People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University M'Hamed BOUGARA – Boumerdes

Institute of Electrical and Electronic Engineering

Department of Electronics

Final Year Project Report Presented in Partial Fulfilment of the Requirements for the Degree of

MASTER

In **Electronics**

Option: **Computer Engineering**

Title

Robot Based Drive-Through System For a Super Market

Presented by

- **BOUDEBOUDA Walid**

Supervisor

Dr. BENZEKRI

Registration Number:……../2019

Abstract

This report describes the design and implementation of a robot based drive-through system for a super market. Drive-through is a type of service provided by businesses to allow their customers to purchase products without leaving their cars.

The systems consists of three parts : an Android application that is used by the customers to remotely select the desired products, a Java program to process the orders, manage the products and customers database, control the robot, and monitor the state of the orders and the robot position on the map in real time. Finally a mobile robot controlled by an ESP32 Microcontroller used to collect the products in the supermarket.

Orders are sent from the android application to the java program through internet, whereas the communication between the java program and the robot is done via wireless WIFI connection in a local area network.

The results obtained were satisfactory for both the software and hardware parts, even though some constraints were faced on the hardware one.

Dedication

I would like to dedicate this work to my parents and my brother who supported me through my entire curriculum.

Acknowledgements

All praise is due to Allah the most gracious and the most merciful, whom with his willing, gave me the opportunity to complete this Project.

I would like to express my special gratitude to my supervisor Dr.BENZEKRI for his patience and his support on the way, and for his useful comments, remarks, and engagement through the completion of this master thesis.

My gratitude also goes to my family and friends, and all the teachers who encouraged and supported me during the past 5 years at the institute.

Table of content

ABSTRACT ..... **I**

DEDICATION ..... **II**

ACKNOWLEDGEMENTS ..... **III**

TABLE OF CONTENT ..... **IV**

LIST OF TABLES ..... **VI**

LIST OF FIGURES ..... **VII**

LIST OF ABBREVIATIONS AND ACRONYMS ..... **IX**

CHAPTER 1 INTRODUCTION **1**

1.1. INTRODUCTION 1

1.2. MOTIVATION 1

1.3. RELATED WORKS 2

1.4. OVERALL SYSTEM DESCRIPTION 2

1.5. ORGANIZATION OF THE REPORT : 3

CHAPTER 2 THEORETICAL BACKGROUND **4**

2.1. INTRODUCTION 4

2.2. THE ESP32 MICROCONTROLLER 4

2.2.1. ESP32 chip *4*

2.2.2. ESP32 development board (DOIT DevKit v1) *6*

2.2.3. Software development environment *7*

2.3. ANDROID APPLICATION 7

2.3.1. Android OS *7*

2.3.2. Android studio *8*

2.3.3. Activities *9*

2.3.4. Activity lifecycle *9*

2.3.5. UI Design *11*

2.3.6. RecyclerViews *12*

2.3.7. AsynchronousTasks *13*

2.4. JAVAFX 14

CHAPTER 3 HARDWARE SYSTEM DESIGN 15

3.1. INTRODUCTION 15

3.2. ROBOT CHASSIS 15

3.3. ELECTRONIC CIRCUIT 16

3.3.1. Motors control unit 16

3.3.2. Feedback unit 17

3.3.3. Overall circuit diagram 19

CHAPTER 4 SOFTWARE SYSTEM DESIGN 20

4.1. INTRODUCTION 20

4.2. NETWORK CONFIGURATION 20

4.2.1. Port forwarding 21

4.2.2. Dynamic Host Configuration Protocol reservation (static DHCP) 21

4.2.3. Dynamic Domain Name Server (DDNS) 21

4.3. ANDROID APPLICATION 22

4.3.1. LoginActivity 24

4.3.1. ProductsActivity 25

4.3.2. OrderActivity 26

4.3.3. MyOrdersActivity 27

4.4. THE JAVA DESKTOP PROGRAM 28

4.4.1. Classes organization 29

4.4.2. Background threads 32

4.4.3. Database design 35

4.4.4. Graphical User Interface Design 37

4.4.5. Map representation 42

4.4.6. Path planning 43

4.5. ROBOT SOFTWARE 49

CHAPTER 5 CONCLUSION 55

REFERENCES 56

List of tables

TABLE 3-1 ROBOT MOVE COMBINATIONS 17

TABLE 4-1 NEEDED ORIENTATION WITH RESPECT TO (CURRENT POSITION - NEXT POSITION) 51

TABLE 4-2 CALCULATION OF NUMBER AND DIRECTION OF ROTATION 51

List of figures

Figure 1.1 Overall system block diagram 3

Figure 2.1 ESP32 functional block diagram [4] 5

Figure 2.2 DOIT Esp32 DevKit v1 6

Figure 2.3 DOIT ESP32 DEVKit v1 pin mapping 7

Figure 2.4 A simplified illustration of the activity lifecycle [9]. 10

Figure 2.5 Android UI layouts hierarchy [10] 11

Figure 2.6 Android view recycling 12

Figure 2.7 Android Asynctask methods calling order [11] 14

Figure 3.1 Robot chassis 15

Figure 3.2 Motors control unit circuit diagram 16

Figure 3.3 Feedback unit circuit diagram 18

Figure 3.4 Motor speed encoder 19

Figure 3.5 Overall circuit diagram 19

Figure 4.1 Network Topology 20

Figure 4.2 DDNS service 22

Figure 4.3 Android Application flowchart 23

Figure 4.4 Android Use Case Diagram 24

Figure 4.5 Login activity 24

Figure 4.6 Products Activity 26

Figure 4.7 OrderActivity 27

Figure 4.8 MyOrdersActivity 28

Figure 4.9 Java Program Use Case Diagram 29

Figure 4.10 background package UML diagram 30

Figure 4.11 robot package UML diagram 31

Figure 4.12 shared package UML diagram 32

FIGURE 4.13 ROBOTCONTROLLER FLOWCHART 34

FIGURE 4.14 DATABASE UML DIAGRAM 37

FIGURE 4.15 ORDERS TAB 38

FIGURE 4.16 ORDER DETAILS WINDOW 39

FIGURE 4.17 PRODUCTS TAB 40

FIGURE 4.18 ADD NEW PRODUCT WINDOW (LEFT), SELECTION OF PRODUCT POSITION (RIGHT) 40

FIGURE 4.19 CUSTOMERS TAB 41

FIGURE 4.20 NEW CUSTOMER WINDOW 42

FIGURE 4.21 MAP REPRESENTATION (D = 20) 42

FIGURE 4.22 A* VS DIJKSTRA'S ALGORITHM 44

FIGURE 4.23 ROBOT POSITION 50

FIGURE 4.24 ROBOT MOTORS CONTROL ALGORITHM FLOWCHART 53

FIGURE 4.25 PICTURE OF THE ROBOT 54

List of abbreviations and acronyms

ADC	Analog to Digital Converter
CSS	Cascading Style Sheet
DAC	Digital to Analog Converter
DDNS	Dynamic Domain Name Server
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Server
GUI	Graphical user interface
IC	Integrated Circuit
IP	Internet Protocol
ISP	Internet Service Provider
JDBC	Java DataBase Connectivity
LAN	Local Area Network
MAC	Media Access Control
OS	Operating System
PLC	Programmable Logic Controller
PWM	Pulse Width Modulation
SQL	Structured Query Language
TCP	Transmission Control Protocol
UI	User Interface
UML	Unified Modeling Language
USB	Universal Serial Bus
XML	Extensible Markup Language

Chapter 1

Introduction

1.1. Introduction

The use of technology in human's everyday life to save time and efforts on daily activities is becoming an important challenge. The fast evolution of technology has changed the way that people used to do basic things, like shopping. Robots took an important place in this area, they are taking the lead in many fields such as manufacturing, utilities, transportation and exploration, by replacing humans in complex and dangerous tasks, but also in time consuming jobs.

Numerous innovative applications of robots have been used during the last years by some of the biggest tech companies in the world. As an example, the online retail giant Amazon uses robots to manage its warehouses. Traditionally, goods in a warehouse are moved around a distribution centre using a conveyor system or by human operated machines. In Kiva's approach (Amazon robotics system), items are stored on portable storage units. When an order is entered into the Kiva database system, the software locates the closest automated guided vehicle (bot) to the item and directs it to retrieve it. The mobile robots navigate around the warehouse by following a series of computerized barcode stickers on the floor. Each drive unit has a sensor that prevents it from colliding with others. When the drive unit reaches the target location, it slides underneath the pod and lifts it off the ground through a corkscrew action. The robot then carries the pod to the specified human operator to pick the items [1].

1.2. Motivation

Shopping is a daily task that consumes time and efforts for almost everyone, especially when it comes to supermarkets where the customer has to go through endless shelves to collect the products one by one, and then wait on a queue to pay and finally carry the products to the car.

This task could be much simpler if technology was more involved in it. The aim of this project is to design and implement a system to allow the customers to order their products remotely using an Android application on their smartphones. The orders are then received and processed by a computer, which will send instructions wirelessly to a robot to collect the products and prepare them for the customer. The customer can follow the state of his order in real time from his

smartphone, so that he can pick it up when ready without even leaving the car. This system would save valuable time and efforts for any random person.

1.3. Related works

Some similar online shopping systems have been implemented before, different approaches and technologies have been used in these projects. As in Zhao G. and Zhou Z.'s work, the system is based on MVC architecture and adopts ASP.NET, Dreamweaver, SQL Server 2005, ADO.NET Entity Framework and other related technologies. Their system includes some foreground functions such as user registration and login, checking and buying commodities, the shopping cart, the personal order management, the customer complaint and personal information management, etc. The background functions include the administrator login, the commodities category management, the commodities management, the order management, etc [2]. However, the collection of the products is done manually without the use of robots. Another similar work was done by "M. Z. A Rashid, T. A. Izzuddin, N. Abas, N. Hasim, F. A. Azis and M. S. M. Aras", it concerned the control of an automatic food drive-through system using Programmable Logic Controller (PLC). In their approach, no human operator is needed, it is fully operated by machine, like order and payment machine and conveyor that deliver the food, while human are needed and involved only to prepare the food [3].

