-2 has influenced the whole world during 2019 and 2020 and continues to influence it. Several governments across the world decided to take restriction measures to avoid the infection. The system to be should work in order to make store's customers able to line up digitally to enter a grocery store, or to make a reservation for a given time on a given day. This part of the document explains the purpose of the project, the scope, definitions and acronyms used, document's revision history, reference documents and the document structure.

1.1 Purpose

The system is aimed to be used on the Italian territory for a grocery store's chain by customers and employees of the stores. The system's purpose can be divided in two different categories:

+ **Service for customers**: a service is offered to grocery stores' clients and is realized with a digital system. The service must allow users to queue online in order to access the store until is their time of entering and to book visits to the store later than at the time of reservation;
+ **Avoiding crowds**: avoiding crowds is the aim of the system which is intended to substitute when possible the classical lining up methods (tickets and queues).

The system is intended to be easily usable and accessible almost to everyone who has access to an electronic device where external applications can be installed. No hardware design is requested and customers use their own devices. The system can be used by the user to queue and to see how much wait time there is to achieve the possibility to get in the store. These two system's characteristics are used to ensure the goal of having the minimum amount of people waiting outside the store to be able to enter on it.

The system should be able to trace customers visits and to save customers mean visit time and store it for future computations. Once the system has traced a certain number of visits of a customer, it will be able to estimate the visit duration time for it and use this estimation to compute the number of people able to access the market at that visit time. The system should trace every customer which visits the store and save the visits durations in order to be able to compute the mean time of a visit for that client.

The system should be possible to be used parallelly to a physical system of lining up at the store’s entrance for those people who cannot afford the application due to technology limitations. The system goal is to balance the digital queue with physical queue of people which cannot use the digital system. Goal of the system can then be summarized as follows:

Goal number	Goal description
1	customers can queue online without reaching the store physically
2	customers can make a reservation for a store registered in the system
3	customers can choose their preferred store by all present
4	customers must be safe during their visit at a store
5	every customer must be allowed to queue aiming to enter in a store
6	no crowds have to be present outside the store
7	customers who queue first are the first to enter the store
8	precise estimation of the waiting time must be accessible to the queuing customers
9	customers receive a notification when they need to get out in order to reach the store
10	customers cannot overload a store's queue
11	system is configurable for the needs of every store
12	customers' access must be supervised
13	customers must be helped through suggestions to choose the safest way of buying their products

1.2 Scope

The system is put on an environment composed of different entities which are part of the world: customers, store managers and checkpoint controllers. With checkpoint controllers we mean the people involved in the activity of controlling people's numbers when a number is called and a person approaches the entrance wanting to be granted the access. Here follow lists of phenomena for this system and the world related to it.

1.2.1 World phenomena

Phenomena	Controller	Description
1	W	Customer choose where to go shopping
2	W	Customer opens the application
3	W	Customer goes to the store
4	W	Customer gets access to the application for lining up
5	W	Customer takes its shopping items
6	W	Customer pays

1.2.2 Shared phenomena

Phenomena	Controller	Description
1	W	The store manager sets the maximum amount of people for a group
2	W	The store manager sets the maximum amount of people inside a certain sector
3	W	Customer enters in the (digital) queue
4	W	Customer checks the estimated waiting time
5	W	Customer books a visit
6	W	Customer inserts the items or items' categories in to-buy list
7	W	Customer inserts the approximate duration of the visit
8	M	Customer receives an alert about the time needed to get to the store
9	M	Customer receives a list of alternative time slots
10	M	Customer receives a notification of a free slot

11	W	Customer shows the QR code representing their number to the checkpoint controller
12	W	Checkpoint controllers controls a QR code shown by a Customer
13	W	Customer takes a physical ticket for a store

1.3 Definitions, acronyms and abbreviations

Definitions:

+ **Booking**: a booking refers to a reservation asked by the client to visit a store, sometimes it is called visit;
+ **Category:** a set of products with similar characteristics;
+ **Checkpoint:** a point at which ticket and temperature checks are performed;
+ **Checkpoint controller:** a worker who performs checkpoint checks;
+ **Customer**: is a person which wants to visit the store in order to buy items;
+ **Detection time**: the service provider is informed of the fault;
+ **Item**: a product which is sell in a store;
+ **Queue:** it is the queue that customers need to be in before entering a store (FIFO);
+ **Queue display:** a screen displaying the number of the ticket whose holder is to be admitted into the store;
+ **Response time**: time required by the service provider to respond to the user;
+ **Repair time**: time required to restore the service or the components that caused the fault;
+ **Recovery time**: time required to restore the system;
+ **Sector:** a store's well defined area where products of certain category are stored;
+ **Store manager**: is a worker of a specific store at which is granted the access of store's parameters modification service;
+ **Ticket:** it can be digital or paper based. It contains a unique QR code used for verification and a number indicating the position in the queue;
+ **Ticket machine**: a machine that provides tickets;
+ **Visit**: a visit is the action of visiting a store entering it, we refer to it when we speak about bookings.

Abbreviations:

+ **MTTR**: mean time to repair. It is the sum of detection time, response time, repair time and recovery time;
+ **OS:** operating system;
+ **S2B**: system to be.

1.4 Revision history

Version	Subversion	Notes
0	1	Initial draft about world phenomena, shared phenomena, machine phenomena, user characteristics and domain assumptions
	2	Initial draft of the overall description: product perspective with connected scenarios, class diagram and statecharts. Product function with high-level requirements. Some definitions.
	3	General assestment on parts. Each member of the group revised the parts written by the others in parts 1 and 2. Part 3 is started by creating a draft of 3A and 3B.
	4	Refinements on goals, requirements and domain assumptions. The mapping between requirements and goal is almost completed. Some requirements are under modification. Software system attributes section has been developed.
	5	Design constraints completed introduced in the RASD documentation. Mapping between goal and requierments updated with the update of the goals definition. Sequence diagrams, use cases and performance requirements sections completed. Initial alloy signatures.
	6	Alloy initial code, fixed some requirements and use cases introducing the possibility to get out by a queue and to delete a booking (all relative mappings are updated). Domain assumptions rewritten.
1	0	Alloy finished and all the document revised and controlled. Sequence diagram fix and added a missing use case to the general use case diagram.
	1	Added some missing descriptions on sequence diagrams and fixed a missing call on a sequence diagram.

1.5 Document structure

Each chapter contains an introduction to the chapter explaining what in the chapter is going to be said. It is a recall of this part of the document to keep track easily what is going to be analysed. In this part the definition is more specific and contains all the needed details in order to read the documentation.

Chapter 1: this chapter introduces the problem and the system required for the realization in remarkably high terms. The chapter includes a high-level description of the stakeholder's goals and the description of the environment where the S2B is going to be realized. The chapter includes also information needed to read and understand the whole document.

Chapter 2: this chapter describes extensively the system and is completely devoted to the system's description in high-level terms. The system is described under different views: scenarios of utilization of the system, definition and characterization of actors, functions to be realized in the system and the assumptions and constraints to the system.

Chapter 3: this chapter treats all the requirements of the system and effectuates a mapping between the goals and the requirements providing a view of the satisfaction of the goals by the functions required for the system. The chapter provides a detailed description of the system attributes and describes the constraints for the realization of the system. Moreover it contains high-level interfaces' view.

Chapter 4: this chapter includes an Alloy code used to describe some system aspects formally. The Alloy language is used to describe the system and verify if some important properties of the system are satisfied by the requirements.

Chapter 5: this chapter describes using synthetic tables the effort of the team members in the process of building the document.

Chapter 6: this chapter includes information on references.

2 Overall description

The overall description is the part of the RASD document where is explained and specified at a high level what are the requirements and the functions to be realized by the system. This section contains some UML models which explains the machine and world for the project. The product perspective paragraph explains the scenarios and the behaviour at a high level of the entities involved in the system's scope. The product function contains an explanation of the requirements which must be fulfilled by the system. The user characteristics section details how each user of the system acts.

2.1 Product perspective

The following scenarios are all set during a viral outbreak.
Scenarios:

Ticket machine and in store queue

An elderly woman, who does not have internet access from home, is running out of food and needs to go shopping for groceries. Once she arrives at her favourite store, she retrieves a ticket from the ticket machine by pushing a button. She observes the queue display waiting for her number to appear while making sure she stays at safe distance from other people. Once her number appears she proceeds to the checkpoint to be controlled. She passes the controls, finishes shopping, and heads home.

Online Queue

A young woman wants to shop at a local grocery and does not have any time preference. She logs into her account, chooses to enter a queue, selects the store, the travel mode and then confirms her queueing. When it's time for her to head to the store, she receives a notification to remind her. She then heads to the store by car, as she specified with the travel mode. Once arrived she waits for her number and goes through the entrance control once her number appears. She is allowed in and proceeds shopping.

Booking

A family of three composed of a father, a mother, and a 4 years old child, needs to buy supplies for the whole family. The father logs in the system's application from his phone and starts booking a ticket for their favourite store. Since they will go altogether, he specifies that the ticket is for a group of three people. He then selects the store, the date, and the time as it is suggested by the application. Since they already know what to buy, he inserts the categories of the items to buy. Finally, the father inserts the chosen way to reach the grocery store. They arrive at the store on time and proceed to the checkpoint. The checkpoint controller performs the usual controls and lets them all in together since their ticket is for three people. They complete their purchases and head home.

Store Manager sets store's parameters

The manager of a small grocery shop wants to change the opening hours of his shop. He logs in his store's account on his pc, and he sets the new hours from the console. He then realizes that since he has recently reorganized the dairy section, more people can safely be in there. He therefore also increases the maximum number of people in that section. He saves the changes and keeps on with his day.

Checkpoint controller at entrance

A worker of a bakery is tasked to be the checkpoint controller of his store. He needs to make sure that the customers respect the queue. To do so, he scans the QR code on every customer's ticket with his phone. A girl approaches and hands him her phone displaying the QR code. He scans it, his app confirms the code validity, and he lets the girl in. Then a woman approaches him with a paper ticket from the ticket machine. He scans it but this time the code is not valid, as that number is yet to be called. He tells the woman to wait for her number to be displayed on the queue display and keeps on working.

Checkpoint controller at exit

A cashier is tasked to scan the ticket of the paying customers. Every time a customer pays she asks them for the ticket and scans the QR code with her phone. She diligently handle her task for all her work hours.

Customer's registration

An elderly man has been shopping using the ticket machine to get his tickets. He realizes that it would be both more convenient and safer for him to use the web app from home. He would avoid unnecessary waiting time in the supermarket. This leads him to buy a new smartphone with a data plan included. He rapidly makes it to the application. Once there, he is guided to register by a very simple and clear UI. During the registration, he inserts personal information. Finished the registration he starts using the app by booking a visit.

Booking deletion

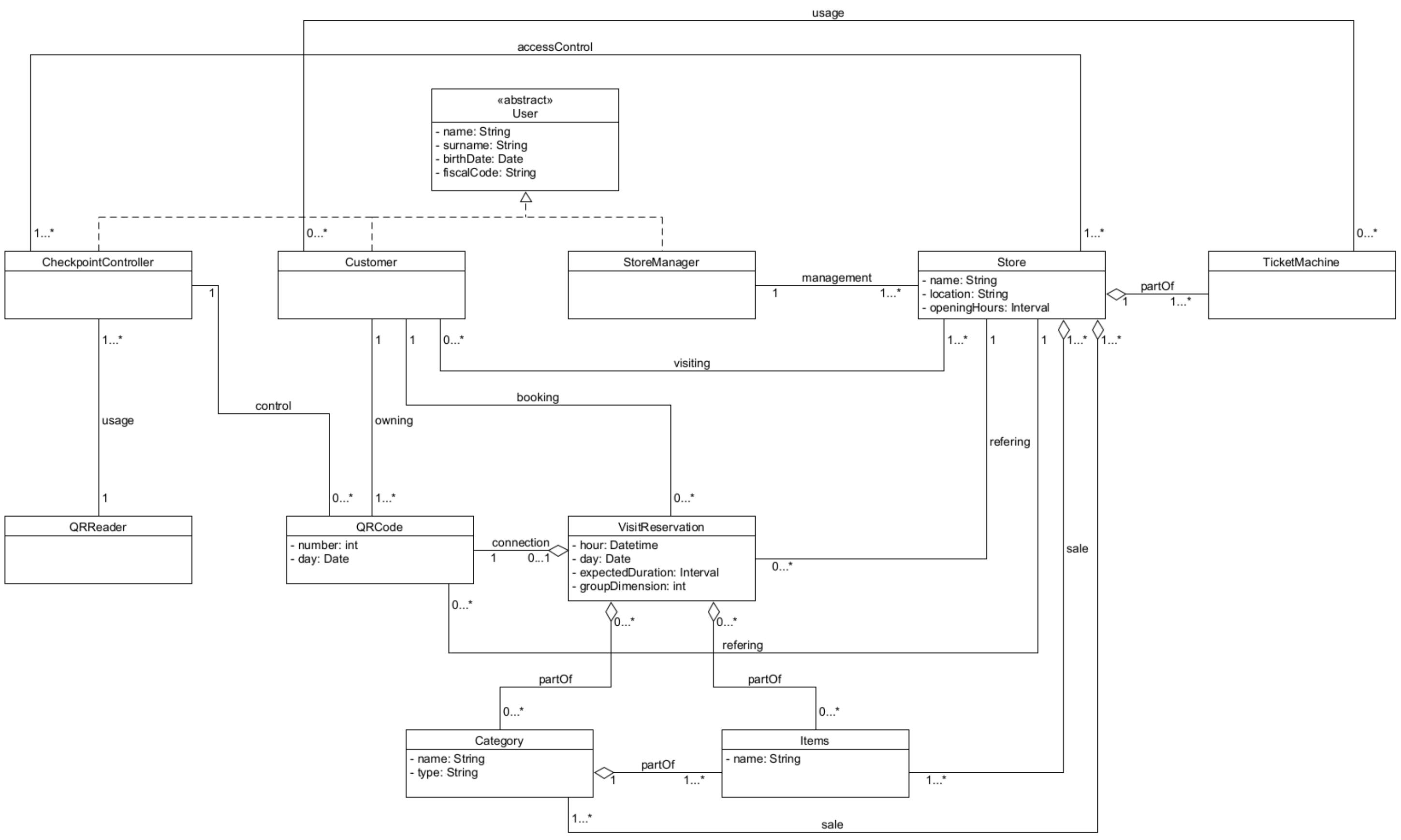
A nurse has been infected by a virus at work. She had previously booked a visit to a store, and since she is sick, she decides not to go. She logs in into her account and deletes her booking. In this way someone else can book instead of her.

Queue change

A man who online queued to shop at a bakery decides that he would rather shop at a supermarket. He therefore logs into his account and enters a new queue. A pop-up message appears on the screen warning him that by doing this he would automatically exit the previous queue. He agrees and queues for the supermarket.

Queue exit

A worker realizes that he cannot attend his queue spot since he now has an important videocall to attend. He therefore logs into his account and exits from his queue, shortening the queue for those behind him.



The application domain model is composed of different actors, the main actors are the customers, the checkpoint controllers and the store managers. The first can visit stores multiple times both visiting it by booking a visit and by either queuing online or physically. After the turn of the customer has been called it can enter the supermarket, however in order to be allowed to access it, it must show its QR code to the checkpoint controller. The checkpoint controller uses a QR Reader on a mobile device which gives it the information about the possibility of the customer to enter the supermarket. Every QR code is related to a visit of a customer to a supermarket and its code is unique and composed of a number and a QR Code. Here we show a top-level class diagram of the application involving all actors which are recorded by the system.
The system is composed of different actors as we can see in the class diagram. The behaviour of the system and its actors is expressed using UML State Diagrams as formalism to represent machines' states.