1.4. Overall system description

The whole system of this project consists of three entities which are : An android application that is used by the user to select the products he wants to order, a java program that receives the orders and processes them to send instructions to the robot, but also manages the system's database and monitors orders and robot state, and finally a mobile robot controlled by an ESP32 microcontroller that collects the products. The block diagram of **Figure 1.1** describes the system :

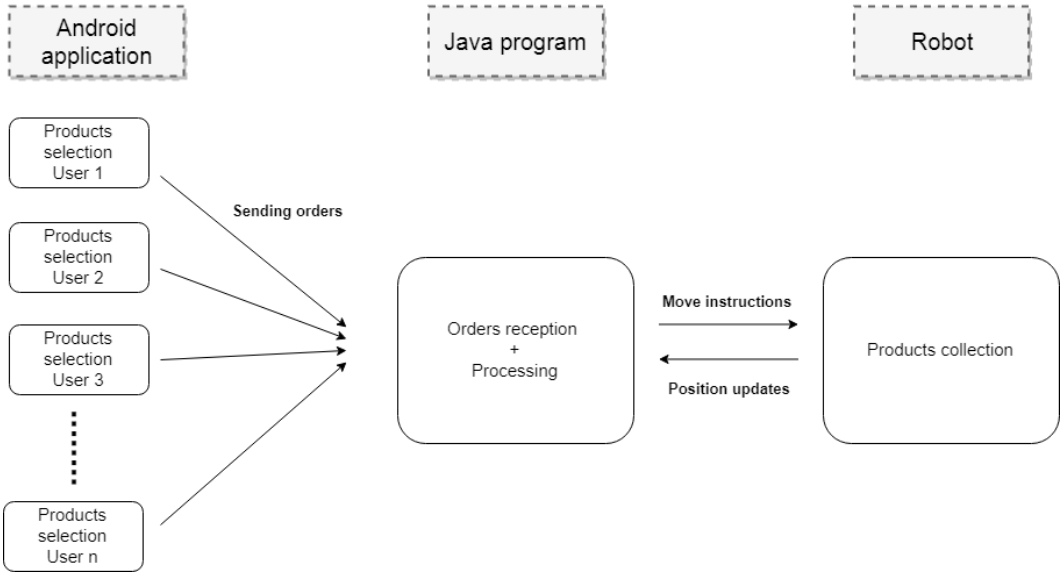

Figure 1.1 Overall system block diagram

1.5. Organization of the report :

This report is organized into five chapters. In addition to this introduction, Chapter 2 covers the theoretical background about some of the hardware and software tools used in this project. Chapter 3 deals with the system's hardware design. In chapter 4 the software design will be discussed. And finally Chapter 5 will be a conclusion about this project with comments and suggestions for further improvements.

Chapter 2

Theoretical background

2.1. Introduction

This chapter introduces the theoretical background needed to design the hardware and software of our system, listing the main components used, their description, and principle of operation.

2.2. The ESP32 Microcontroller

2.2.1. ESP32 chip

The ESP32 is a single 2.4 GHz Wi-Fi-and-Bluetooth combo chip designed with the TSMC ultra-low-power 40 nm technology. It is designed to achieve the best power and RF performance, showing robustness, versatility and reliability in a wide variety of applications and power scenarios. Features of the ESP32 include the following [4] :

- Processors:
  - ✓ CPU: Xtensa dual-core (or single-core) 32-bit LX6 microprocessor, operating at 160 or 240 MHz and performing at up to 600 DMIPS
  - ✓ Ultra low power (ULP) co-processor
- Memory: 520 KiB SRAM
- Wireless connectivity:
  - ✓ Wi-Fi: 802.11 b/g/n
  - ✓ Bluetooth: v4.2 BR/EDR and BLE
- Peripheral interfaces:
  - ✓ 12-bit SAR ADC up to 18 channels
  - ✓ 2 × 8-bit DACs
  - ✓ 10 × touch sensors (capacitive sensing GPIOs)
  - ✓ 4 × SPI
  - ✓ 2 × I²S interfaces
  - ✓ 2 × I²C interfaces
  - ✓ 3 × UART
  - ✓ SD/SDIO/CE-ATA/MMC/eMMC host controller
  - ✓ SDIO/SPI slave controller
  - ✓ Ethernet MAC interface with dedicated DMA and IEEE 1588 Precision Time Protocol support
  - ✓ CAN bus 2.0
  - ✓ Infrared remote controller (TX/RX, up to 8 channels)
  - ✓ Motor PWM
  - ✓ LED PWM (up to 16 channels)
  - ✓ Hall effect sensor
  - ✓ Ultra low power analog pre-amplifier
- Security:

- ✓ IEEE 802.11 standard security features all supported, including WFA, WPA/WPA2 and WAPI
- ✓ Secure boot
- ✓ Flash encryption
- ✓ 1024-bit OTP, up to 768-bit for customers
- ✓ Cryptographic hardware acceleration: AES, SHA-2, RSA, elliptic curve cryptography (ECC), random number generator (RNG)

- Power management:
  - ✓ Internal low-dropout regulator
  - ✓ Individual power domain for RTC
  - ✓ 5µA deep sleep current
  - ✓ Wake up from GPIO interrupt, timer, ADC measurements, capacitive touch sensor interrupt

Figure 2.1 shows the functional block diagram of the ESP32 :

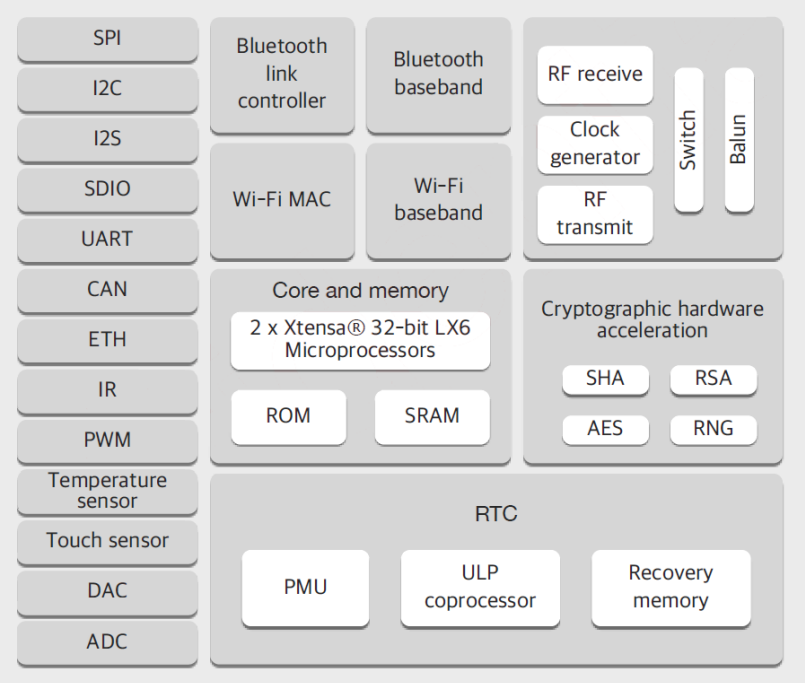

Figure 2.1 ESP32 functional block diagram [4]

All these peripherals make this chip ideal for a variety of applications, including internet of things, smart agriculture, digital signal processing, home automation and others.

2.2.2. ESP32 development board (DOIT DevKit v1)

Different ESP32 development boards have been created to make the prototyping process and interfacing easier. Some boards (ex: ESP-EYE) include a camera that can be interfaced directly with the ESP32 chip, which makes it suitable for image recognition, or IP camera applications. Some others like the ESP32-LCDKit are intended for human machine interface applications, they integrate an SD-Card, DAC-Audio and can be connected to external displays. For this application, the board that have been used is the DOIT ESP32 DevKit v1 as shown in **figure 2.2**.

Figure 2.2 DOIT Esp32 DevKit v1

The DevKit v1 comes with a serial-to-usb chip that allows programming and opening of the UART of the ESP32. A USB Micro B connector is used to connect the board to the computer for uploading the program and for serial communication. The board can be powered through that port, or directly via the VIN pin. A power regulator is integrated on the board and accepts an external power supply between 7 and 12 volts. The operating voltage of the board is 3.3 volts. It also include two push buttons, one for reset and one for boot mode to upload the sketch. **Figure 2.3** illustrates the pin mapping of the board.

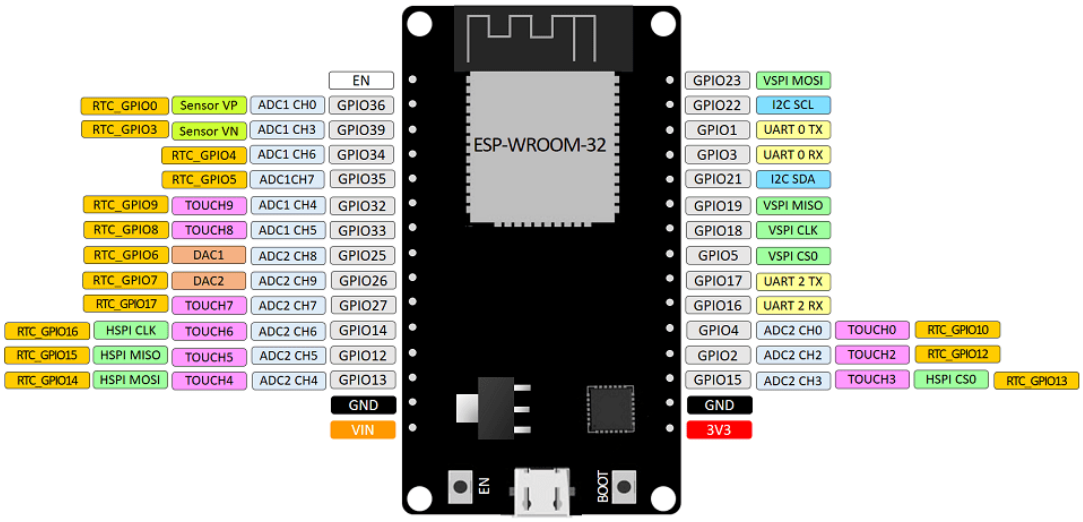

Figure 2.3 DOIT ESP32 DEVKit v1 pin mapping

2.2.3. Software development environment

Several development environments, frameworks, and libraries can be used to program the ESP32, making it accessible for a large community of developers. A non exhaustive list includes :

- ESP-IDF (Official IOT platform by Espresssif)
- Arduino
- MicroPython
- Lua
- Free RTOS

The Arduino platform is the one that have been used for this project, since it is the most familiar one, and it uses C/C++ as a programming language.

2.3. Android application

2.3.1. Android OS

In the last 20 years, smartphones and tablets took an important place in human's everyday life. They are more and more used as life companions, or personal assistants, to facilitate daily actions and save time and efforts to accomplish them. As smartphones and tablets are advancing in technology, their operating systems are becoming more important, they have a crucial role in taking the most out of the Smartphone's hardware performances, as well as minimizing the battery power consumption.

The Android mobile operating system is the most popular OS for mobiles, it is the best selling worldwide on smartphones since 2011 and on tablets since 2013. Its app store called Google play counts more than 2.6 millions apps [5].

Android is developed by Google. It is based on a modified version of the Linux kernel and other open source software, and is designed primarily for touchscreen mobile devices such as smartphones and tablets. In addition, Google has developed Android TV for televisions, Android Auto for cars, and Wear OS for wrist watches, each with a specialized user interface. Variants of Android are also used on game consoles, digital cameras, PCs and other electronics.

2.3.2. Android studio

Android studio is the official integrated development environment for Android OS, based on InteliJ IDEA [6]. It is designed specifically for android applications development, and include a set of tools and utilities to build, test, and debug android applications. Two language are used in the development of android applications : java and xml. Java is used to program the logic and behavior of the application whereas xml is used mainly for UI design and in some specific files. Since October 2017, Kotlin programming language is also supported by android studio and can be used instead of Java.

Each project in Android Studio contains one or more modules with source code files and resource files. A module is a collection of source files and build settings that allow you to divide your project into discrete units of functionality. Your project can have one or many modules and one module may use another module as a dependency. Within each Android app module, files are shown in the following groups [7]:

- Manifest : Contains the **AndroidManifest.xml** file, which is a file that every android app must have, it describes essential information about the app to the Android build tools, the Android operating system, and Google Play.
- java : Contains the Java source code files, separated by package names
- res : Contains all non-code resources, such as XML layouts, UI strings, and bitmap images, divided into corresponding sub-directories.
- build.gradle : contains build configuration files.

2.3.3. Activities

The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model. Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.

The mobile-app experience differs from its desktop counterpart in that a user's interaction with the app doesn't always begin in the same place. Instead, the user journey often begins non-deterministically. For instance, if you open an email app from your home screen, you might see a list of emails. By contrast, if you are using a social media app that then launches your email app, you might go directly to the email app's screen for composing an email.

An activity provides the window in which the app draws its UI. This window typically fills the screen, but may be smaller than the screen and float on top of other windows. Generally, one activity implements one screen in an app. For instance, one of an app’s activities may implement a Preferences screen, while another activity implements a Select Photo screen. Most apps contain multiple screens, which means they comprise multiple activities. Typically, one activity in an app is specified as the main activity, which is the first screen to appear when the user launches the app. Each activity can then start another activity in order to perform different actions [8].

2.3.4. Activity lifecycle

Over the course of its lifetime, an activity goes through a number of states, for example, users of an app might receive a call, respond to a notification, or switch to another task, and they should be able to continue using the app seamlessly after such an event. The Activity class provides a number of callbacks that allow the activity to know that a state has changed (that the system is creating, stopping, or resuming an activity, or destroying the process in which the activity resides). For each state a call back method exists and is called by the android system when the application enters the corresponding state, therefore the developer should override these methods in order to respond correctly to such events and handle transitions properly. **Figure 2.4** illustrates the lifecycle of an android activity.

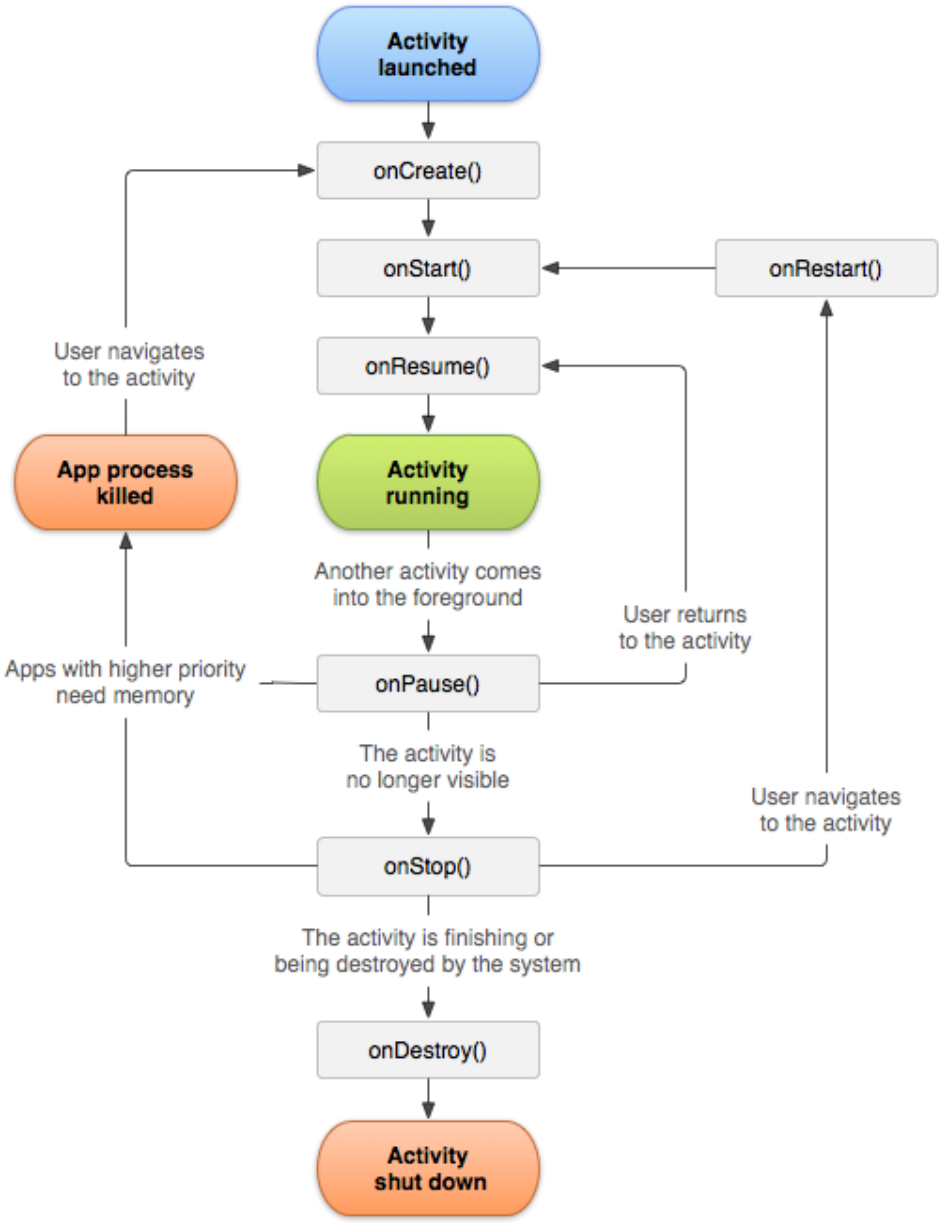

Figure 2.4 A simplified illustration of the activity lifecycle [9].

- onCreate() : This is the method that is fired when the system creates the activity for the first time, all the essential components of the activity should be initialized in this method.
- onStart() : As onCreate() exits, the activity enters the Started state, and the activity becomes visible to the user. This callback contains what amounts to the activity's final preparations for coming to the foreground and becoming interactive.
- onResume() : The system invokes this callback just before the activity starts interacting with the user. At this point, the activity is at the top of the activity stack, and captures all user input.

- onPause() : The system calls onPause() when the activity loses focus and enters a Paused state. This state occurs when, for example, the user taps the Back or recent button.
- onStop() : The system calls onStop() when the activity is no longer visible to the user. This may happen because the activity is being destroyed, a new activity is starting, or an existing activity is entering a Resumed state and is covering the stopped activity. In all of these cases, the stopped activity is no longer visible at all.
- onRestart() : The system invokes this callback when an activity in the Stopped state is about to restart.
- onDestroy() : This callback is the final one that the activity receives. onDestroy() is usually implemented to ensure that all of an activity's resources are released when the activity, or the process containing it, is destroyed.