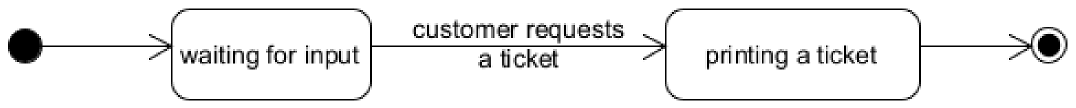


Figure 2.1

When customers are unable use the online application, they can use the ticket machine instead, but for queueing only. The diagram (figure 2.1) describes the evolution of the ticket machine actor during its execution.

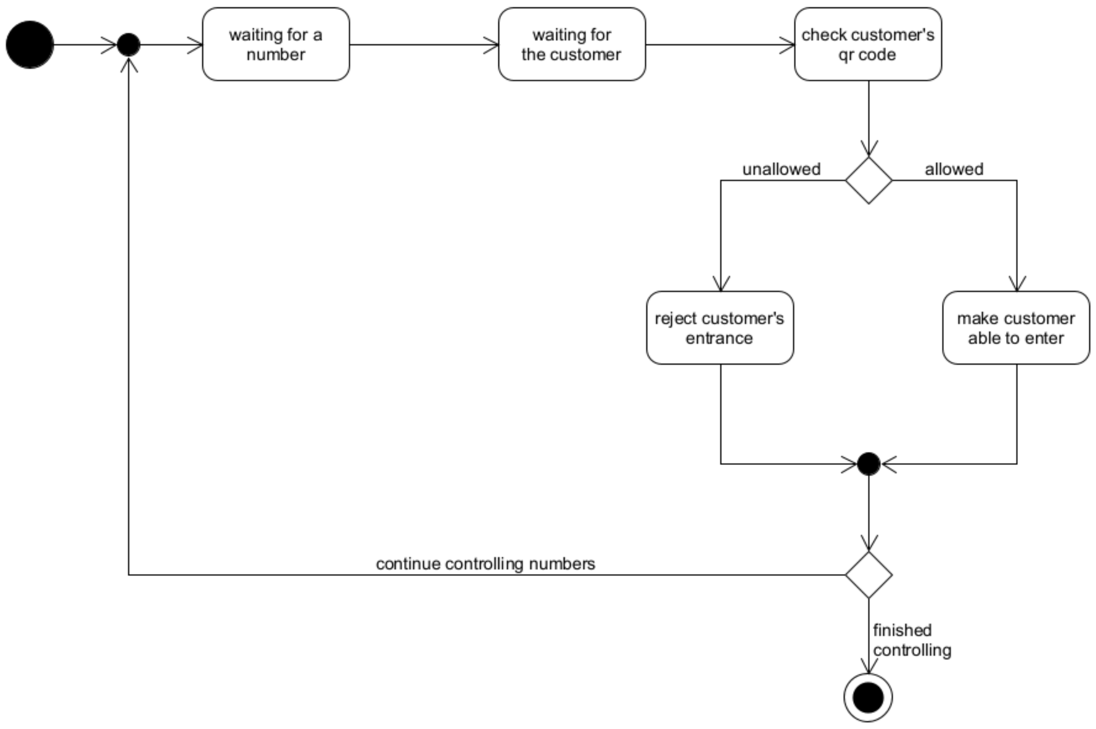


Figure 2.2

Checkpoint controllers have the role of controlling every ticket of the customers which reach the supermarket and want to enter. Their state machine diagram (figure 2.2) expresses how they evolve while doing this specific activity. They control ticket until they must do that.

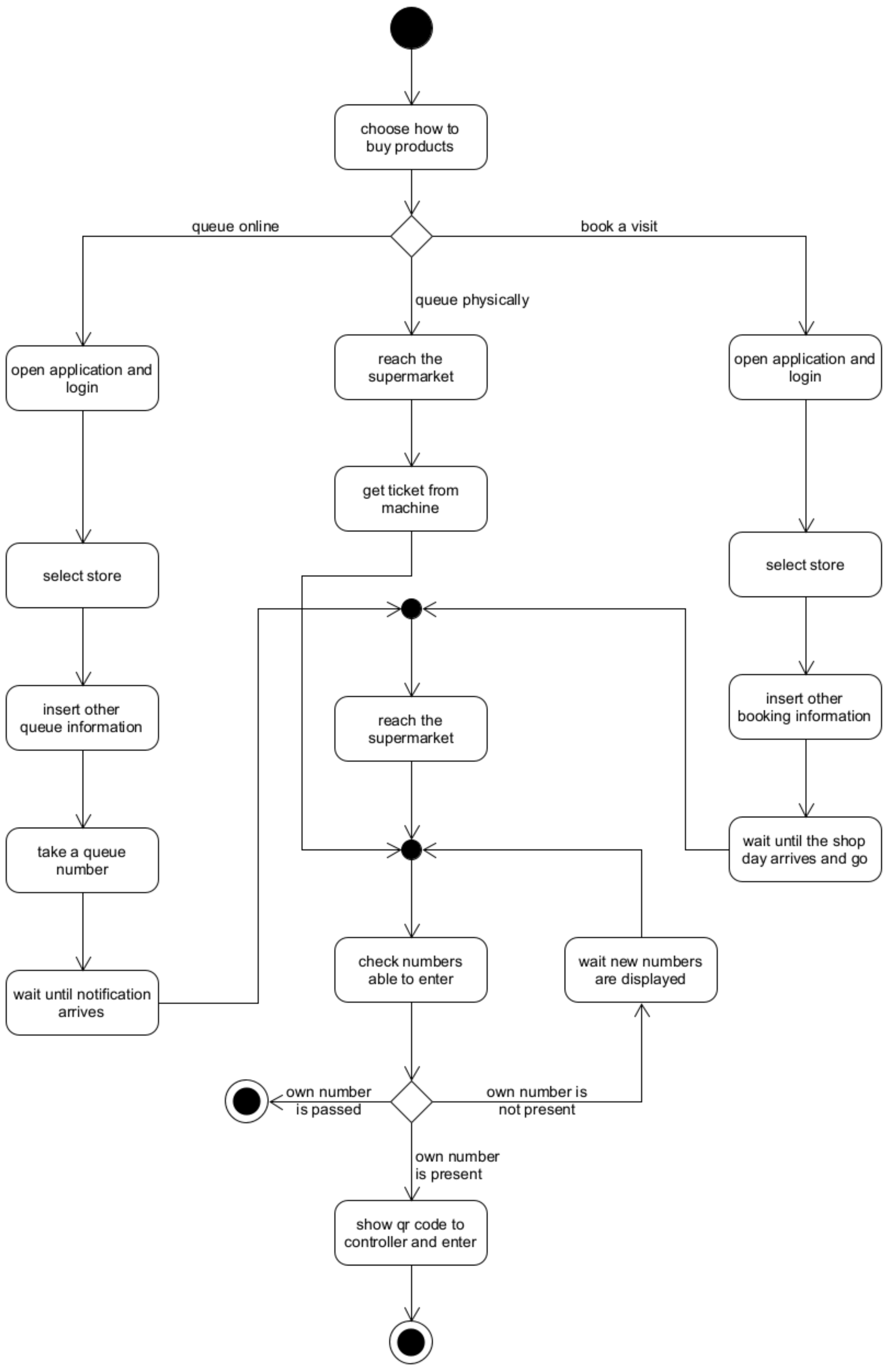

choose how to buy products
queue online
book a visit
queue physically
open application and login
reach the supermarket
open application and login
get ticket from machine
select store
select store
insert other queue information
reach the supermarket
insert other booking information
take a queue number
wait until the shop day arrives and go
wait until notification arrives
check numbers able to enter
wait new numbers are displayed
own number is passed
own number is not present
own number is present
show qr code to controller and enter

Figure 2.3

Customers can arrive to the supermarket because of three reasons: taking a ticket, arriving because of a booking or arriving because its digital turn is going to be called or has called. In this diagram is described customer's state diagram with each possible state for a user interested by the point of view of the system. A user start its flow deciding when and where to buy products and end until it buys the products in the supermarket.

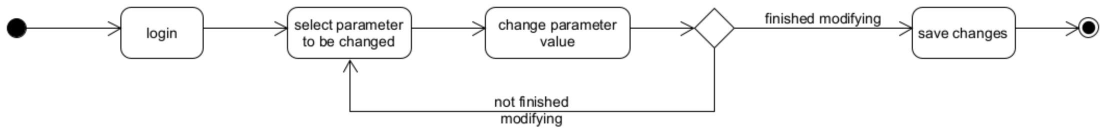


Figure 2.4

Store managers can modify parameters regarding their store from the web app. The diagram (figure 2.4) shows how they can do it.

2.2 Product functions

The system will provide useful functionalities to both customers and store managers/checkpoint controllers.

The main customer-oriented functions are **remote queuing** and **booking a visit.**

The first functionality consists in *letting the customer join a digital queue* for a given store by providing the customer (upon request via their device) a valid digital ticket. Such ticket is the union of a waiting number plus a QR code aimed at proving the validity of the ticket itself. In order to obtain their ticket, customers will have to select a store *from a digital map that displays all available stores* within their surroundings. Before the number associated with their ticket is called by the store, customers will also be able to see *an estimate of their waiting time*, dynamically calculated by the system and displayed on their device. The system will also *notify customers when they should leave* from home to get to the store: such notification will be planned by the system based on an estimation of the time that a customer might take to arrive at the store. In addition to that, the backend will be in charge of avoiding an excessive number of people in the store. In order to do so *the system will not call a new number unless there is a free spot* in the store.

The latter function permits a customer to plan and *book a visit in advance*, opposed to the remote queuing function whose role is the management of real-time queues. Visits can be for a single person or for multiple-people groups, with an upper limit to the size of a group. Customers who select this option will have to specify a store for their purchases among the ones displayed in the same map as before. Then the system will *display a time-table with all the available and occupied time slots* in the store, for the user to choose the most suitable one. The system will also send the user a *list of alternative solutions*, both possible pre-defined time slots or alternative stores for their purchases. Regardless of the option that they choose, the system will send customers a QR code that shall be shown upon entering the store. Customers also have the optional functionality of *providing a list of items that they intend to buy*, for more accurate estimations. If the customer is not able to provide a list of items he or she can optionally *provide a list of categories of items they intend to buy.*

In any case, the system will not accept a QR code that has already been used for a valid entrance: once the customer has entered he or she will need a new ticket or booking in order to re-enter.

Once a customer has purchased all the desired items and paid, he or she will have to show their QR code to a checkpoint controller once again. This step does not check for the validity of the token, it just registers that a customer has left the store and thus a new available spot can be occupied by a customer in the store.

On the other hand, the system will also provide some key **configuration functionalities** to store managers. They will in fact be able to *set some core parameters* in order to regulate fluxes and clusters of people within their store: e.g. they will decide how many people are permitted to stay within one group, how many customers are allowed to stay in a given store section, the duration of time slots and so on. Furthermore, one or more checkpoint controllers will have access to a **QR scanning functionality**, in order to scan the QR codes of employees and customers to monitor their access. Scanning a QR code (on entrance) will *establish whether it is valid or not* and will automatically make it non-reusable.

Finally, the system will provide general purpose functionalities to all of its users, such as **registering, login, account management…** Distinction will be made between regular customer accounts, checkpoint controller accounts and store manager accounts. Each user will obviously have *access to the functionalities of their role only* e.g. customers will only have access to customer-oriented functionalities.

2.3 User characteristics

This section presents the users and their characteristics:

- **Store Managers**
    - They are in charge of their store's organization. Particularly, they decide the opening hours and the logistics regarding the customers.
- **Customers**
    - They can be of any demographics. Shopping for groceries is a need that everybody has;
    - Their ability to interact with technology varies greatly. While younger generations are accustomed to technology the elderly might struggle to use complex applications;
    - They may have internet at home as well as they may not. Even though in developed countries the vast majority of customers are expected to have internet access at home, a sizable portion still doesn't. According to a 2020 statistic, in Italy 76% of households have it;[1]
    - Normally customers go grocery shopping with a least a vague idea of the items to buy;
    - The time spent in the store by customers varies greatly. It can usually be loosely predicted by the customers;
- **Checkpoint controllers**
    - They have a mobile devices used to scan QR codes;
    - They are workers who control the flow of customers at the entrance of the store;
    - They may grant access to customers as well as they may disallow customers to enter the grocery store.

2.4 Assumptions, dependencies and constraints

The following statements are assumed to be true:

1. Customers are not allowed to enter unless they have received a ticket or booked a visit.
2. Customers can get their ticket by either using their app or physical machines at the store, and in no other way. Visits can be booked by using the app, and in no other way.
3. The maximum number of people allowed to be in a store sector at the same time N is known.
4. Customers who book a visit and declare in advance their intended actions will respect their declarations, i.e. they will not visit sectors other than the declared ones. Customers who do not explicitly declare their intention might instead visit any store sector.
5. Customers who book separate visits/ get different tickets will not get in direct contact inside the store, thanks to some employees in charge of controlling their behaviour.
6. Checkpoint controllers shall make sure that customers can enter only after proving that their ticket is valid and, in case they booked a visit, that the real size of their group does not exceed the declared size.
7. Every access point to the store or exit is monitored by one or more checkpoint controller who is in charge of preventing irregularities. There is no hidden way in which a customer can enter/exit the store.
8. Customers enter one group at a time if they have booked a visit. Queuing customers cannot form groups, they can only be single clients.
9. Customers respect all safety norms while waiting outside, and all checkpoint controllers will make sure to monitor that. E.g. in case a customer is not respecting social distance or is not wearing a mask, a checkpoint controller will take care of the matter.
10. A customer can reach the store by foot, car or public transport. No other ways of getting to the store are considered.
11. At the exit of the store, some employee will scan the QR code once again to register that a customer has left the store. Customers are not permitted to exit unless such event is registered.
12. We assume that customers who retrieve a paper ticket at the machine will not take multiple tickets at a time (this could be enforced by adding an employee who is in charge of checking customers who use the machine).

3 Specific requirements

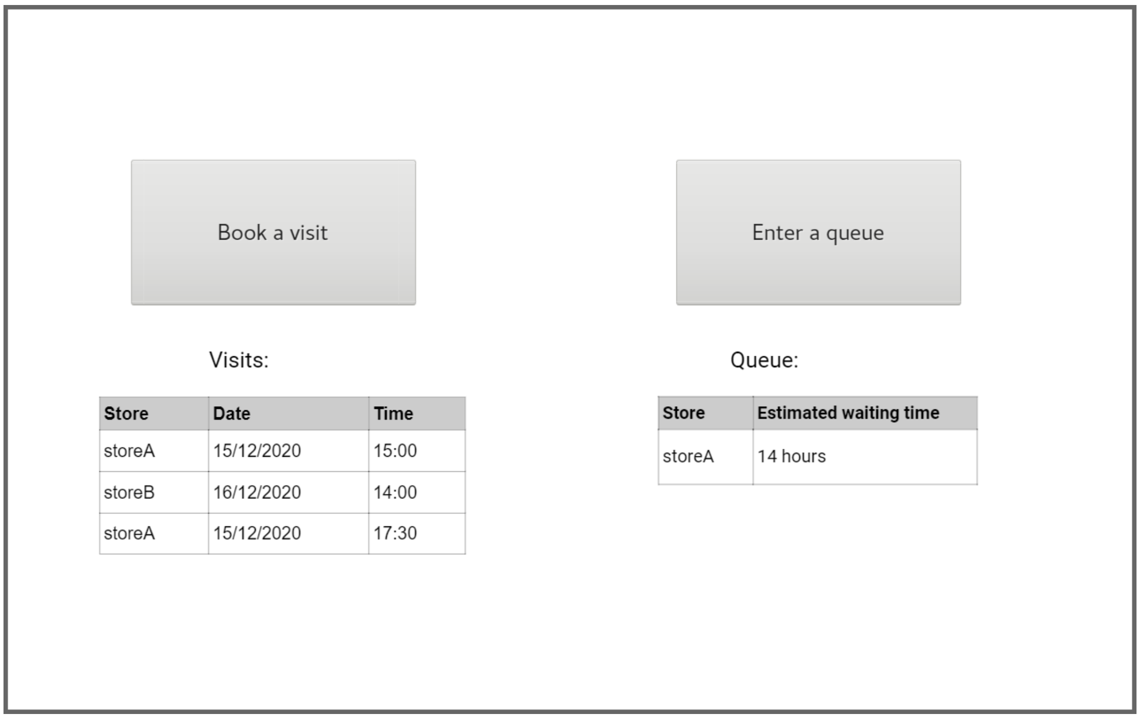
In chapter 3 we show the requirements for the S2B and we map the goals to the requirements and the domain assumption for the S2B. Then after the mapping we list some use cases and sequence diagrams representing the interaction of actors with the system and the workflow of some functionalities which are granted by the systems and can be used by customer, store manager or checkpoint controllers. At the end of the chapter we introduce some specific constraints for that system and we express the software attributes.