2.3.5. UI Design

The user interface for an Android app is built using a hierarchy of *layouts* (ViewGroup objects) and *widgets* (View objects). Layouts are containers that control how their child views are positioned on the screen. Widgets are UI components such as buttons and text boxes. **Figure 2.5** shows how ViewGroup objects form branches in the layout and contain View objects

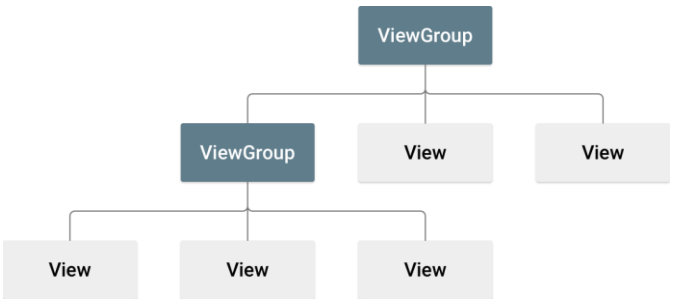

Figure 2.5 Android UI layouts hierarchy [10]

Android provides an XML vocabulary for ViewGroup and View classes, so most of the UI is defined in XML files. Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once the root element is defined, additional layout objects or widgets can be added as child elements to gradually build a View hierarchy that defines the layout. The most common ViewGroup objects are :

- LinearLayout : A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen.
- RelativeLayout : Specifies the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).
- ConstraintLayout : Used to create large and complex layouts with a flat view hierarchy (no nested view groups).

2.3.6. RecyclerViews

RecyclerView objects are widgets that are used when an application needs to display a scrolling list of elements based on large data sets (or data that frequently changes).

When working with large datasets, it is not possible to create a view object for each item to be displayed on the list, because view creation is an expensive task to be performed, and memory resources are limited. In Android, the concept of View recycling is used to overcome this problem. The idea is that, instead of creating views every time user scrolls, the views are created once, and recycled (reused) as needed. Initially, a number of views needed to fit the screen size are created, at that moment a part of the dataset is displayed only, then the user scrolls the list to see other elements. When the scroll action is performed, the first item on the list is hidden and the new item appears. Since the first item is no more visible to the user, the view object used to display its data can be reused to display the next element on the list if a scroll action is performed again. **Figure 2.6** illustrates the process.

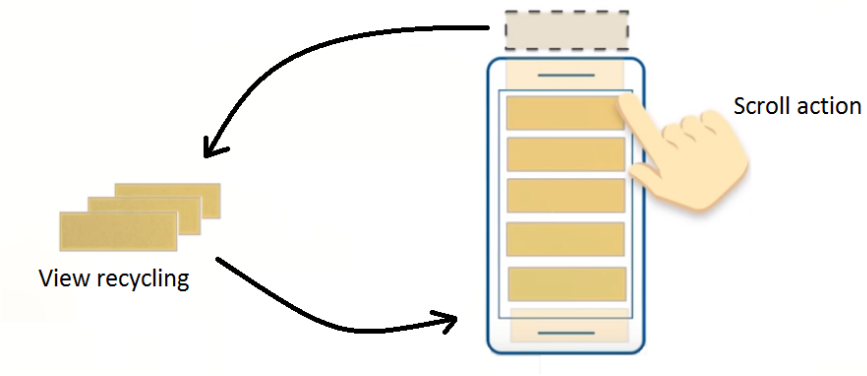

Figure 2.6 Android view recycling

An Adapter object is used as a bridge between the dataset and the recycler view (visual representation of the data), its role is to bind each data set item to a view and to manage the recycling operation (chose when to create a new view and when to recycle).

2.3.7. AsynchronousTasks

When an Android app starts, it creates the main thread, which is often called the User Interface (UI) thread. The UI thread dispatches events to the appropriate user interface (UI) widgets. The UI thread is where the app interacts with components from the Android UI toolkit. Android's thread model has two rules [11]:

- Do not block the UI thread
- Do UI work only on the UI thread

The UI thread needs to give its attention to drawing the UI and keeping the app responsive to user input. If everything happened on the UI thread, long operations such as network access or database queries could block the whole UI. From the user's perspective, the app would appear to hang. The android framework offers an effective solution which is the **AsyncTask** class, it is used for running tasks on a background thread and publish results on the UI thread without needing to directly manipulate threads or handlers. When AsyncTask is executed, it goes through four steps :

- onPreExecute( ) : Invoked on the UI before the task is executed, often used to show a progress bar before starting.
- doInBackground(Params ...): is invoked on the background thread immediately after onPreExecute() finishes. This step performs a background computation, returns a result, and passes the result to onPostExecute().
- onProgressUpdate(Progress ...) : runs on the UI thread if publishProgress( ) is called in doInBackground( ) , can be used to update a progress bar.
- onPostExecute(Result) : runs on the UI thread after doInBackground is finished, to update the UI with the results.

Figure 2.7 shows the calling order of these methods :

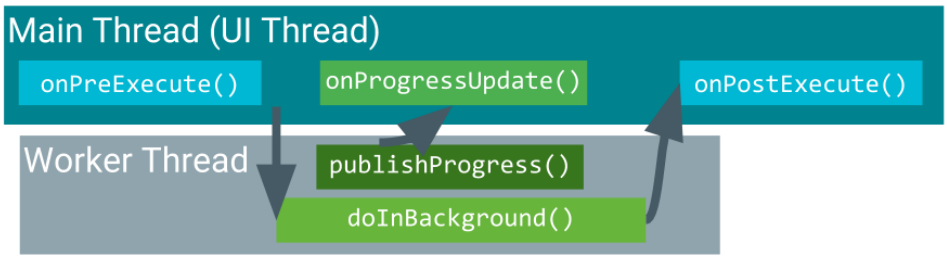

Figure 2.7 Android Asynctask methods calling order [11]

2.4. JavaFx

JavaFX is a software platform for creating and delivering **desktop** applications, as well as Rich Internet Applications (RIAs) that can run across a wide variety of devices. It enables developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms.

Written as a Java API, JavaFX application code can reference APIs from any Java library. For example, JavaFX applications can use Java API libraries to access native system capabilities and connect to server-based middleware applications.

The look and feel of JavaFX applications can be customized. Cascading Style Sheets (CSS) separate appearance and style from implementation so that developers can concentrate on coding. Graphic designers can easily customize the appearance and style of the application through the CSS.

It is possible to separate the development of user interface (UI) and the back-end logic, by developing the presentation aspects of the UI in the FXML scripting language and using Java code for the application logic. JavaFX Scene Builder can also be used to design the UI graphically and generate the FXML code automatically without prior knowledge of the FXML language.

Chapter 3

Hardware System Design

3.1. Introduction

This chapter discusses the hardware system design. The hardware part of this project consists of a mobile robot controlled by a microcontroller that receives instructions from a server through a WIFI wireless network. The instructions received are in the form of coordinates of the points where the robot has to go to collect the products. In this project we focus only on the robot displacement, the action of collecting the products will not be discussed due to the mechanical complexity and lack of equipments.

3.2. Robot chassis

The robot that have been used for this project is a differential drive robot. Differential drive robots are mobile robots whose movements are based on two separately driven wheels, one on each side of the robot. A third free turning wheel is usually added to balance the robot. **Figure 3.1** illustrates the top and bottom view of the robot.

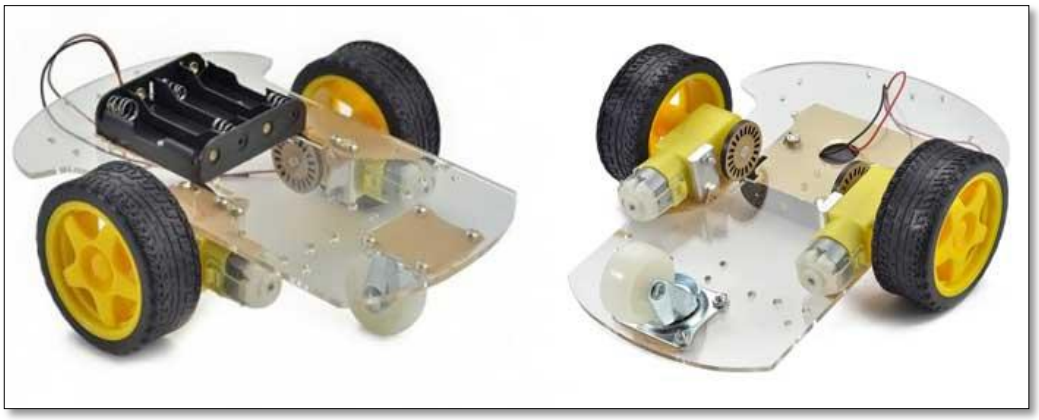

Figure 3.1 Robot chassis

The advantage of using differential drive robots compared to other robots having one motor for the forward and backward motion, and one for the direction, is that they can rotate around the central axis of the wheels by turning the two wheels in opposite directions, and thus allowing sharp 90° rotations, which is of course not possible using the other category of robots. This feature is essential in this project, this is why a differential drive robot have been chosen. In the counterpart,

controlling such a robot is more difficult, due to problems on the synchronization of the two wheels, this problem will be discussed later on this chapter.

3.3. Electronic circuit

3.3.1. Motors control unit

The two motors of the robot are controlled by a L293D dual H-bridge integrated circuit. The IC receives its control signals from the ESP32 microcontroller. The L293D energizes the two motors from an external power source of 7.4 Volts, and controls their directions at the same time. **Figure 3.2** shows the circuit diagram of the motors control unit.

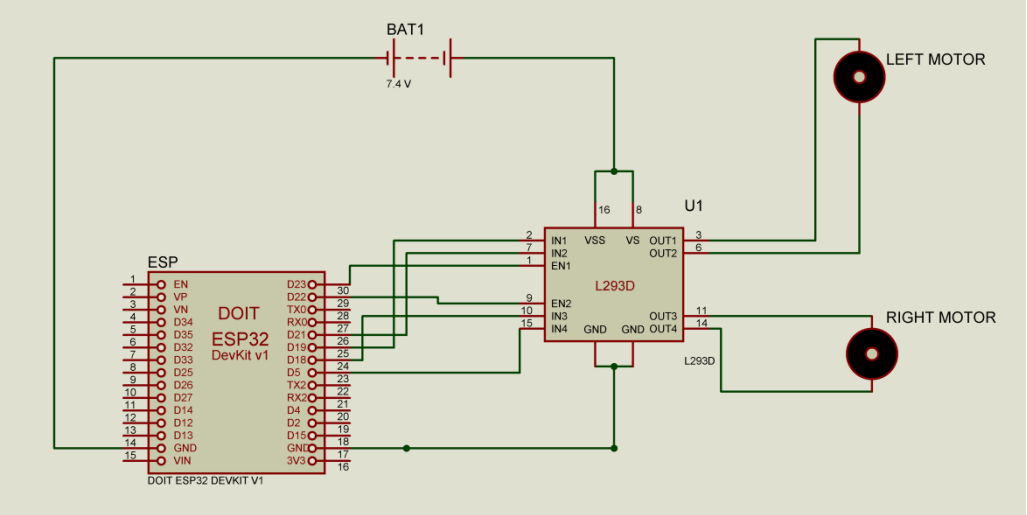

Figure 3.2 Motors control unit circuit diagram

The four control inputs (IN 1, IN 2, IN 3, IN 4) of the L293D chip are connected to the digital pins (D 19, D 21, D 18, D 5) of the microcontroller to control the direction of each motor. The Enable pins EN1 and EN2 are driven by digital pins D23 and D22, which are used to generate a PWM signal to control the speed of each motor individually.

As mentioned previously, the control of the direction of a differential drive robot is done by controlling the direction of each motor individually, and each combination results in a different move. **Table 3.1** resumes the different combinations of the left motor direction and the right motor direction, and the resulting robot move.

Table 3-1 Robot move combinations

Comb	IN1	IN2	IN3	IN4	Left Motor	Right motor	Robot move
1	0	0	0	0	Brake	Brake	Brake
2	0	0	0	1	Brake	Backward	Turn Right
3	0	0	1	0	Brake	Forward	Turn Left
4	0	0	1	1	Brake	Brake	Brake
5	0	1	0	0	Backward	Brake	Turn Left
6	0	1	0	1	Backward	Backward	Backward
7	0	1	1	0	Backward	Forward	Turn Left
8	0	1	1	1	Backward	Brake	Turn Left
9	1	0	0	0	Forward	Brake	Turn Right
10	1	0	0	1	Forward	Backward	Turn Right
11	1	0	1	0	Forward	Forward	Forward
12	1	0	1	1	Forward	Brake	Turn Right
13	1	1	0	0	Brake	Brake	Brake
14	1	1	0	1	Brake	Backward	Turn Right
15	1	1	1	0	Brake	Forward	Turn Left
16	1	1	1	1	Brake	Brake	Brake

As we can see, many different combinations result in the same move. In our case, we will be using three moves which are : Forward move, turn left, and turn right. Since we want the robot to rotate around the central axis of the wheels, we will use combinations number 7 and 10 to turn left and right. Braking will be done using combination 1.

3.3.2. Feedback unit

As mentioned earlier, the drawback of using differential drive robots is that it is difficult to have both motors rotating at the exact same speed, even if the two motors are powered using the same source, and this is due to the fact that the low cost

motors used do not guarantee to respect their technical specifications, and thus, are not exactly identical. As a result, the robot will not be able to move in a straight line or to rotate by an exact angle. This is why a feedback unit is needed in order to adjust the speed of each motor in real time and to keep them synchronized. **Figure 3.3** illustrates the feedback unit circuit diagram.

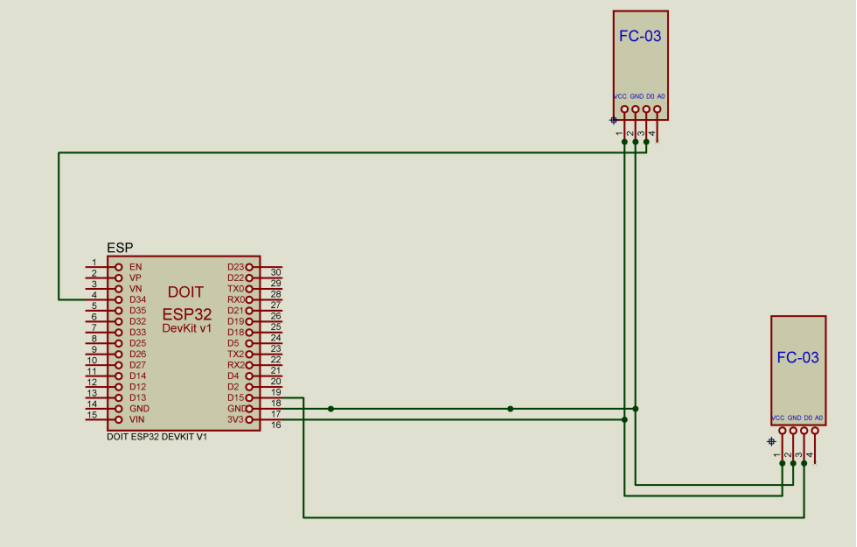

Figure 3.3 Feedback unit circuit diagram

The feedback unit consists of two FC-03 infrared speed sensors, one for each motor. The sensor contains an opto-interrupter that detects when an object passes between the IR emitter and the IR receptor, as well as an L393 comparator chip to get a High output when no object is detected, and a Low when an object is detected. A 100nF capacitor is added between the D0 output of each sensor and the ground to eliminate undesirable rebounds on rising and falling edges. The sensors are used with two speed encoder disks that are attached to each wheel of the robot, the disks contain equally spaced holes so that when the wheel is rotating the holes cross the IR light of the sensor and generate a pulse that interrupts the microcontroller, as shown in **Figure 3.4**.

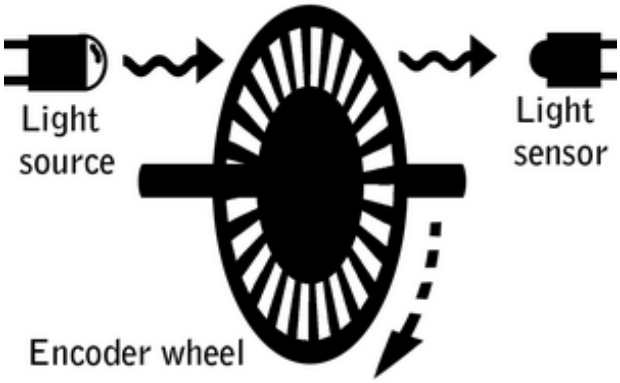

Figure 3.4 Motor speed encoder

By counting the number of times an interrupt is generated for each motor, we can calculate the speeds and adjust the PWM waves accordingly to make the actual speeds as close as possible to each other. In addition to that, it can be used to calculate the exact distance traversed by the robot, the algorithms used for that will be discussed in the next chapter.

3.3.3. Overall circuit diagram

The overall circuit diagram is the combination of both the motors control unit, and the feedback unit. The ESP32 microcontroller board is the common element between the two units as shown in **Figure 3.5.**

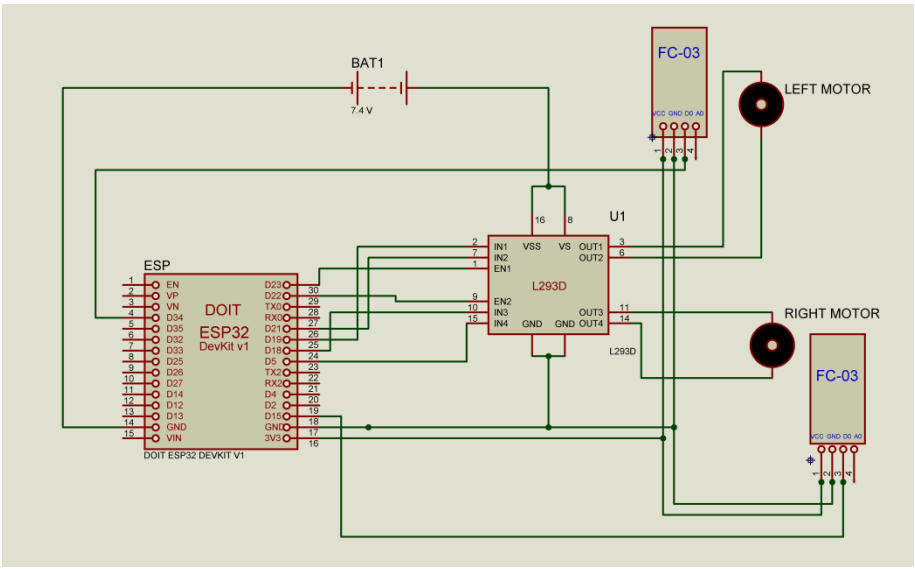

Figure 3.5 Overall circuit diagram

Chapter 4

Software System Design

4.1. Introduction

In this chapter, we are going to discuss the software design and the network configuration for this project. The software consists of three parts which are : An Android application used by the clients, a Java program used by the administrator, and the robot software.