3.1 External interface requirements

3.1.1 User interfaces

The user interfaces presented are extremely minimal. They are then greatly expanded in the design document when the platforms have been decided.

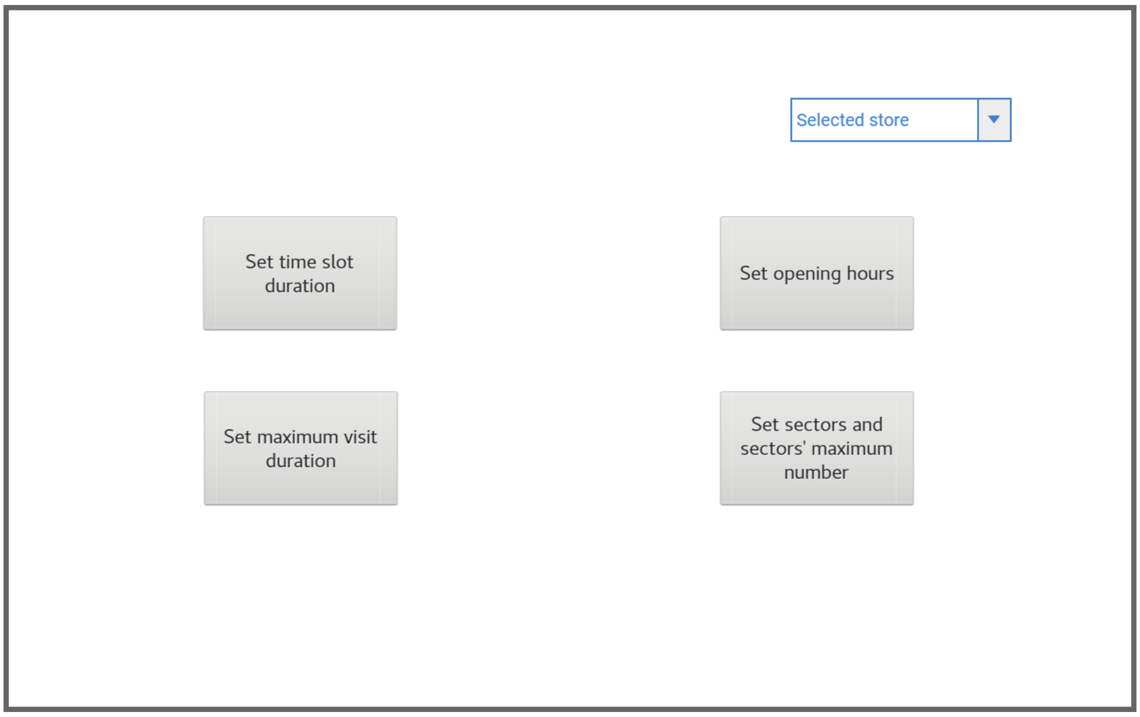
- **Customers interface:** offers the queueing and booking functionalities. The customer can also see the current booking or queue if he has one. On selecting a queue or a booking the customer will access the digital ticket and will be able to delete the booking or exit the queue.



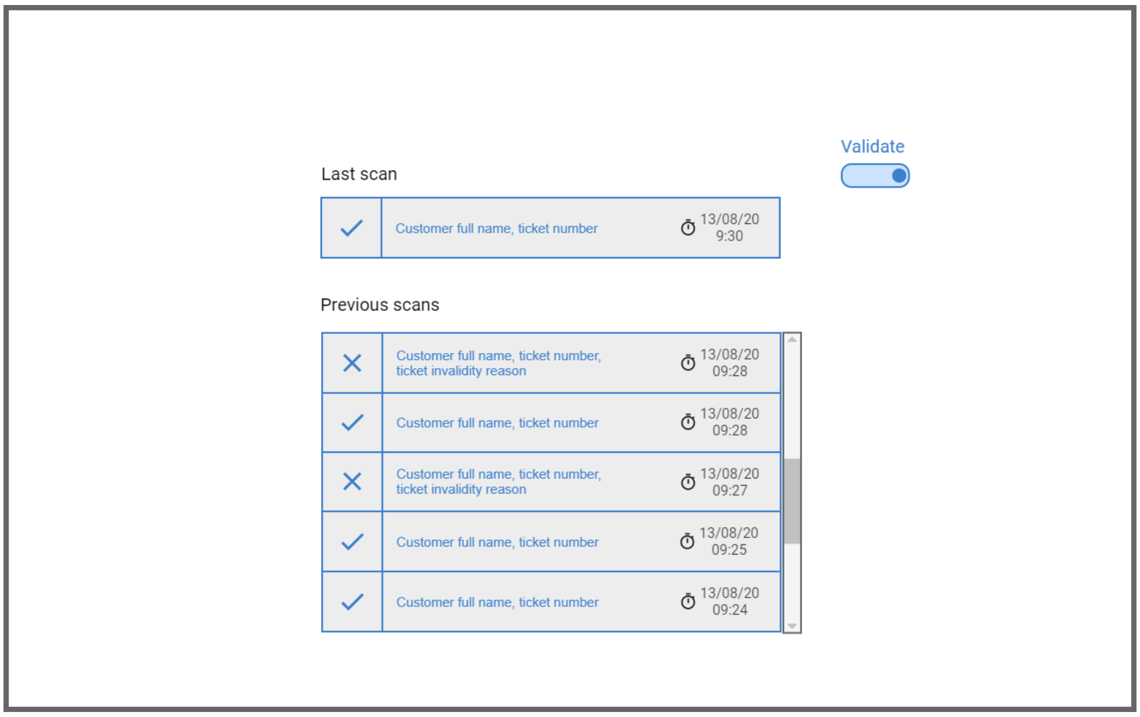
Store	Date	Time
storeA	15/12/2020	15:00
storeB	16/12/2020	14:00
storeA	15/12/2020	17:30

Store	Estimated waiting time
storeA	14 hours

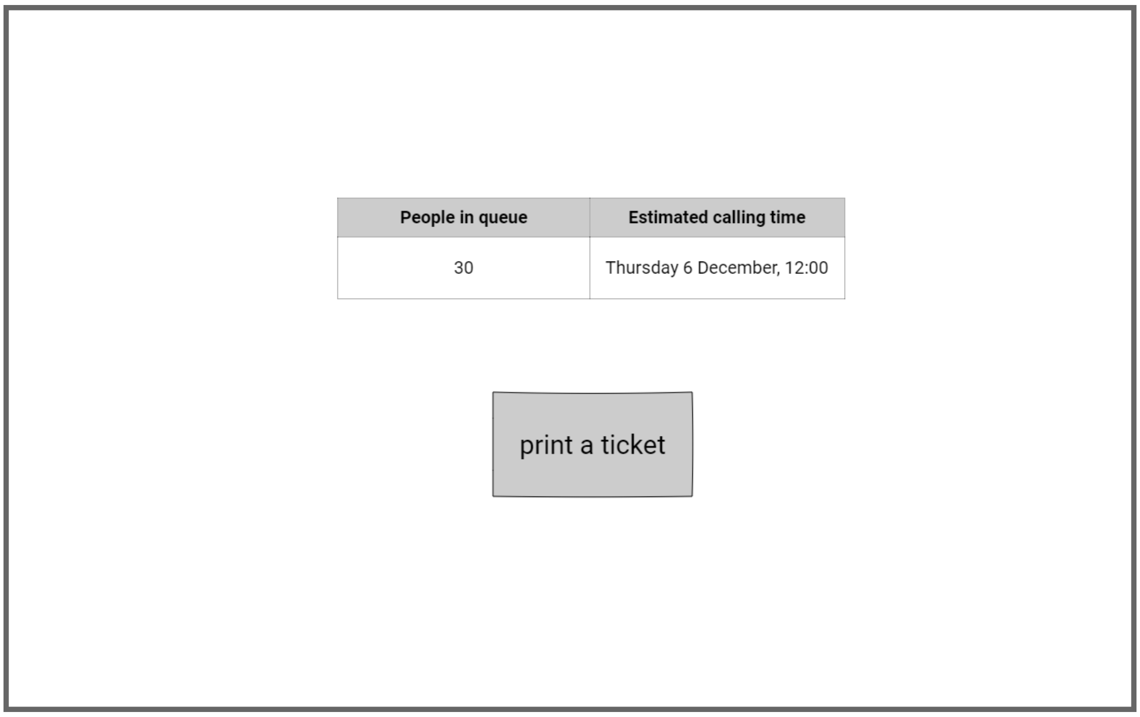
- **Store managers interface:** offers a console where the store manager can select a store of his and set the store's parameters. Particularly he can set the time slot duration, the maximum visit duration, the opening hours, the sectors and their maximum number of people.


Selected store
Set time slot duration
Set opening hours
Set maximum visit duration
Set sectors and sectors' maximum number

- **Checkpoint controllers interface:** scans the ticket's QR code, showing on screen its validity. The previous scans remain on screen on the list. The toggle can be used to switch between the validate mode, used to admit costumers into the store, and the check-out mode, used to register the customers exit.

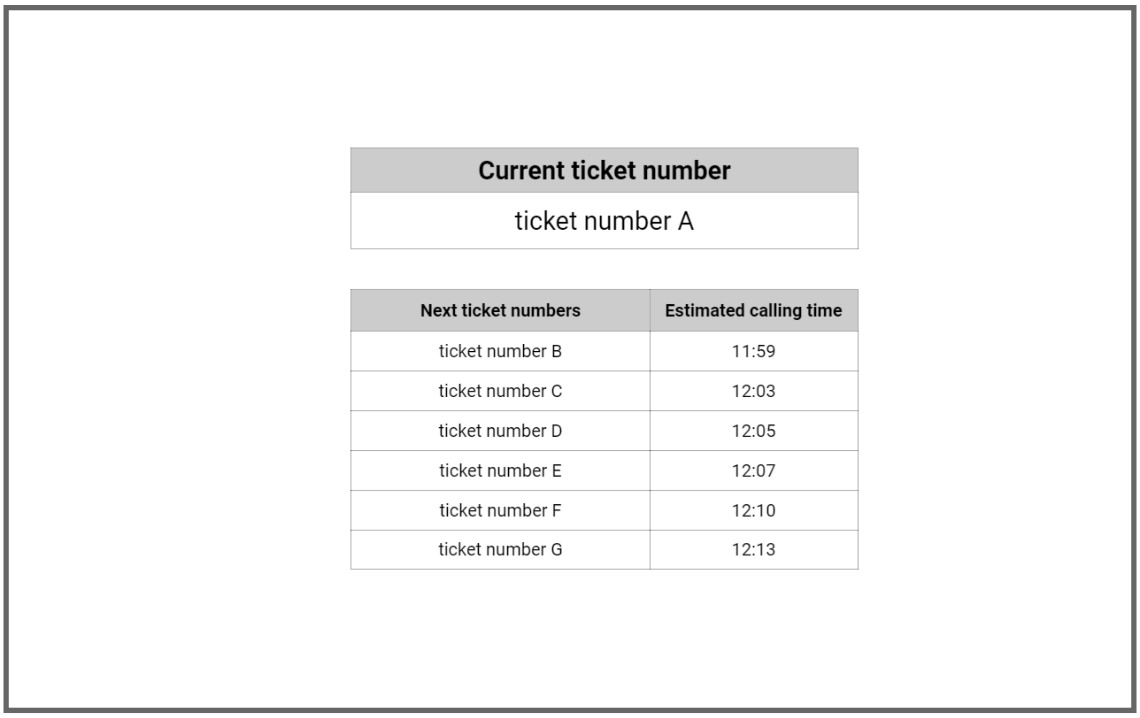

Validate
Last scan
Customer full name, ticket number 13/08/20 9:30
Previous scans
Customer full name, ticket number, ticket invalidity reason 13/08/20 09:28
Customer full name, ticket number 13/08/20 09:28
Customer full name, ticket number, ticket invalidity reason 13/08/20 09:27
Customer full name, ticket number 13/08/20 09:25
Customer full name, ticket number 13/08/20 09:24

- **Ticket interface:** very easy and basic UI with a button to print a ticket. It shows how many people are already in the queue and how much time will approximately take for the printed ticket to be called.



People in queue	Estimated calling time
30	Thursday 6 December, 12:00

- **Queue display interface:** shows the queue state of a store. Particularly it shows the current ticket number, whose owner is supposed to be welcomed into the store, the next tickets numbers that will be called, and their approximate calling time.



Current ticket number
ticket number A

Next ticket numbers	Estimated calling time
ticket number B	11:59
ticket number C	12:03
ticket number D	12:05
ticket number E	12:07
ticket number F	12:10
ticket number G	12:13

3.1.2 Hardware interfaces

- **Speaker**: it is used by the checkpoint controller at the entrance to call out loud the ticket's numbers.
- **Queue display**: it is used to show the ticket's numbers called and about to be called.

3.1.3 Software interfaces

- **Map service:** it is needed to estimate how much time the user will need to reach the store
- **Storage service:** it is used by the application to store and retrieve data to perform its main activities
- **QR code reader software:** it is used read the QR code
- **Ticket machine OS:** it is necessary to run the ticket machine app.
- **User app OS:** it is necessary to run the user's app.