4.2. Network configuration

This section discusses the network topology and configuration for the project. The robot and the computer running the main server are connected to a router which is connected to internet as shown in **Figure 4.1.**

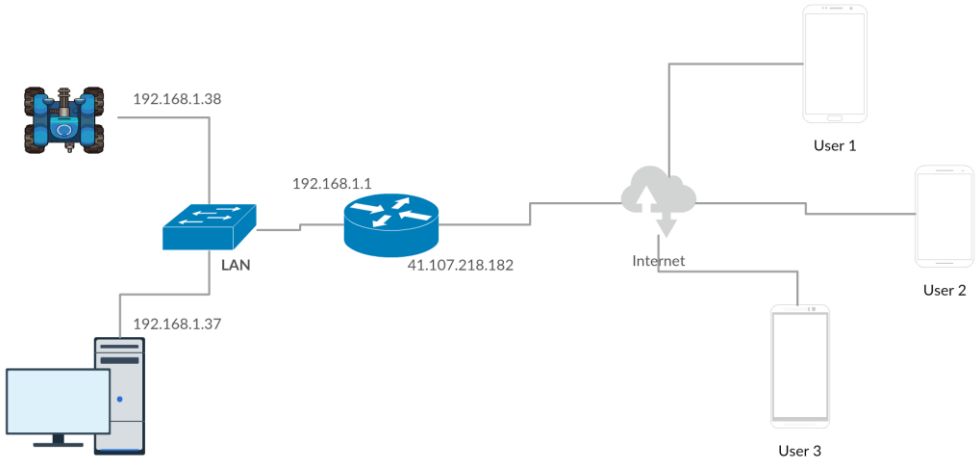

Figure 4.1 Network Topology

In order to receive the orders sent from any device connected to internet, the server needs to be reachable from outside the local area network (LAN). By default this is not possible due to the following problems : First, there is no way to send a packet to the server from outside the LAN network directly, since the router is not configured to forward the packets to that device. Second, the local IP addresses of the devices inside the LAN are assigned using dynamic host configuration protocol (DHCP), each time a new device is connected to the network the protocol picks up a random address from a pool of available IP addresses and assigns it to the device for a certain amount of time, and this does not ensure that the same device (Robot, PC) always get the same IP address. And finally, the public IP address of the router may

change since it is dynamically provided by the Internet Service Provider (ISP), and we have no control over it.

All these problems have been solved by setting the following configuration to the router.

4.2.1. Port forwarding

To enable our server to receive packets from outside the LAN, the router needs to be configured to forward the packets coming from outside the LAN to the computer which is running the server, and this is called "port forwarding". Basically, it means that the router will map the external ports starting from 5563 to 5566 on the public IP address to the corresponding local ports of the computer running the server, and hence, all the packets from internet received by the router having a destination port between 5563 and 5566 will be sent to the host which is running the server on the LAN (i.e. : 192.168.1.37).

4.2.2. Dynamic Host Configuration Protocol reservation (static DHCP)

For the second problem, we need to configure our router to always assign the same IP addresses to both the robot and the computer running the server. This can be configured in the DHCP settings of the router, where static entries can be defined. The router has a DHCP reservation list, and each list entry consists of a tuple (IP address, MAC address), where the user can reserve an IP address such that it can never be assigned to a host except the one having the corresponding MAC address.

4.2.3. Dynamic Domain Name Server (DDNS)

Since the public IP address is assigned by the ISP and may change at any time, a dynamic domain name server (DDNS) is used to overcome this problem. The principle of working of the DDNS is almost the same as the DNS, it associates each domain name to an IP address, such that when a client tries to access to a server using its domain name, it sends a request to the DNS to get its IP address. This is used because it is much easier for humans to work with domain names than with IP addresses. The only difference with DDNS is that the associated IP address is not fixed, which means that the server's IP address may change frequently, and hence,

the DDNS server needs to receive updates each time the address is changed. **Figure 4.2** illustrates the working principle of DDNS.

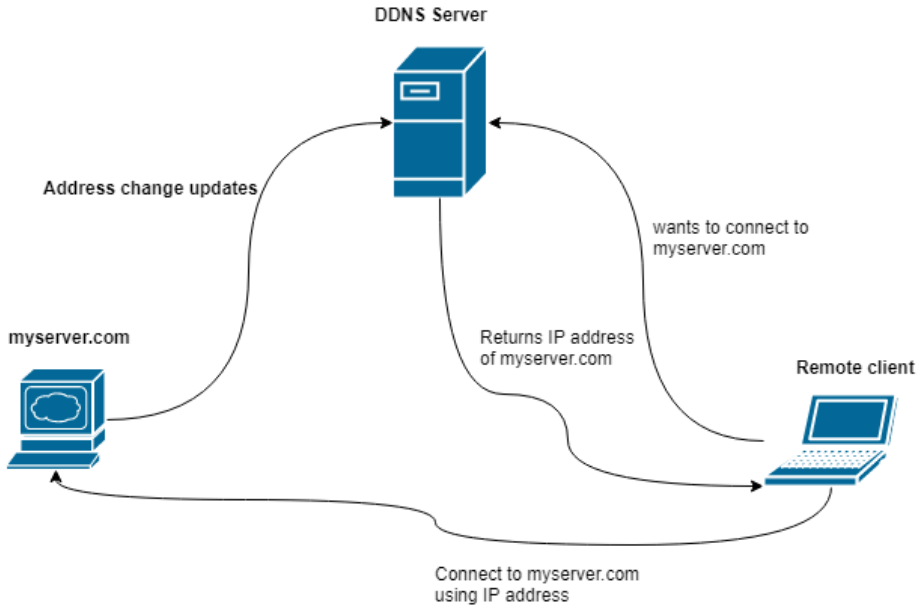

Figure 4.2 DDNS service

The DDNS service that have been used for this project is called NOIP. It is free and easy to set up, first an account is created on www.noip.com, then an available domain name is selected for the server, and then two options are possible : either install the software client on the computer running the server and the software will keep running on the background and send IP address updates periodically, or configure the service on the router directly, in this case the router will be sending the address updates to the DDNS server each time to associate them with the selected hostname. The latter option is the one that have been used.

4.3. Android Application

In this section we will discuss the android application that have been designed for this project. Flowchart of **Figure 4.3** describes the general operation of the application.

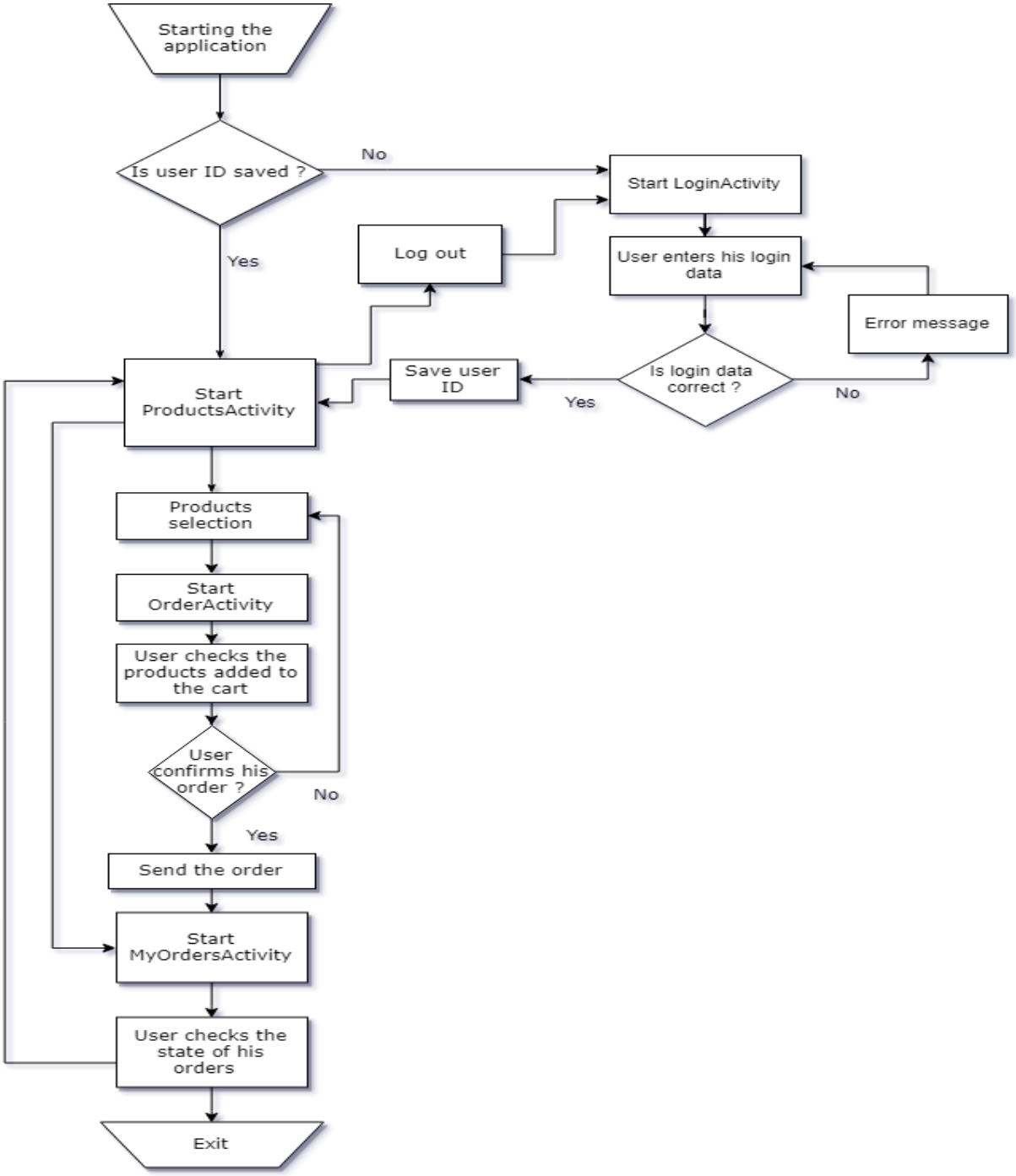

Figure 4.3 Android Application flowchart

The Android application is intended to be used by the customers of the supermarket, **Figure 4.4** shows the UML use case diagram that describes all the possible use scenarios and their relationships :