3.1.4 Communications interfaces

- **Internet:** the application uses internet for remote communication with users

3.2 Functional requirements

Requirement	Definition
R1	A user who wants to use the system online as a customer must register for free. Further uses require customers to login with valid credentials
R2	Store managers and checkpoint controllers need to log in as well, but their accounts are created by a sys-admin. Thus they do not need to register as store managers or controllers
R3	After logging in, a user can only access the functionalities that are specific of their role
R4	The system will display a digital map with all the available stores in their area. Customers can select a store where they intend to make a purchase, among the displayed ones
R5	Customers can join a digital FIFO queue for a store. In order to join such queue, they also need to specify how they intend to reach the store. After joining a queue, the system will send the user a digital ticket. A user can have at most one valid ticket for a given store at any time, i.e. it is not allowed to get another ticket for a store S while being in the digital queue for S_2: their requests for another ticket is simply overwritten (if the selected store is the same one for which the customer is already queuing, the request is simply denied).
R6	A digital ticket consists of a number representing the position in the queue and a QR code. For any two customers waiting in the same queue, their waiting numbers are not equal
R7	If a customer A is in a digital queue, the system will display an interval (ti, tf) representing the estimated waiting time. Such interval is calculated as a 95% confidence interval for the real waiting time T.
R8	When the estimated time calculated at point 7 is less or equal than the estimated travel time for a customer A, they will receive a notification telling them to reach the store
R9	Customers can retrieve a paper printed ticket at the store. Such ticket is identical to a digital ticket for what concerns its validity constraints
R10	The system will "call" waiting numbers by displaying them on a monitor at the entrance of the store
R11	Assuming that at most N people are allowed to be in the store at the same time, and M people are currently in the store, the system will call N-M numbers sequentially. If N-M > 1, then the calls will have a temporal distance of two minutes
R12	A non-scanned ticket for a queue is valid for store S if and only if it was issued for S and its number has not been called by S yet or if the call happened no longer than 2 minutes before. In any other case, the ticket is marked as invalid, and the next waiting number will be called

R13	A customer may also ignore the digital queue and book a visit for a store instead
R14	If the functionality specified at point 13 was selected, the system will display a time table on the customer's device, plus a set of pre-calculated suggested visits. Each day is divided in time slots of equal length. The customer may either select a finite number of contiguous and free time slots for their visit, or one of the suggested visits
R15	In case the customer chooses to specify their own time interval, the total time they specify must not exceed a time limit established by the store manager. Furthermore, customers can only choose time slots that start and end after the opening time and before the closing time of the selected store. Finally, the visit must not overlap with any other visit booked by the customer.
R16	While booking the user may input a list of items and categories of items that they intend to buy. Their app will show a list of categories and items that they can choose
R17	Then the system will send the user a QR code that certifies their booking
R18	A non-scanned QR code for a visit is valid for store S if and only if it was issued for S and either the selected time of arrival has not arrived yet or no more than 5 minutes have passed since the selected time of arrival
R19	Checkpoint controllers can scan a customer's QR code upon letting them into the store. The system will check the validity of the code, and will display an error message if the token has lost its validity. After being scanned and approved, the token is marked as invalid, and thus cannot be reused for entrance
R20	A store manager who has previously logged in may set parameters for their store only. Such parameters are: the maximum number of people allowed in a store sector at the same time, the length of each time slot, the maximum permitted duration of a visit, opening and closing time of the store
R21	If a customer has used their app for longer than one month, the system will send them a notification once every two days. Such notification contains a list of 3 stores that are within their area and that have free time slots. The stores whose data is sent are always the 3 stores for which the sum of free time intervals at the time of sending is maximum
R22	A checkpoint controller can also scan the customer's QR code when they are exiting the store. However, this time the system will not check the validity of the token
R23	Each booking requires the specification of the number of people that will make the purchases within the same group

R24	Customers are allowed to delete a booking at most 30 minutes before the expected time of the booking. A customer can also exit their queue at any time if they wish.

3.2.1 Mapping between requirements and goals

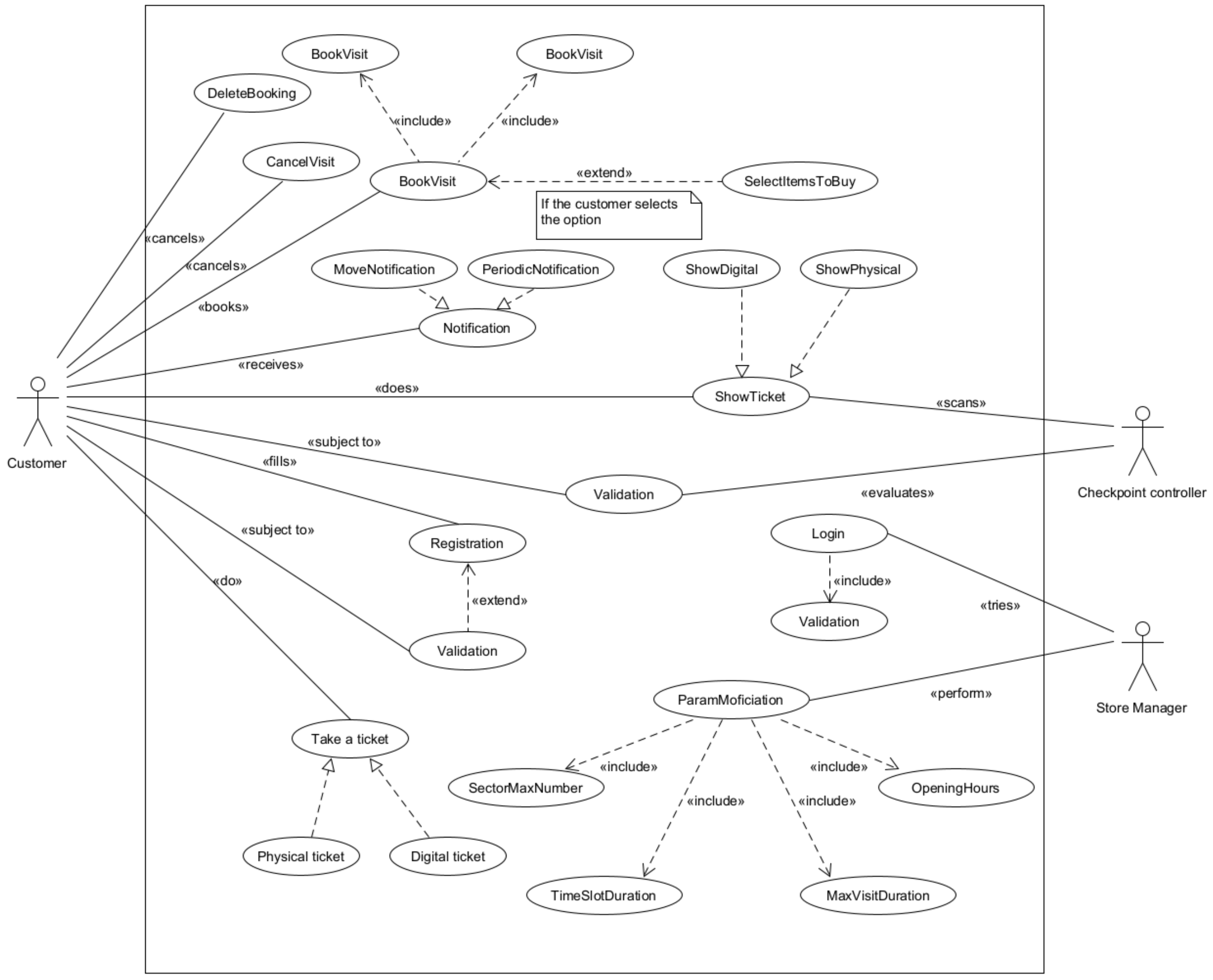
Requirements must be mapped in the goals of the application. Goals must be granted given the domain assumptions and the requirements. Here there is a list of the mappings between the requirements and the goals of the application. We adopt the convention to identify the requirements with the format RN where N is the requirement number and GN is the goal number N:

Req.	Goals
R1	G1, G2, G3, G9
R2	G4, G11
R3	G1, G2, G9, G11
R4	G1, G2, G3
R5	G1, G5, G7, G10
R6	G1, G4, G6, G7
R7	G4, G6, G8
R8	G4, G6, G8, G9
R9	G4, G5
R10	G5, G7
R11	G6, G12
R12	G4, G12
R13	G2, G4, G6
R14	G2, G13
R15	G2, G11
R16	G4, G13
R17	G8, G9
R18	G4, G12
R19	G12
R20	G4, G11
R21	G13
R22	G8
R23	G2
R24	G3, G5

The previous mapping states where requirements goes, this is helpful in identifying the purpose of a requirement in the project. However, to spot the importance of a goal and to identify how the goal can be guaranteed is important to apply the inverse mapping, the mapping from goals to requirements:

Goal	Requirements	Domain assumptions
G1	R1, R3, R4, R5, R6	D2
G2	R1, R3, R4, R13, R14, R15, R23	D2
G3	R1, R4, R24	-
G4	R2, R6, R7, R8, R9, R12, R13, R16, R18, R20	D3, D4, D5
G5	R5, R9, R10, R24	D2
G6	R6, R7, R8, R11, R13	D9
G7	R5, R6, R10	D1, D2
G8	R7, R8, R17, R22	D3, D11
G9	R1, R3, R8, R17	D10
G10	R5	D12
G11	R2, R3, R15, R20	-
G12	R11, R12, R18, R19	D1, D5, D6, D7
G13	R14, R16, R21	-

3.2.2 Use cases



3.2.2.1 Registration

Registration involves a customer which is not already registered to the service. It must use the generic web application accessible to everyone and start registration filling the form.

Use case	Customer Registration
Actor	Nonregistered customer
Entry condition	A non-registered customer has accessed their app
Basic Flow	1. In the homepage, the non-registered customer clicks the "Register" button. 2. The non-registered customer inserts their data. 3. The non-registered customer submits their data and completes registration
Exit condition	Non-registered customer owns an active Customer account.
Exception	The user inputs invalid data. An error message is displayed.

	The user already owns an account with the inputted credentials. An error message is displayed.

3.2.2.2 User login

Actions on the system performed by users may be of different type depending on the different category. Every user must log in.

Use case	User Login
Actor	User
Entry condition	A non-logged user opens the app
Basic Flow	1. In the homepage, User clicks the "Login" button. 2. User enters their username and password 3. User clicks the "Log me in" button.
Exit condition	User has successfully logged in and has access to the functionalities of their role.
Exception	One between the username and password fields is invalid. An error message is displayed.

3.2.2.3 Store manager sets parameters

The modification of parameters done by the store manager passes through its application, which is an important actor in this use case.

Use case	Store Manager Configuration
Actor	Store Manager
Entry condition	A user is logged in as a Store Manager
Basic Flow	1. In the homepage, the Store Manager enters the Configuration section of the app. 2. Store Manager selects the store whose parameters he wishes to change. 3. The system displays an overview of the current parameters of the selected store. 4. The Store Manager selects and modifies the following parameters: maximum number of people allowed in a sector, maximum duration of a visit, length of time slots and opening/closing hours, maximum people in one group for a booking. Subsets of such parameters can be modified as well. 5. The Store Manager confirms their submission.
Exit condition	The value of the parameters selected by the Store Manager are set to the values that were chosen.
Exception	The Store Manager inputs invalid data. An error message is displayed.

3.2.2.4 Notifications

Notification is the process of the system intended to notify the user about a free slot or about the necessity of getting out to reach the grocery store in time.

Use case	Receive Move Notification
Actor	Customer
Entry condition	Customer has a valid digital ticket for a store and is queuing
Basic Flow	1. Customer receives a notification telling them that it is time to go to the store. 2. Customer clicks ok and starts travelling to the store.
Exit condition	Customer knows that it is time to get to the store.
Exception	

Use case	Receive Periodic Notification
Actor	Customer
Entry condition	Customer has logged in their account
Basic Flow	1. Customer receives a notification with a list of some available stores in their area. 2. Customer clicks on the notification.
Exit condition	Customer is redirected to the Book a Visit use case
Exception	Customer deletes the notification.

3.2.2.5 Taking a ticket

Taking a ticket is an activity involving three different actors of the S2B.

Use case	Physical ticket
Actor	Customer
Entry condition	Customer is physically at the store and can use the ticket machine.
Basic Flow	1. Customer clicks “Print ticket” on the machine. 2. The machine prints a valid ticket containing a waiting number and a QR code to be shown.
Exit condition	The Customer can now use their ticket to enter the store when their turn arrives.
Exception	Machine ran out of paper. An error message is displayed and checkpoint controllers are notified.

Use case	Digital ticket
Actor	Logged Customer
Entry condition	Customer has successfully logged in their account
Basic Flow	1. Logged Customer clicks the “Queue for a store” button.

	2. The system displays a map containing all the stores in their area. 3. Logged Customer selects a store from the map and specifies how they intend to reach it. 4. Logged Customer is added to the queue for the selected store and receives a digital ticket with a waiting number and a QR code. 4.a If the customer already has a ticket for another store (not for the same store), it is simply replaced by the new one
Exit condition	Logged Customer can use the digital ticket to enter the store when the waiting number is called. They can also see the state of the queue, and the estimated waiting time.
Exception	Logged Customer already has a digital ticket for the selected store's queue. The request is denied and an error message is displayed..

3.2.2.6 QR Code validation

Taking a ticket is an activity involving only the customer and the checkpoint controller.

Use case	QR Code Validation
Actor	Customer, Checkpoint controller
Entry condition	A Customer has approached the Checkpoint controller to show their ticket. The Checkpoint controller has previously accessed the app and successfully logged in.
Basic Flow	1. Checkpoint controller scans the QR code provided by the Customer. 2. After being scanned, the QR code is examined and validated. 3. Access to the store is granted to Customer.
Exit condition	Customer can enter the store, but their ticket is marked as invalid. Thus it can no longer be used.
Exception	The ticket is invalid (for any reason). An error message with a brief description of the reason for invalidity is displayed on the Checkpoint controller 's device. The Customer is not allowed to enter.

3.2.2.7 Visit booking

Use case	Book a Visit
Actor	Customer
Entry condition	Customer has accessed the app and successfully logged in

Basic Flow	1. Customer selects the Book a Visit functionality. 2. A map containing all the available stores is shown. 3. Customer clicks on one of the stores. 4. The app displays a time table with some free time slots for the selected store. Some suggested time slots (either for that store or other stores) are displayed as well. 5. Customer selects the time slots that they need and clicks “Next”. 6. Optionally, Customer can open a list of items and select some items that they intend to buy from the store. 7. Customer inserts the number of people that will visit the store together. 8. Customer confirms their booking and click “Submit”. 9. The app displays message with a QR code and the day and time at which the visit starts.
Exit condition	The visit is successfully registered and the Customer can use their QR code to enter the store.
Exception	Customer’s selected time slots overlap with a Visit that they have already booked. The Booking operation is denied and an error message is displayed.

3.2.2.8 Booking deletion

Use case	Delete Booking
Actor	Logged Customer
Entry condition	Customer has booked a visit and the current time is at least 30 minutes before than the visit time
Basic Flow	1. Logged Customer selects the booking. 2. Logged Customer clicks the “delete visit” button.
Exit condition	Logged Customer has successfully deleted the booking and cannot enter the store at the time of the previous booking.
Exception	

3.2.2.9 Queue exit

Use case	Exit Queue
Actor	Logged Customer
Entry condition	Customer is in a queue for a store and its ticket is digital
Basic Flow	1. Logged Customer enters the queue info screen section. 2. Logged Customer clicks the “exit queue” button.
Exit condition	Logged Customer is no longer in the queue. Their ticket is deleted and thus entrance is no longer permitted.

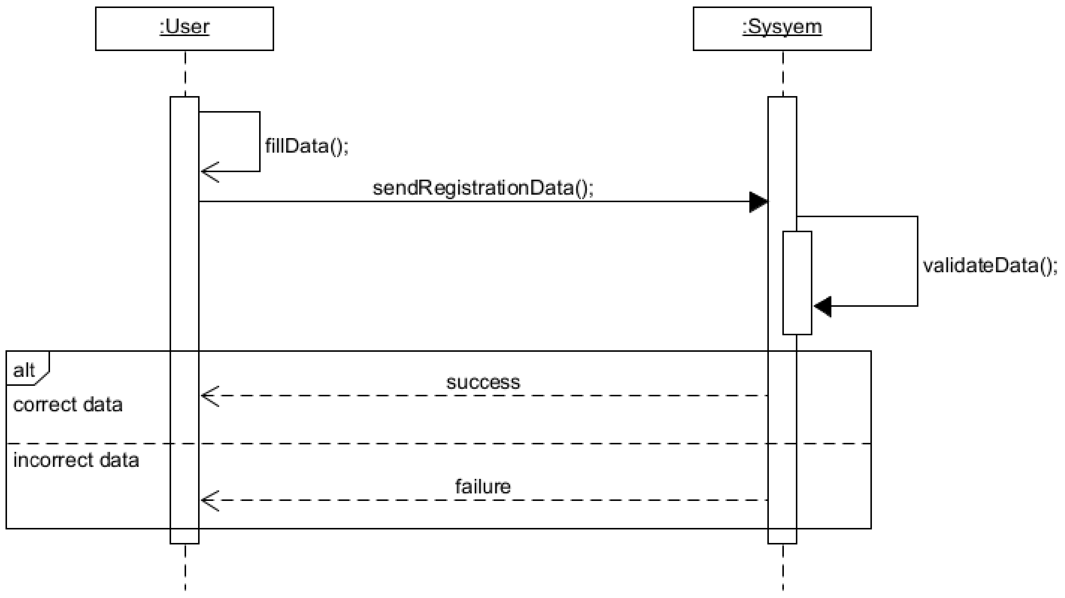
Exception	

3.2.3 Sequence diagrams

Here the behaviour of the actors of the systems is specified over different types of actions at high level terms. Some cases of high-level functions have been identified and are described in terms of UML Sequence diagrams and Activity diagrams.

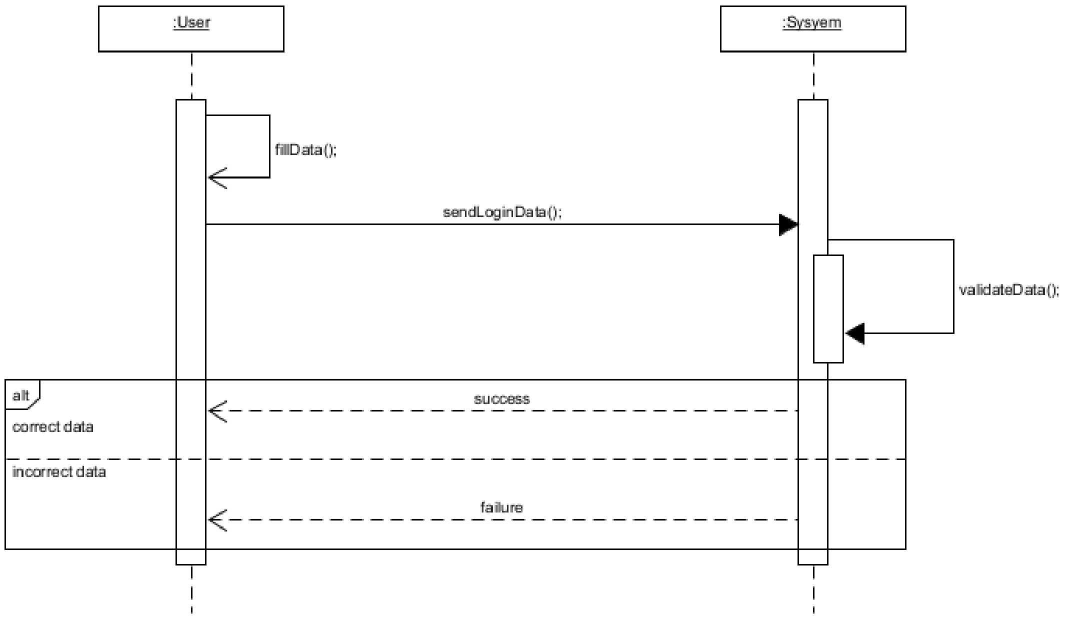
3.2.3.1 Registration

In the registration process, the only external actor present is the Customer who is intended to subscribe to the system to be able to access to online services like: online queuing and visit booking. Here is the sequence diagram regarding the associated registration use case:



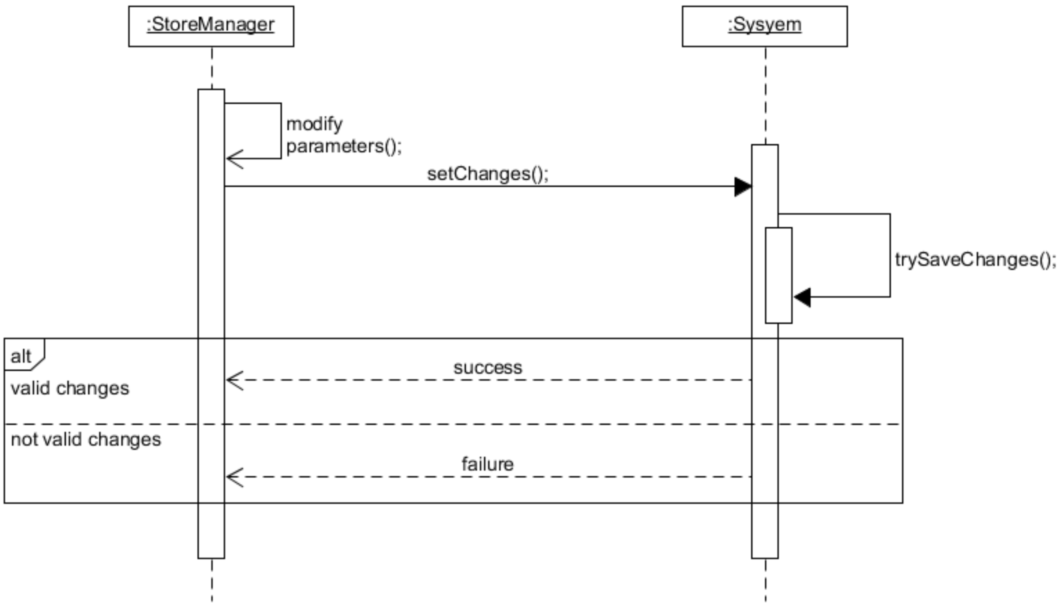
3.2.3.2 Login

The login process is the same for every user of the application. It consists as any other login of inserting the credentials so that the system can validate them.



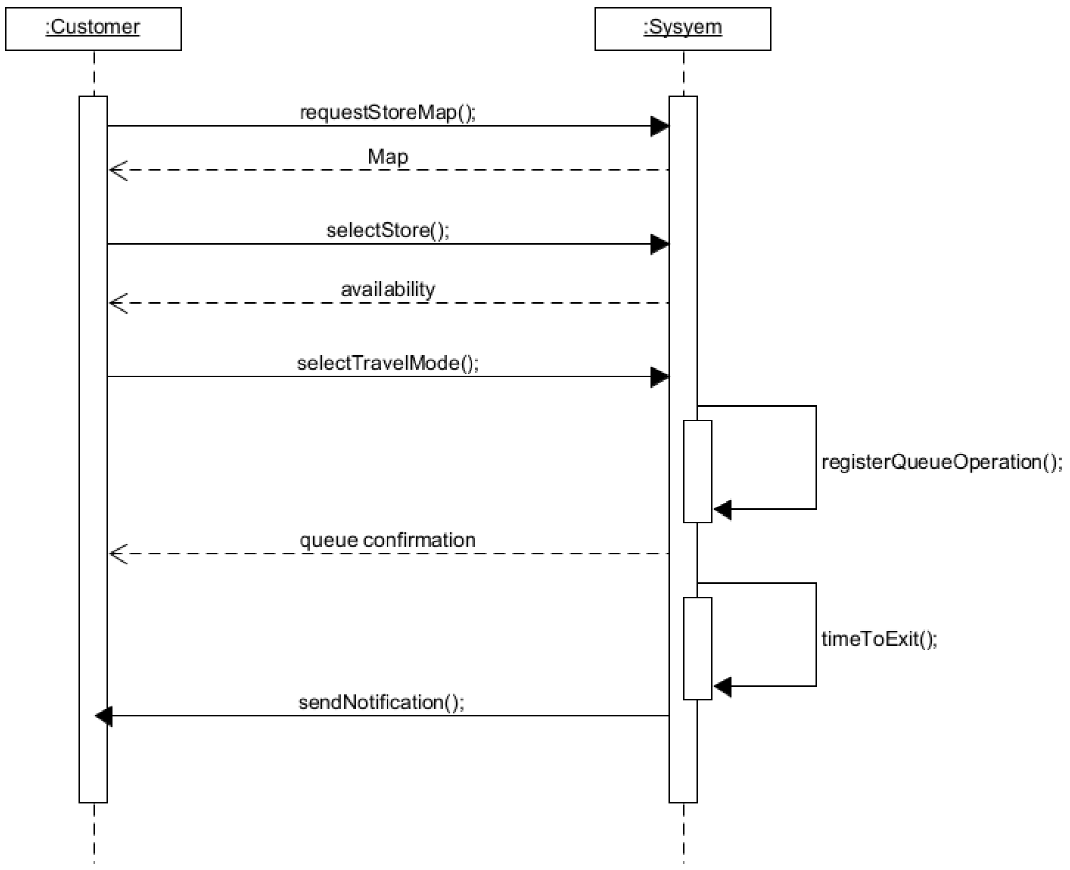
3.2.3.3 Store manager’s parameters’ modification

The store manager can modify the parameters he desires. When he is done, he needs to set his changes so that the system can save them if they are valid.



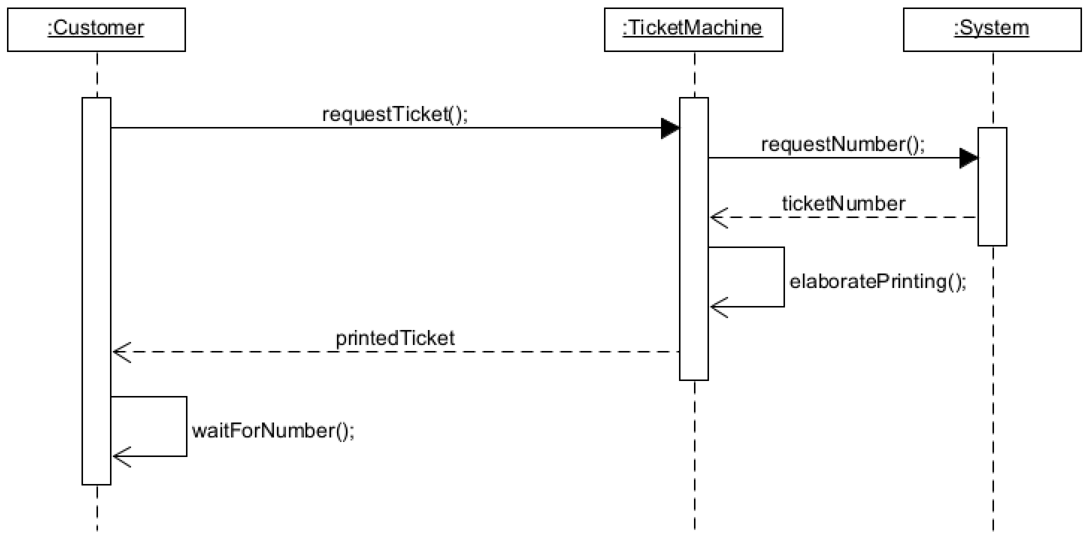
3.2.3.4 Online queue with notification

The online queue allows customers to get a ticket for the store they selected. It also sends a notification when it is time for them to head to the store.



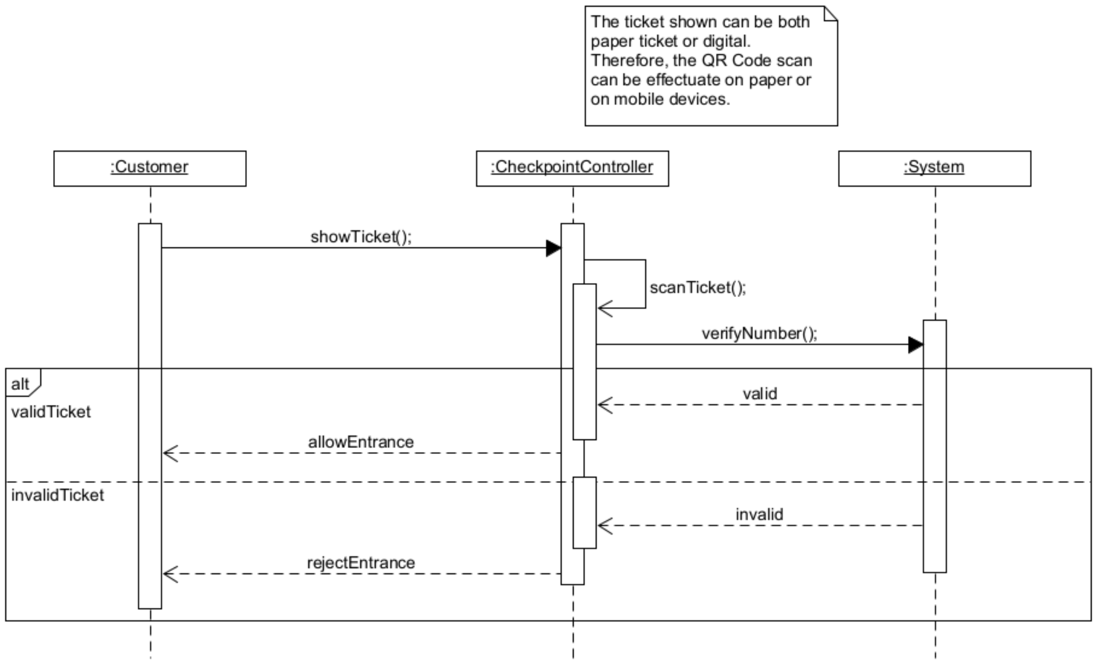
3.2.3.5 Physical queue

The case of a physical queue happens when a person does not have the possibility to queue online due to technology gap or because simply does not want to use the application. In each case the system must provide a ticket and continue to call numbers.



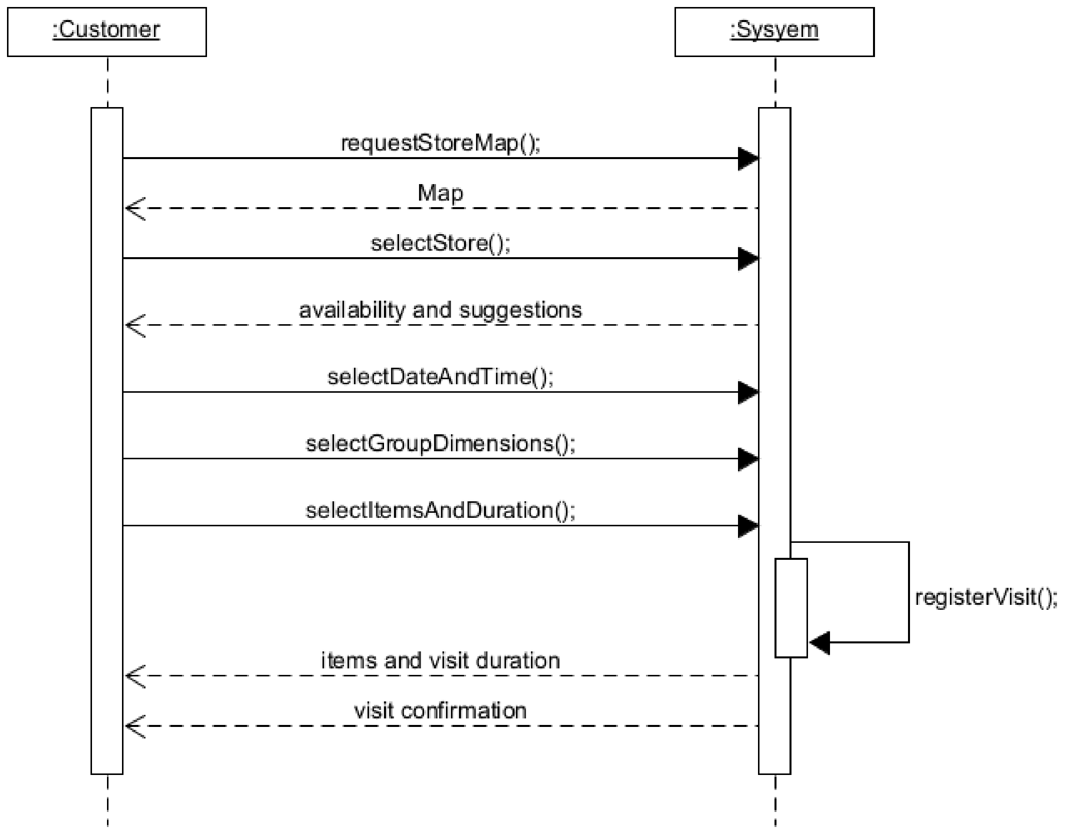
3.2.3.6 Verification of digital or physical ticket

A person approaches the checkpoint controller at the entrance of the store. The external actors involved are the customer and the checkpoint controller. In this sequence diagram are represented both the cases of acceptance and rejection of a ticket from a certain customer.