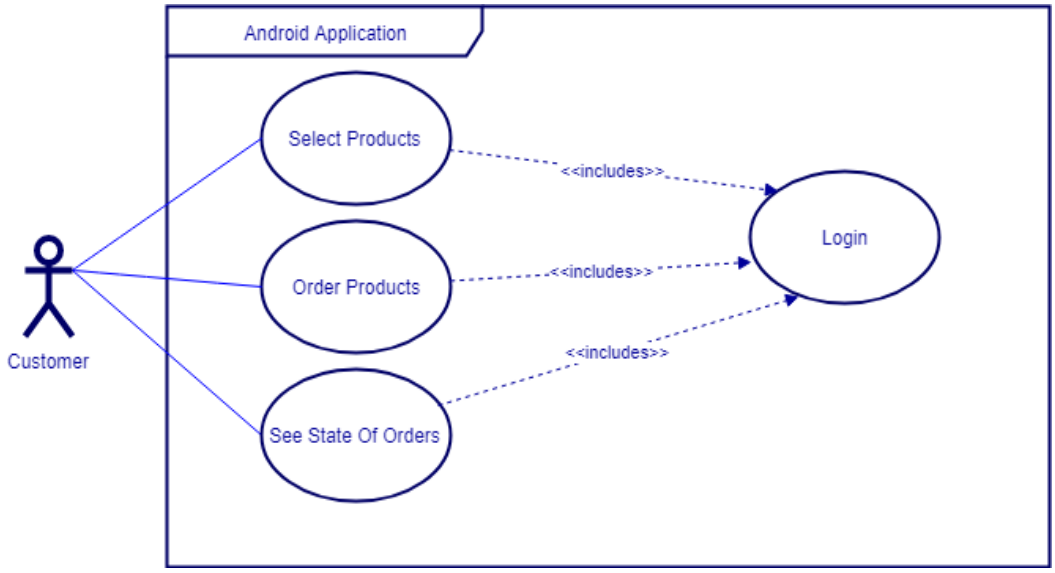

Figure 4.4 Android Use Case Diagram

The application has a total of 4 Activities that makes it simple and intuitive for the end user, each activity is designed to accomplish a certain task and has its own layout. The four activities are :

4.3.1. LoginActivity

This is the first activity that is launched when the application starts. The layout of this activity contains two EditText's (text fields), one for the phone number and the second for the password, it also contains a login button as shown in **Figure 4.5**.

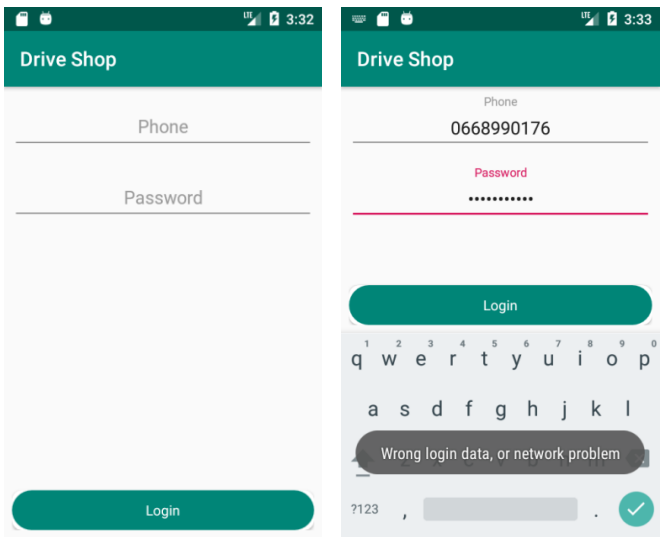

Figure 4.5 Login activity

The aim of this activity is to identify the user before going to the next steps. No registration is allowed via the application, it is done directly on the java desktop program for a better control of the customers database. Once the user enters his credentials, and presses the login button, an AsynchronousTask is started to perform the network operation in the background and keep the UI thread free. The AsynchronousTask creates a Socket object that establishes a TCP connection to the main server, then it sends the login data through an ObjectOutputStream and waits for an answer using an ObjectInputStream. The answer is in the form of a value of type long, two cases are possible : if the data sent corresponds to a user in the database, then the ID of the user is returned from the server and the ProductsActivity is started, otherwise a negative value is returned meaning that no matching user have been found and an error message is displayed as shown in **Figure 4.5**. When the user is successfully identified, the ID of the user is saved in the application data to be used for making orders, and so that the next time the user starts the application he doesn't need to login again and the ProductsActivity is started directly, until he decides to log out from his account.

4.3.1. ProductsActivity

In this activity, the layout contains a RecyclerView with the list of products, and a "My Cart" button on the toolbar which is disabled by default. An AsynchrounousTask is started in the onCreate method to fetch the data from the server, it establishes a TCP (Transmission Control Protocol) connection and uses an ObjectInputStream to get the list of products which are then stored in an ArrayList<Product> and passed to a custom Adapter object which is responsible for binding each object on the list with a recycler view item.

Each recycler view item has the following elements : three TextView's for product name, description, and price, an ImageView to display a small image of the product, and an "Add" Button used to add the product to the cart, once pressed a dialog window pops up and prompts the user to select the desired quantity (between 0 and the Maximum available for that product). **Figure 4.6** shows screenshots of the products activity.

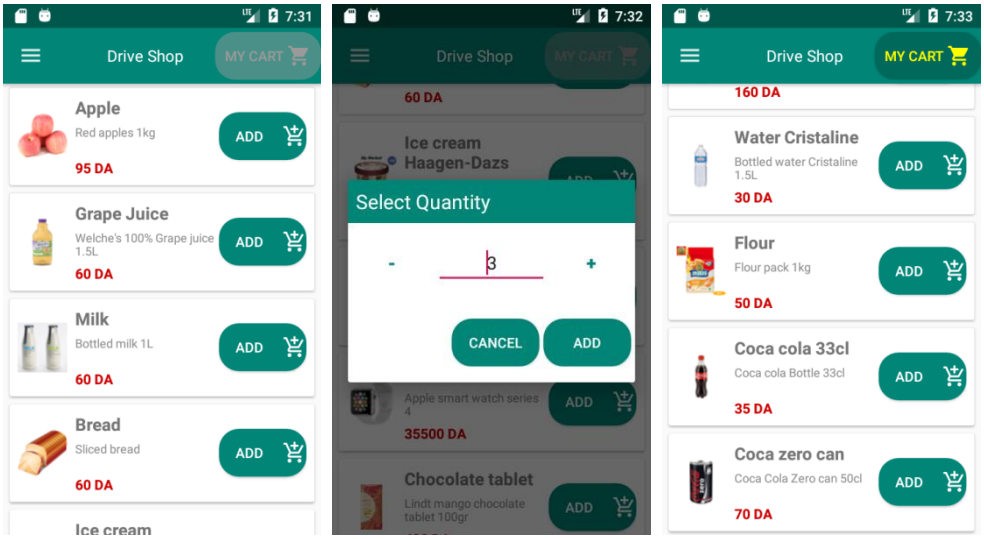

Figure 4.6 Products Activity

Once a product is added to the cart, the "My Cart" button is enabled allowing the user to check the content of his cart in the Order Activity. The user can refresh the list of products by swiping down the list when already at the top. A side menu is also available to navigate to the other activities, three options are possible : My cart, have the same effect as clicking the button on the toolbar, My orders : starts MyOrdersActivity, and Logout to delete the registered user ID and go back to LoginActivity.

4.3.2. OrderActivity

Once the user has selected his products and pressed the "My Cart" button, the cart object, which is of type ArrayList<SubOrder>, is passed in an intent using the putExtra method to the OrderActivity, and the activity is started.

The OrderActivity contains a RecyclerView that displays the list of products in the cart, each recycler view item has the following elements : four TextView's for product name, description, quantity, and sub-total price (which is calculated by unit price x quantity), an ImageView for the product image, and an ImageButton with a bin icon to remove the product form the cart. The total price, which is the sum of sub-totals of each item is displayed in another TextView at the bottom of the screen. Screenshots of the order activity are shown in **Figure 4.7**.

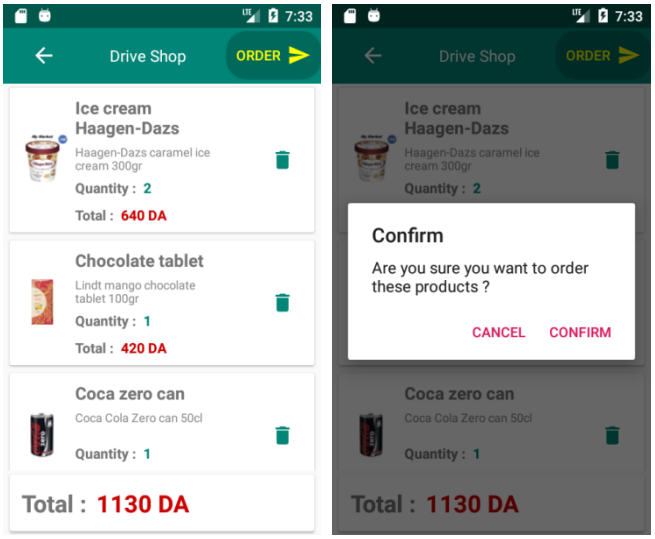

Figure 4.7 OrderActivity

To confirm his order, The user presses the "Order" button on the top, this will show a confirmation message before sending an instance of the Order class with the registered user ID through a TCP socket to the main server to process the order. A boolean value is then returned from the server to indicate whether the order has been accepted or not, this could happen in the case where many users added the same product to their carts and the product ran out of stock before one of them confirmed his order, in that situation an error message is displayed to the user stating that one of the products is no more available. Otherwise, the order is accepted and user is redirected to MyOrders Activity.

4.3.3. MyOrdersActivity

In MyOrdersActivity, a list of orders is displayed to the user in a RecyclerView. The same process is used as in the ProductsActivity, an AsynchronousTask is started to connect to the server through a TCP socket, then the user ID is sent to the server that will query the database for the corresponding orders. The server responds with the list of Order objects through an ObjectOutputStream, the list is then passed to the Adapter to bind them with the corresponding Views. Each list element contains two information : the order ID and the state of the order as shown in **Figure 4.8**.