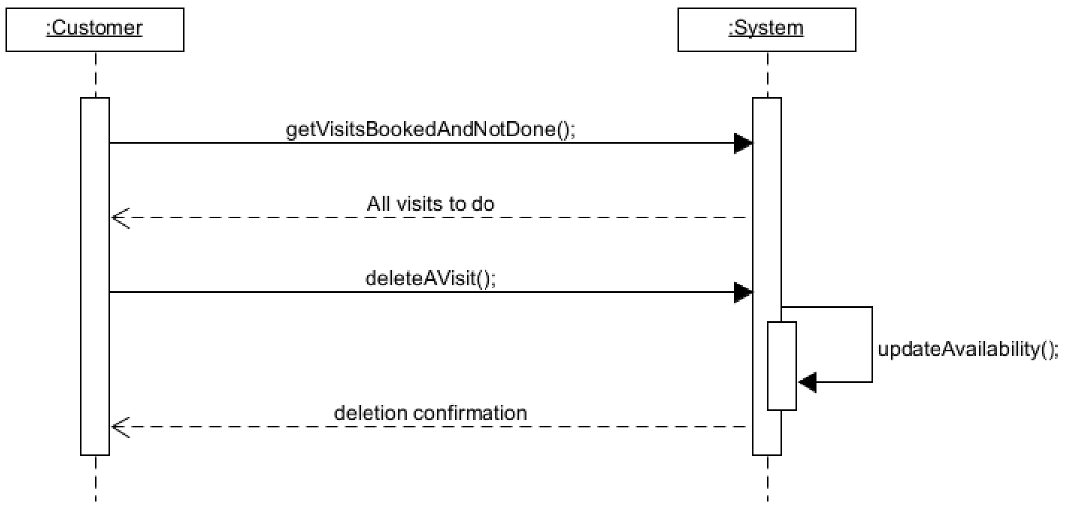
3.2.3.7 Visit booking

Visit booking involves only the customer as external actor. All the interactions happen between the customer who wants to book a visit and the system.



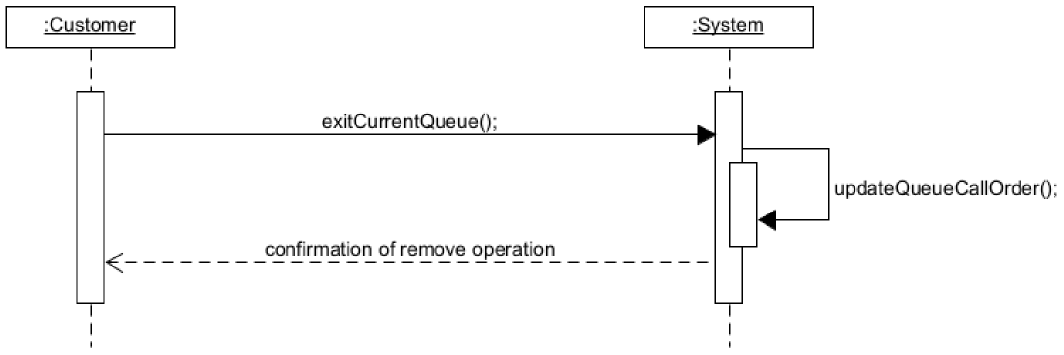
3.2.3.8 Booking deletion

Booking deletion sees customer as external actor which interacts only with the system digitally.



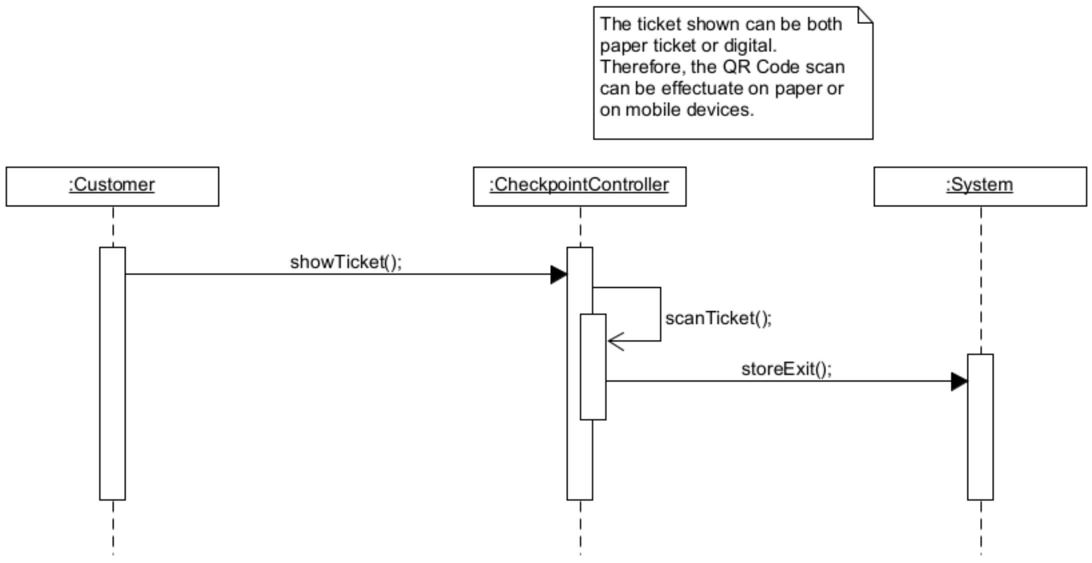
3.2.3.9 Queue exit

Queue exit operation is possible for all the people being in a queue online with a digital ticket. Customer is the only external actor present in this sequence derived by the use case.



3.2.3.10 Scanning at exit a customer

A checkpoint controller may scan a ticket of a customer who is about to exit the store.


The ticket shown can be both paper ticket or digital. Therefore, the QR Code scan can be effectuate on paper or on mobile devices.
:Customer
:CheckpointController
:System
showTicket();
scanTicket();
storeExit();

3.3 Performance requirements

In order to avoid misleading the user, the system should provide quick responses when it comes to real time queue management. The expected waiting time should be re-calculated once every minute, and should be sent to the user no later than 10 seconds after every calculation. For the same reason, whenever a QR code is scanned and approved, the system should register the event at most 5 seconds after it received its notification.
Visit booking is instead not affected by the same type of constraints, since due to its very nature a visit is booked with reasonable anticipation. The only constraint is that the booking should be registered by the system before a new booking is made for the same store.

3.4 Design constraints

The software is realized using some hardware pieces and software pieces. The software is constrained by three main categories: standards, hardware and other. Standard constraints are those limitation of the software regarding standards. Standard constraints regard the utilization of well-known standards in the system's realization. Hardware constraints are boundaries for the system's realization. Hardware constraints are those constraints regarding the architecture and the components type. Other constraints are a set of constraints more general and domain specific. The other constraints are those constraints specific for the S2B.

3.4.1 Standard compliance

The system must be compliant to the General Data Protection Regulation (GDPR)[4], as it is mandatory by law in the European Union and will ensure proper handling of users' data. Moreover the system must be compliant to the ISO/IEC 27001 international standard[5]. This would ensure that the best security practices are followed to develop the system.

3.4.2 Hardware limitations

S2B must be able to communicate with each possible ticket machine through an appropriate design parent and the usage of an appropriate communication method. The choice of the ticket machine must be independent by the system's implementation and must constrain as less as possible the store's choices.

3.4.3 Any other constraint

Actions like queuing online, booking a visit and receive notifications are all tracked permanently on a storage system and are accessible if needed. Data must be secured by replicating it frequently. All the visits saved on the storage system are never deleted before a month.

3.5 Software system attributes

3.5.1 Reliability

The system must have a high rejection of errors and should be extremely reliable. In such a situation, errors cannot happen in the queue management. If an error happens it may make people's life at risk, especially for weak categories like elderly people which are weaker to the virus. Specifically: the system must not have errors in the queue management independently by the number of people waiting in the queue and booking. The approach should be conservative, an error must let less people be in the store and not more people than how many are allowed to be. In the reliability expression the most important factors which have to be reduced are the detection time and the response time, which are the most critical. Indeed, recovery time and repair time, also if important, are less critical. This means that on MTTR components, the major contributors should be the repair time and recovery time.

3.5.2 Availability

The system is normally expected to run 24/7, except for short interruptions (4 hours maximum) due to maintenance. To minimize the damage caused by the temporary lack of the service, these interruptions need to happen when hardly any store is open, and hardly anyone will shop. As an example they could happen during the middle of the night. Moreover they will need to be notified at least 48 hours in advance to allow stores and customers to adapt. Overall the system will at least be available 99.9% of the time, corresponding to roughly 9 hours of downtime every year. In order to guarantee such high availability, possible failures need to be mitigated by introducing redundancy into the system infrastructure. In this way, a failure to a component will not cause the system to stop working, as there will be more components of the same type to redistribute the workload.

3.5.3 Security

The security measures are mainly aimed at preventing data leaks and system overloading.
The first goal is achieved by using a secure hash function to store passwords in the storage system, as well as a reliable data encryption algorithm for secure communication.
The second goal is instead achieved by appropriately limiting the user's freedom of action. While customers can freely register, store managers and employees will have their accounts manually created by a sys-admin, thus preventing malicious users from creating fake accounts for a non-existent store. Also, the system sets a limit to the maximum amount of time slots that a customer can book and queuing tickets that they can get. Each customer can only have one active ticket at a time, and cannot be in two queues at the same time. These measures will prevent malicious users from overloading a queue or from booking an excessive number of time slots, so that legitimate customers can regularly use the system.

3.5.4 Maintainability

The system will be monitored on a weekly basis, by collecting data about performance and anomalies/faults during its execution. Such data will be used to build useful statistics that will be used by a dedicated team in charge of maintenance. They will have to investigate deeper and possibly find parts in the code that jeopardize performance or cause faults. Once such parts were individuated, they will be appropriately modified by the team, so that the system can improve and evolve adequately. If necessary, the maintenance team can cooperate with the development section in order to ask for elucidations regarding the code and agree on possible solutions.
However, it is also desirable that our system does not require frequent maintenance. For this reason, the development team (or a dedicated testing team) should extensively test the source code and achieve coverage of 80% or above.
The system will have a modular structure that makes adding new features pretty simple. An appropriate design pattern should be chosen in order to achieve that. Furthermore, the modular structure will both allow to modify an existing component and/or add new modules that implement further functionalities.

3.5.5 Portability

Portability is a major concern for the system. All the user applications will therefore be developed to run on all modern desktop and mobile devices. User data can be accessed on multiple devices, by authenticating as the same user. The ticket machine application will be compatible with most smart ticket machines on the market, such that stores will be able to freely choose which ticket machine model to use.

4 Formal analysis using Alloy

With Alloy we want to verify some properties of the system. We introduce facts representing some assumptions on the system's configuration and we use time to denote the system's evolution in time. Our goal is to show that the system's properties are preserved by the operations that we defined in our predicates. Such operations include: getting in and out of a queue, entering or leaving a store, booking or deleting visits and printing tickets. We aim to prove that these operations do not violate some invariants of the system (e.g. customers controlled at the exit are always less or equal than the ones controlled at the entrance). To do so we use assertions to verify that, given the facts and the predicate, performing an operation on a consistent state leads to another consistent state.

Our code uses some built in libraries that deal with ordering and time. Such libraries require signatures to be **exact**, or else the predicates using them will not run properly. For this reason, we used **run commands** in which we specify many exact signatures.

4.1 Alloy source code

```
open util/ordering[QueueTicket]
open util/time

abstract sig Person {}
abstract sig Ticket {
    ticketOwner: one Customer,
}

// general entities
sig DateTime {}

// entities which extend Person
sig StoreManager extends Person {}
sig Customer extends Person {}
sig CheckpointController extends Person {
    //tickets admitted into the store
    controllerCheckIns: dynamicSet[Ticket],
    //tickets scanned at the exit
    controllerCheckOuts: dynamicSet[Ticket],
}

// entities extending ticket
sig BookingTicket extends Ticket {}
sig QueueTicket extends Ticket {}

// store's signature
sig Store {
    storeManagers: some StoreManager,
    storeControllers: some CheckpointController,
    storeTicketMachines: some TicketMachine,
    storeProducts: some Item,
    //customers currently inside the store
    storeCustomersInStore: dynamicSet[Customer],
    //current queue for the store
    storeQueue: one Queue,
    //current bookings for the store
    storeBookings: dynamicSet[Booking],
    /* tickets that have been used to enter the store, but not necessarely
        to exit */
    storeUsedTickets: dynamicSet[Ticket],
    /* tickets that have not been used to enter the store, and are no more
         valid */
    storeNotUsedInvalidTickets: dynamicSet[Ticket]
}{
    #storeControllers > 0
    #storeTicketMachines > 0
}

sig TicketMachine {
    machinePrintedTickets: dynamicSet[QueueTicket]
}

sig Category {}

sig Item {
    itemCategory: one Category
}

// bookings and queues
sig Booking {
    bookingTicket: one BookingTicket,
    bookingDateTime: one DateTime,
    bookingItems: set Item,
    bookingCategories: set Category
}

sig Queue {
    queueTickets: dynamicSet[QueueTicket],
}

/***********************
 *                       *
 *                       *
 *        FACTS          *
 *                       *
 *                       *
 ***********************/

/*-------------------------------------------------------------------------
    Ubiquity Facts: in this section we include facts stating that some
    entities cannot be shared by other entities.
    For instance: customers cannot be simoultaneously inside two stores
  -------------------------------------------------------------------------*/
/* No customer has superpowers that allow them to be in two stores at the
    same time */
fact noUbiquitySuperpowers{
    all t:Time | no disj s1,s2:Store |
        #(s1.storeCustomersInStore.t & s2.storeCustomersInStore.t) > 0
}

//No queue is owned by two stores
fact noSharedQueues{
    all  q:Queue | one s:Store | q in s.storeQueue
}

//No controller is owned by two stores
fact noSharedControllers{
    all  chk:CheckpointController | one s:Store | chk in s.storeControllers
}

//No ticket machine is shared by two stores
fact noSharedTicketMachine{
    all  tm:TicketMachine | one s:Store | tm in s.storeTicketMachines
}

//Each booking ticket is for one booking only
fact noSharedBookingTicket{
    all bt:BookingTicket | one b:Booking | b.bookingTicket = bt
}
```

```
//Every ticket is for only one store
fact ticketsAreForAtMostOneStore {
    all t: Ticket, time:Time | one s: Store |
        t in (s.storeUsedTickets.time + s.storeNotUsedInvalidTickets.time +
        s.storeBookings.time.bookingTicket + s.storeQueue.queueTickets.time)
}

//Every booking is for only one store
fact bookingsAreForAtMostOneStore {
    all b: Booking, t:Time| lone s: Store | b in s.storeBookings.t
}


fact ticketMustBeOnlyOfOneType {
    //tickets can either be:
    all t: Ticket, s: Store, time:Time |
        //used
        (t in s.storeUsedTickets.time implies
            t not in (s.storeNotUsedInvalidTickets.time +
            s.storeQueue.queueTickets.time +
            s.storeBookings.time.bookingTicket))
        and
        //not used and invalid
        (t in s.storeNotUsedInvalidTickets.time implies
            t not in (s.storeUsedTickets.time + s.storeQueue.queueTickets.time
            + s.storeBookings.time.bookingTicket))
        and
        //valid for booking
        (t in s.storeBookings.time.bookingTicket implies
            t not in (s.storeNotUsedInvalidTickets.time +
            s.storeQueue.queueTickets.time + s.storeUsedTickets.time))
        and
        //valid for queueing
        (t in s.storeQueue.queueTickets.time implies
            t not in (s.storeNotUsedInvalidTickets.time +
            s.storeUsedTickets.time + s.storeBookings.time.bookingTicket))
}

//Tickets may be printed at most once
fact ticketMayBePrintedByAtMostOneTicketMachine {
    all t: Ticket, time:Time| lone tm: TicketMachine |
        t in tm.machinePrintedTickets.time
}

/* A customer cannot be in a queue and at the same time in the store owning
    that queue */
fact queueCannotContainInStoreCustomer {
    all s: Store, t:Time | no c: Customer |
        c in s.storeCustomersInStore.t and
        c in s.storeQueue.queueTickets.t.ticketOwner
}

/*------------------------------------------------------------------------
    No random changes : the following facts state that no new entities
    appear in time unless an operation makes them appear.
    For instance, no new customer can appear in a store unless someone
    entered
-----------------------------------------------------------------------*/
//A new customer appears iff someone enters or exits
fact CustomersDontMultiplyRandomly{
    all t:Time, s:Store | (some c:Customer |
        c in s.storeCustomersInStore.(t.next)
        and c not in s.storeCustomersInStore.(t) )
        iff
        ((some qt:QueueTicket |enterStoreQueue[s,qt,t]) or
        (some b:Booking | enterStoreBooking[s,b,t]))
}

//A customer disappers iff somedy left the store
fact CustomersDontVanishRandomly{
    all t:Time, s:Store | (some c:Customer | c in  s.storeCustomersInStore.(t)
        and c not in s.storeCustomersInStore.(t.next) )
        iff
        ( some tick:Ticket | leaveStore[s,tick,t])
}

//A ticket leaves a queue iff somebody left the queue or entered the store
```

```
fact queueticketsDontDisappearRandomly{
    all t:Time, s:Store |
        (some qt:QueueTicket |
            qt in  s.storeQueue.queueTickets.t and
            qt not in s.storeQueue.queueTickets.(t.next))
        iff
        (some qt:QueueTicket, c:Customer |
            leaveQueue[s.storeQueue,c, qt,t] or enterStoreQueue[s, qt,t ])
}

//A ticket appears in a queue iff somebody joined the queue
fact queueticketsDontAppearRandomly{
    all t:Time, s:Store |
        (some qt:QueueTicket |
            qt in  s.storeQueue.queueTickets.(t.next) and
            qt not in s.storeQueue.queueTickets.(t) )
                iff
            (some qt:QueueTicket, c:Customer | joinQueue[s.storeQueue,c,qt,t])
}

//Bookings appear iff a new visit is booked
fact bookingsDontAppearRandomly{
    all t:Time, s:Store |
        (some b:Booking|
            b in s.storeBookings.(t.next) and b not in s.storeBookings.(t) )
                iff
            ( some b:Booking, c:Customer | bookVisit[s,b,c,t])
}

/*Bookings disappear iff a visit is deleted or somebody enters the store
    by using a booking */
fact bookingsDontAppearRandomly{
    all t:Time, s:Store |
        (some b:Booking|
            b in  s.storeBookings.(t) and b not in s.storeBookings.(t.next))
        iff
        (some b:Booking, c:Customer |
            deleteAVisit[s,c,b,t] or enterStoreBooking[s,b,t])
}

//A new control appears iff somebody entered the store
fact ControlsDontMultiplyRandomly{
    all t:Time, s:Store |
        (some tick:Ticket |
            tick in  s.storeControllers.controllerCheckIns.(t.next) and
            tick not in s.storeControllers.controllerCheckIns.t )
        iff
        ( (some qt:QueueTicket |
            enterStoreQueue[s,qt,t]) or
            ( some b:Booking | enterStoreBooking[s,b,t]) )
}

/*---------------------------------------------------------------------
    No Useless Entities: the following facts state that if
    an entity exists, it must belong somewhere, i.e. we
    do not want isolted/useless entities. For instance, we can sureòy
    have unemployed Store Managers in real life, but they are not
    relevant for our analysis, therefore we include these facts to
    avoid their generation
----------------------------------------------------------------------*/
//Store managers manage at least one store
fact storeManagersManageAtLeastOneStore {
    all sm: StoreManager | some s: Store | sm in s.storeManagers
}

//Items are at least in one store
fact itemsAreAtLeastInOneStore {
    all i: Item | some s: Store | i in s.storeProducts
}

//Categories contain at least one item
fact categoriesContainsAtLeastOneItem {
    all c: Category | some i: Item | c = i.itemCategory
}


/* If tickets are printed, they are printed by a machine of the store they
    are for */
```

```
fact printedTicketsAreInQueueOrAreUsed {
    all t: Ticket , time:Time, s: Store |
        t in s.storeTicketMachines.machinePrintedTickets.time
        implies
        (t in s.storeQueue.queueTickets.time or
        t in s.storeNotUsedInvalidTickets.time
        or t in s.storeUsedTickets.time)
}

//All used tickets have been checked at entrance
fact usedTicketsHaveBeenScanned {
    all t: Ticket, s: Store , time:Time|
        t in s.storeUsedTickets.time implies
        t in s.storeControllers.controllerCheckIns.time

    all t: Ticket, time:Time, s: Store |
        t in s.storeControllers.controllerCheckIns.time
        implies
        (t in s.storeUsedTickets.time or
        t in s.storeNotUsedInvalidTickets.time or
        t in s.storeBookings.time.bookingTicket)
}

/************************
 *                      *
 *                      *
 *      PREDICATES      *
 *                      *
 *                      *
 ************************/

//1. Join a queue
pred joinQueue[q:Queue, c:Customer, qt:QueueTicket, t:Time]{
    //preconditions
    //c owns the ticket
    qt.ticketOwner = c
    //the ticket is not already in the queue
    qt not in q.queueTickets.t
    //customer is not already in the queue
    no tick:QueueTicket |
        tick in q.queueTickets.t and tick.ticketOwner = c
    //the ticket must not have been used
    no chk:CheckpointController |
        qt in (chk.controllerCheckIns.t + chk.controllerCheckOuts.t)

    //postconditions
    //the ticket is now in the queue
    all ticket:QueueTicket |
        (ticket in q.queueTickets.(t.next))
        iff
        (ticket in q.queueTickets.t or ticket = qt)
    //the new ticket is greater than any other ticket in the queue
    all ticket:QueueTicket |
        ticket in q.queueTickets.(t) => lt[ticket,qt]
}

//2. Leave a queue
pred leaveQueue[q:Queue, c:Customer, qt:QueueTicket, t:Time]{
    //preconditions
    qt.ticketOwner = c
    qt in q.queueTickets.t
    //the ticket must not have been used
    no chk:CheckpointController |
        qt in (chk.controllerCheckIns.t + chk.controllerCheckOuts.t)

    //postconditions
    //the ticket is no longer in the queue
    all ticket:QueueTicket |
        (ticket in q.queueTickets.(t.next))
        iff
        (ticket in q.queueTickets.t and ticket != qt)
}

//3. Book a visit
pred bookVisit[s:Store, b:Booking, c:Customer, t:Time]{
    //preconditions
    /*the customer must own the new booking*/
    b.bookingTicket.ticketOwner = c
    /*the new booking cannot overlap with any already existing booking for s*/
    no ovlpBooking: Booking |
        ovlpBooking in s.storeBookings.t and
        ovlpBooking.bookingDateTime = b.bookingDateTime and
        b.bookingTicket.ticketOwner = ovlpBooking.bookingTicket.ticketOwner
    /*the new booking cannot overlap with any already existing booking for
    any other store*/
    all s2:Store |
        s != s2 implies
        (no ovlpBooking: Booking |
            ovlpBooking.bookingTicket.ticketOwner = c and
            ovlpBooking.bookingDateTime = b.bookingDateTime and
            ovlpBooking in s2.storeBookings.t)

    //postconditions
    /*at time t.next, the store will only have the bookings that it had
        before + b*/
    all book:Booking |
        (book in s.storeBookings.(t.next))
        iff
        (book = b or book in s.storeBookings.t)
}

//4. Delete a visit
pred deleteAVisit[ s:Store, c:Customer, b:Booking, t:Time] {
    // preconditions
    /*b must be in the store before*/
    b in s.storeBookings.t
    /*b's customer must be c*/
    c = b.bookingTicket.ticketOwner

    // postconditions
    /*if a booking was in the store before and is different from b, it
    is in the store afterwards */
    all book:Booking |
        (book in s.storeBookings.t and book != b)
        iff
        book in s.storeBookings.(t.next)

}

//5.Enter a store from its queue
pred enterStoreQueue[s:Store, qt:QueueTicket, t:Time]{
    //preconditions
    /*entering customer is in the queue for s*/
    qt in s.storeQueue.queueTickets.t
    /*qt is the first of the queue*/
    all ticket:QueueTicket |
        (ticket in s.storeQueue.queueTickets.t and ticket != qt)
        implies
        lt[qt,ticket]
    /*the ticket must not have been used*/
    no chk:CheckpointController |
        qt in (chk.controllerCheckIns.t + chk.controllerCheckOuts.t)

    //postconditions
    /*the owner of the ticket is now in the store*/
    all c:Customer |
        c in s.storeCustomersInStore.(t.next)
        iff
        (c=qt.ticketOwner or c in s.storeCustomersInStore.t)
    /*the owner of the ticket is removed from the queue*/
    all ticket:QueueTicket |
        (ticket in s.storeQueue.queueTickets.(t.next))
        iff
        (ticket in s.storeQueue.queueTickets.t and ticket != qt)
    one chk:CheckpointController |
        qt in chk.controllerCheckIns.(t.next) and
        chk in s.storeControllers
    no chk:CheckpointController |
        qt in chk.controllerCheckIns.(t.next) and
        chk not in s.storeControllers
    all chk:CheckpointController |
        chk.controllerCheckOuts.(t.next) = chk.controllerCheckOuts.(t)
    all chk:CheckpointController |
        qt not in chk.controllerCheckIns.(t.next)
        iff
        chk.controllerCheckIns.(t.next) = chk.controllerCheckIns.(t)
    all chk:CheckpointController |
        qt  in chk.controllerCheckIns.(t.next)
        iff
        chk.controllerCheckIns.(t.next) = chk.controllerCheckIns.(t) + qt

}

//6. Enter a store by booking
pred enterStoreBooking[s:Store, b:Booking, t:Time]{

    //preconditions
    /*is an active booking at the time of entering*/
    b in s.storeBookings.t
    /*the ticket related to b must not have been used*/
    no chk:CheckpointController |
        b.bookingTicket in (chk.controllerCheckIns.t +
            chk.controllerCheckOuts.t)

    //postconditions
    /*the booking is removed from the active bookings*/
    all book:Booking | book in s.storeBookings.(t.next)
        iff
        ( book != b and book in s.storeBookings.t)
    /*the owner of the ticket is now in the store*/
    all c:Customer |
        c in s.storeCustomersInStore.(t.next)
        iff
        (c=b.bookingTicket.ticketOwner or c in s.storeCustomersInStore.t)
    one chk:CheckpointController |
        b.bookingTicket in chk.controllerCheckIns.(t.next) and
        chk in s.storeControllers and
        chk in s.storeControllers
    no chk:CheckpointController |
        b.bookingTicket in chk.controllerCheckIns.(t.next) and
        chk not in s.storeControllers
    all chk:CheckpointController |
        chk.controllerCheckOuts.(t.next) = chk.controllerCheckOuts.(t)
    all chk:CheckpointController |
        b.bookingTicket not in chk.controllerCheckIns.(t.next)
        iff
        chk.controllerCheckIns.(t.next) = chk.controllerCheckIns.(t)
    all chk:CheckpointController |
        b.bookingTicket in chk.controllerCheckIns.(t.next)
        iff
        chk.controllerCheckIns.(t.next) = chk.controllerCheckIns.(t) +
                                          b.bookingTicket
}

//7.Leave a store
pred leaveStore[s:Store, tick:Ticket, t:Time]{
    //preconditions
    /*the owner of tick is in the store*/
    tick.ticketOwner in s.storeCustomersInStore.t
    tick in s.storeControllers.controllerCheckIns.t

    //postconditions
    /*the customer is no longer in the store*/
    all c:Customer |
        c in s.storeCustomersInStore.(t.next)
        iff
        ( c != tick.ticketOwner and c in s.storeCustomersInStore.(t))
    /*customer has been checked at the exit*/
    one chk:CheckpointController |
        tick in chk.controllerCheckOuts.(t.next) and
        chk in s.storeControllers
    no chk:CheckpointController |
        tick in chk.controllerCheckOuts.(t.next) and
        chk not in s.storeControllers
    all chk:CheckpointController |
        chk.controllerCheckIns.(t.next) = chk.controllerCheckIns.(t)
    all chk:CheckpointController |
        tick not in chk.controllerCheckOuts.(t.next)
        implies
        chk.controllerCheckOuts.(t.next) = chk.controllerCheckOuts.(t)
    all chk:CheckpointController |
        tick  in chk.controllerCheckOuts.(t.next)
        implies
        chk.controllerCheckOuts.(t.next) = chk.controllerCheckOuts.(t) + tick
```

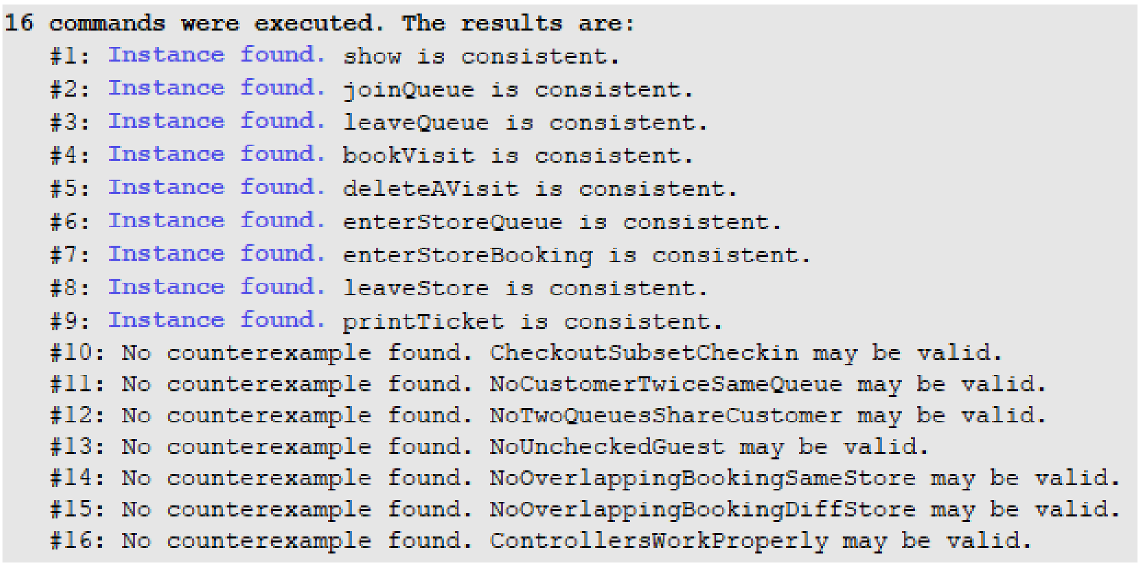
```
}

//8. Print a ticket for a store
pred printTicket[s:Store, qt:QueueTicket ,c:Customer, t:Time]{
    //preconditions
    qt.ticketOwner = c
    c not in s.storeCustomersInStore.t
    qt not in s.storeTicketMachines.machinePrintedTickets.t
    qt not in s.storeQueue.queueTickets.t
    qt not in s.storeControllers.controllerCheckIns.t
    qt not in s.storeControllers.controllerCheckOuts.t

    //postconditions
    s.storeTicketMachines.machinePrintedTickets.(t.next) =
        s.storeTicketMachines.machinePrintedTickets .t + qt
    s.storeQueue.queueTickets.(t.next) = s.storeQueue.queueTickets.t + qt
    all qn1: QueueTicket |
        (qn1 in s.storeQueue.queueTickets.(t.next) and qn1 != qt )
        implies
        lt[qn1, qt]
}

/************************
 *                      *
 *                      *
 *      AUXILIARY       *
 *      PREDICATES      *
 *                      *
 *                      *
 ************************/

pred checkoutSubsetCheckin[s:Store, t:Time]{
     s.storeControllers.controllerCheckOuts.t
        in
    s.storeControllers.controllerCheckIns.t
}

pred storeOutSubsetIns[s:Store, t:Time]{
    s.storeControllers.controllerCheckOuts.t
        in
    s.storeControllers.controllerCheckIns.t
}

pred noCustomerAppearsTwice[q:Queue, t:Time]{
    no disj qt1, qt2: QueueTicket |
        qt1 in q.queueTickets.t and
        qt2 in q.queueTickets.t and
        qt1.ticketOwner = qt2.ticketOwner
}

pred disjointQueues[q1,q2:Queue, t:Time]{

    no c:Customer |
        c in q1.queueTickets.t.ticketOwner and
        c in q2.queueTickets.t.ticketOwner
}

pred allGuestsChecked[s:Store, t:Time]{
    all c:Customer |
        c in s.storeCustomersInStore.t
        implies
        (some chk:CheckpointController |
            c in chk.controllerCheckIns.t.ticketOwner)
}

pred hasOverlappingBooking[ s:Store, t:Time]{
    some disj b1,b2:Booking |
        b1 in s.storeBookings.t and
        b2 in s.storeBookings.t and
        b1.bookingDateTime = b2.bookingDateTime and
        b1.bookingTicket.ticketOwner = b2.bookingTicket.ticketOwner
}

pred haveOverlappingBooking[ s1,s2:Store, t:Time]{
    some disj b1,b2:Booking |
        b1 in s1.storeBookings.t and
        b2 in s2.storeBookings.t and
        b1.bookingDateTime = b2.bookingDateTime and
        b1.bookingTicket.ticketOwner = b2.bookingTicket.ticketOwner
}

pred checkedInButNotOut[c:Customer, s:Store, t:Time]{
    some tick:Ticket |
        tick.ticketOwner = c and
        tick in s.storeControllers.controllerCheckIns.t and
        tick not in s.storeControllers.controllerCheckOuts.t
}

/************************
 *                      *
 *                      *
 *      ASSERTIONS      *
 *                      *
 *                      *
 ************************/
/* Our assertions are aimed at proving that the operations that were defined
    in the predicates do not alter the consistency of a state.
    At first, we prove that the chosen property is respected in a basic case.
    Then, we show that if we are in a state that respects the property, and
    we perform some operation on the entity we are working with, the state in
    the following time will also respect such property.
    For more details, please check the reference section.
*/

/*This assertion proves that at any time and for every controller, the
    tickets that they checkedOut are a subset of those they checkedIn */
assert CheckoutSubsetCheckin{
    /*Base Case: if no one was controlled for a store S, the property holds*/
    all s:Store, t:Time |
        #s.storeControllers.controllerCheckOuts.t = 0 and
        #s.storeControllers.controllerCheckIns.t = 0
            implies
        storeOutSubsetIns[s,t]

    /*Inductive Steps: if we start from a consistent state, and some
    customer enters the Store or leaves it, the property still holds*/
    all s:Store, t:Time |
        storeOutSubsetIns[s,t] and
        (some qt:QueueTicket | enterStoreQueue[s,qt,t])
            implies
        storeOutSubsetIns[s,t.next]
    all s:Store, t:Time |
        storeOutSubsetIns[s,t] and
        (some b:Booking| enterStoreBooking[s,b,t])
            implies
        storeOutSubsetIns[s,t.next]
    all s:Store, t:Time |
        storeOutSubsetIns[s,t] and
        (some tick:Ticket | leaveStore[s,tick,t])
            implies
        storeOutSubsetIns[s,t.next]
}

//This assertion proves that no customer can appear twice in the same queue
assert NoCustomerTwiceSameQueue{
    /*Base Case: the property holds for an empty queue*/
    all q:Queue, t:Time |
        #q.queueTickets.t = 0
        implies
        noCustomerAppearsTwice[q,t]

    /*Inductive Steps: if we start from a consistent queue and some customer
    joins the queue or leaves it, the property still holds*/
    all q:Queue, t:Time |
        noCustomerAppearsTwice[q,t] and
        (some c:Customer, qt:QueueTicket | joinQueue[q,c,qt,t])
            implies
        noCustomerAppearsTwice[q,t.next]
    all q:Queue, t:Time |
        noCustomerAppearsTwice[q,t] and
        (some c:Customer, qt:QueueTicket | leaveQueue[q,c,qt,t])
            implies
        noCustomerAppearsTwice[q,t.next]
}

//This assertion proves that no customer can be twice in different queues
```

```
assert NoTwoQueuesShareCustomer{
    /*Base Case: the property holds for two empty queues*/
    all disj q1,q2:Queue , t:Time |
        #q1.queueTickets=0 and
        #q2.queueTickets=0
            implies
        disjointQueues[q1,q2,t]

    /*Inductive Steps: if we start from two consistent queue sand some
    customer joins on of the queues or leaves it, the property still holds*/
    all disj q1,q2:Queue , t:Time |
        disjointQueues[q1,q2,t] and
        (some c:Customer, qt:QueueTicket | joinQueue[q1,c,qt,t])
            implies
        disjointQueues[q1,q2,t.next]
    all disj q1,q2:Queue , t:Time |
        disjointQueues[q1,q2,t] and
        (some c:Customer, qt:QueueTicket | leaveQueue[q1,c,qt,t])
            implies
        disjointQueues[q1,q2,t.next]
}

/*This assertion proves that any customer that is inside the store was
    checkedIn at some point*/
assert NoUncheckedGuest{
    /*Base Case: the property holds for an empty store*/
    all s:Store, t:Time |
        #s.storeCustomersInStore = 0
        implies
        allGuestsChecked[s,t]

    /*Inductive Steps: if we start from a consistent state, and some
    customer enters the Store or leaves it, the property still holds*/
    all s:Store, t:Time |
        allGuestsChecked[s,t] and
        (some qt:QueueTicket | enterStoreQueue[s,qt,t])
            implies
        allGuestsChecked[s,t.next]
    all s:Store, t:Time |
        allGuestsChecked[s,t] and
        (some b:Booking| enterStoreBooking[s,b,t])
            implies
        allGuestsChecked[s,t.next]
    all s:Store, t:Time |
        allGuestsChecked[s,t] and
        (some tick:Ticket| leaveStore[s,tick,t])
            implies
        allGuestsChecked[s,t.next]
}

//This assertion proves that a store cannot have two overlapping bookings
assert NoOverlappingBookingSameStore{
    /*Base Case: the property holds for an Store without bookings*/
    all s:Store, t:Time |
        #s.storeBookings.t = 0
        implies
        !hasOverlappingBooking[s,t]

    /*Inductive Steps: if we start from a consistent state, and some visit
    is either booked or deleted, the property still holds*/
    all s:Store, t:Time |
        !hasOverlappingBooking[s,t] and
        (some c:Customer,b:Booking | bookVisit[s, b,c,t])
            implies
        !hasOverlappingBooking[s,t]
    all s:Store, t:Time |
        !hasOverlappingBooking[s,t] and
        (some c:Customer,b:Booking | deleteAVisit[s, c,b,t])
            implies
        !hasOverlappingBooking[s,t]
}

//This assertion proves that no two stores can have overlapping bookings
assert NoOverlappingBookingDiffStore{
    /*Base Case: the property holds for two stores both without a booking*/
    all s1,s2:Store, t:Time |
        #s1.storeBookings.t = 0 and
        #s2.storeBookings.t = 0
```

```
            implies
        !haveOverlappingBooking[s1,s2,t]

    /*Inductive Steps: if we start from a consistent state, and some visit
    is either booked or deleted, the property still holds*/
    all s1,s2:Store, t:Time |
        !haveOverlappingBooking[s1,s2,t] and
        (some c:Customer,b:Booking | bookVisit[s1, b,c,t])
            implies
        !haveOverlappingBooking[s1,s2,t]
    all s1,s2:Store, t:Time |
        !haveOverlappingBooking[s1,s2,t] and
        (some c:Customer,b:Booking | deleteAVisit[s1, c,b,t])
            implies
        !haveOverlappingBooking[s1,s2,t]
}

/*This assertion proves that if a customer is inside a store it has some
    ticket that was checked in but not checkedout*/
assert ControllersWorkProperly{
    /*Base Case: the property holds for an empty store*/
    all s:Store, t:Time |
        #s.storeCustomersInStore.t = 0
        implies
        ( all c:Customer |
            c in s.storeCustomersInStore.t
            implies
            checkedInButNotOut[c,s,t])

    /*Inductive Steps: if we start from a consistent state, and some
    customer enters the Store or leaves it, the property still holds*/
    all s:Store, t:Time |
        ( all c:Customer |
            c in s.storeCustomersInStore.t
            implies checkedInButNotOut[c,s,t]) and
        (some qt:QueueTicket | enterStoreQueue[s,qt,t])
            implies
        (all c:Customer |
            c in s.storeCustomersInStore.t
            implies checkedInButNotOut[c,s,t.next])

    all s:Store, t:Time |
        ( all c:Customer |
            c in s.storeCustomersInStore.(t.next)
            implies checkedInButNotOut[c,s,t]) and
        (some b:Booking|
            enterStoreBooking[s,b,t])
            implies
            (all c:Customer |
                c in s.storeCustomersInStore.(t.next)
                implies checkedInButNotOut[c,s,t.next])

    all s:Store, t:Time |
        ( all c:Customer |
            c in s.storeCustomersInStore.t
            implies checkedInButNotOut[c,s,t]) and
        (some tick:Ticket| leaveStore[s,tick,t])
            implies
        (all c:Customer |
            c in s.storeCustomersInStore.(t.next)
            implies checkedInButNotOut[c,s,t.next])
}

pred show{}
run show for 4 but exactly 2 Store, exactly 2 Queue


/********************
*   Run Predicates  *
*********************/
run joinQueue for 7 but
    exactly 2 Store, exactly 2 Queue, exactly 6 QueueTicket,
    exactly 4 Time, exactly 2 CheckpointController
run leaveQueue for 7 but
    exactly 1 Store, exactly 1 Queue, exactly 6 QueueTicket,
    exactly 4 Time, exactly 2 CheckpointController
run bookVisit  for 7 but
    exactly 2 Store, exactly 2 Queue, exactly 6 QueueTicket,
    exactly 4 Time, exactly 2 CheckpointController, exactly 3 Booking,
    exactly 3 BookingTicket
run deleteAVisit  for 7 but
    exactly 1 Store, exactly 1 Queue, exactly 6 QueueTicket,
    exactly 5 Time, exactly 2 CheckpointController, exactly 3 Booking,
    exactly 3 BookingTicket
run enterStoreQueue for 7 but
    exactly 1 Store, exactly 1 Queue, exactly 6 QueueTicket,
    exactly 4 Time, exactly 2 CheckpointController, exactly 4 Customer
run enterStoreBooking for 7 but
    exactly 1 Store, exactly 1 Queue, exactly 6 QueueTicket,
    exactly 4 Time, exactly 2 CheckpointController, exactly 3 Booking,
    exactly 3 BookingTicket
run leaveStore for 7 but
    exactly 1 Store, exactly 1 Queue, exactly 6 QueueTicket,
    exactly 4 Time, exactly 2 CheckpointController, exactly 4 Customer
run printTicket for 7 but
    exactly 1 Store, exactly 1 Queue, exactly 6 QueueTicket,
    exactly 4 Time, exactly 2 CheckpointController, exactly 4 Customer

/*********************
*   Run Assertions   *
*********************/
check CheckoutSubsetCheckin
check NoCustomerTwiceSameQueue
check NoTwoQueuesShareCustomer
check NoUncheckedGuest
check NoOverlappingBookingSameStore
check NoOverlappingBookingDiffStore
check ControllersWorkProperly
```

4.2 Alloy analysis

Given the alloy code we run the assertion and found there are no errors. This is the result we obtain:



```
16 commands were executed. The results are:
   #1: Instance found. show is consistent.
   #2: Instance found. joinQueue is consistent.
   #3: Instance found. leaveQueue is consistent.
   #4: Instance found. bookVisit is consistent.
   #5: Instance found. deleteAVisit is consistent.
   #6: Instance found. enterStoreQueue is consistent.
   #7: Instance found. enterStoreBooking is consistent.
   #8: Instance found. leaveStore is consistent.
   #9: Instance found. printTicket is consistent.
   #10: No counterexample found. CheckoutSubsetCheckin may be valid.
   #11: No counterexample found. NoCustomerTwiceSameQueue may be valid.
   #12: No counterexample found. NoTwoQueuesShareCustomer may be valid.
   #13: No counterexample found. NoUncheckedGuest may be valid.
   #14: No counterexample found. NoOverlappingBookingSameStore may be valid.
   #15: No counterexample found. NoOverlappingBookingDiffStore may be valid.
   #16: No counterexample found. ControllersWorkProperly may be valid.
```

5 Effort spent

This part is the part which summarize the effort spent by each member of the team in the documentation building process.

Davide Li Calsi

Introduction	2hrs
Overall description	5.5hrs
Requirements	13hrs
Alloy	31hrs
Revision	4hrs

Andrea Alberto Marchesi

Introduction	2hrs
Overall description	7hrs
Requirements	13hrs
Alloy	5hrs
Revision	2.5hrs

Marco Petri

Introduction	2.5hrs
Overall description	5hrs
Requirements	12hrs
Alloy	27hrs
Revision	2.5hrs

All

Meetings	20hrs

6 References

This section includes all the document and references used to produce this documentation. We include in this part of the document every website or document used to

1. http://dati.istat.it/Index.aspx?DataSetCode=DCCV_ICT: is an ISTAT statistic about the usage internet diffusion in Italy;
2. Oracle Directory Server Enterprise Edition Deployment Planning Guide: is a document about the operations needed in the process of designing a system [Chapter 12];
3. https://docs.oracle.com/cd/E20295_01/html/821-1217/fjdch.html#scrolltoc;
4. 29148-2018 - 29148-2018 - ISO/IEC/IEEE International Standard - Systems and software engineering -- Life cycle processes -- Requirements engineering - IEEE Standard: ISO/IEC/IEEE document about the building and designing of a system and RASD definition;
5. https://gdpr-info.eu/: contains the official PDF of the Regulation (EU) 2016/679 (General Data Protection Regulation).
6. https://www.iso.org/isoiec-27001-information-security.html: ISO/IEC 27001 international standard.
7. https://alloy.readthedocs.io/en/latest/modules/ordering.html details about the ordering module in Alloy
8. https://alloy.readthedocs.io/en/latest/modules/time.html details about the time module in Alloy
9. http://alloytools.org/tutorials/day-course/s4_dynamic.pdf paper from MIT about the use of Alloy to verify properties