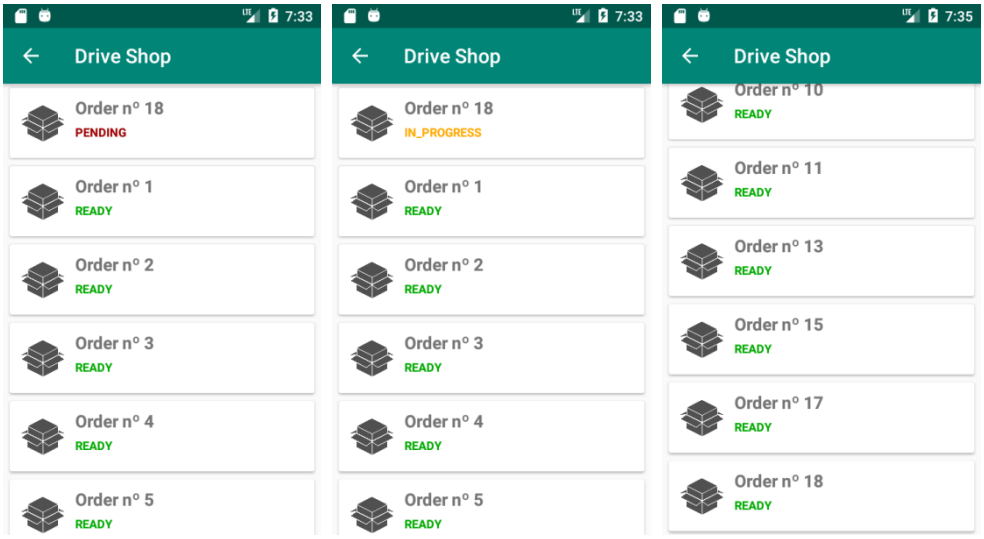

Figure 4.8 MyOrdersActivity

The orders are ordered in a way such that the ones in the "IN_PROGRESS " state are displayed first, followed by the ones in "PENDING" state, and finally the "READY" state. The user can refresh the list by swiping down to follow the state updates of his orders in real time and know when they are ready for pick up.

4.4. The java desktop program

This is the central part of the project, it consists of a java program with a set of background threads that act as a server to send/receive data from/to Android clients, control the robots, and manages the orders. It also contains a GUI that is used to manage the products and customers database, and also monitor the orders state and robot position in real time.

Figure 4.9 shows the UML use case diagram that describes the possible actions that can be done by the administrator.

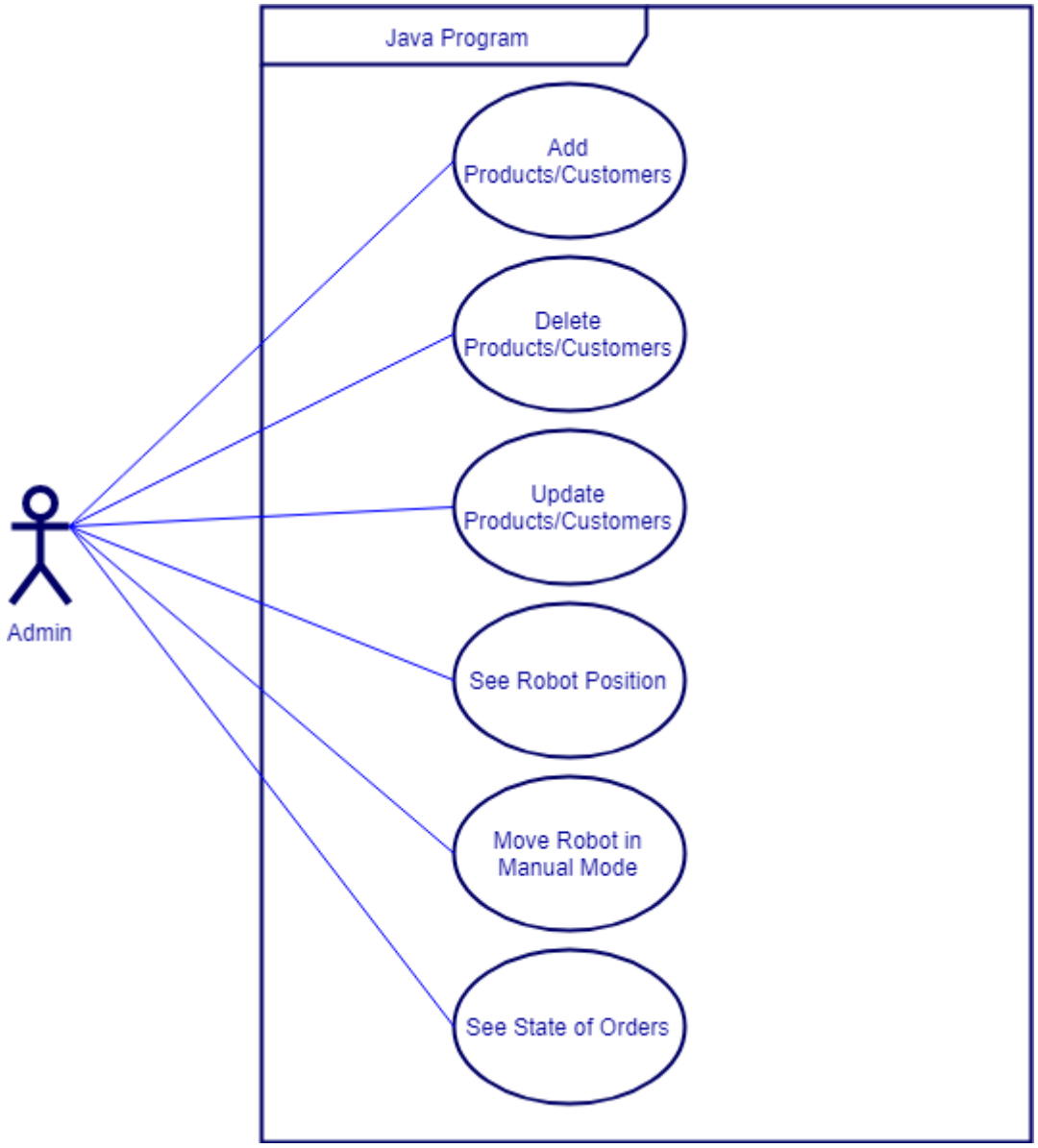

Figure 4.9 Java Program Use Case Diagram

4.4.1. Classes organization

The classes are divided into different packages for a better organization of the source code, the packages are organized in the following way :

- Background: contains the classes responsible of running the background networking tasks, the UML diagram in **Figure 4.10** shows the different classes in this package and their relations :

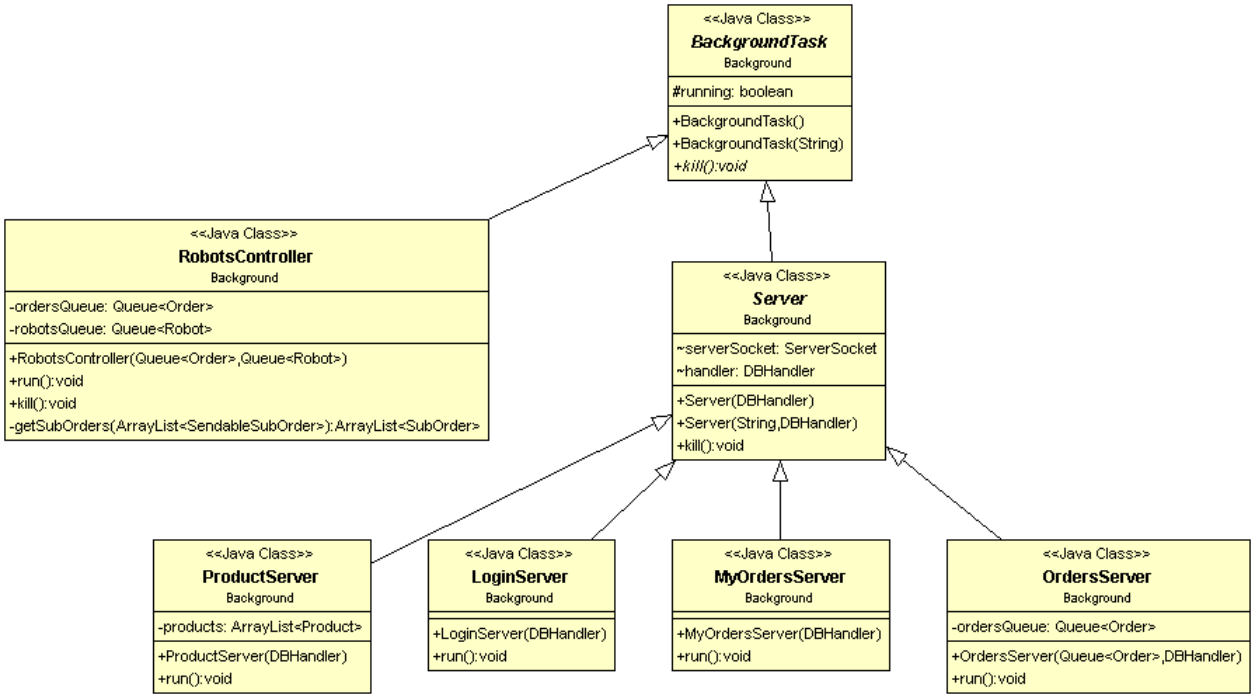

Figure 4.10 background package UML diagram

- database : contains two classes, **DBHandler** which is a class used to handle communication with the SQLite database, and **Queries** which is an abstract class containing constants for tables and attributes names organized into internal classes .
- GUI : contains the different classes used for the graphical user interface, it contains also sub-packages such as : **controller**, which contains the controller classes each fxml layout file, **layout**, contains the fxml files generated using scene builder, and **dialog** containing classes for different dialog windows used in the GUI (error dialog, info dialog..)
- Observer : contains two interfaces used to implement the observer design pattern used to update the robot position on the map when the robot object changes its position.
- robot : contains the Robot class used to communicate with the physical robot and a VirtualRobot class, which is a subclass of Robot that has an additional thread that simulates the physical robot behavior. **Figure 4.11** shows the UML diagram for the two classes :

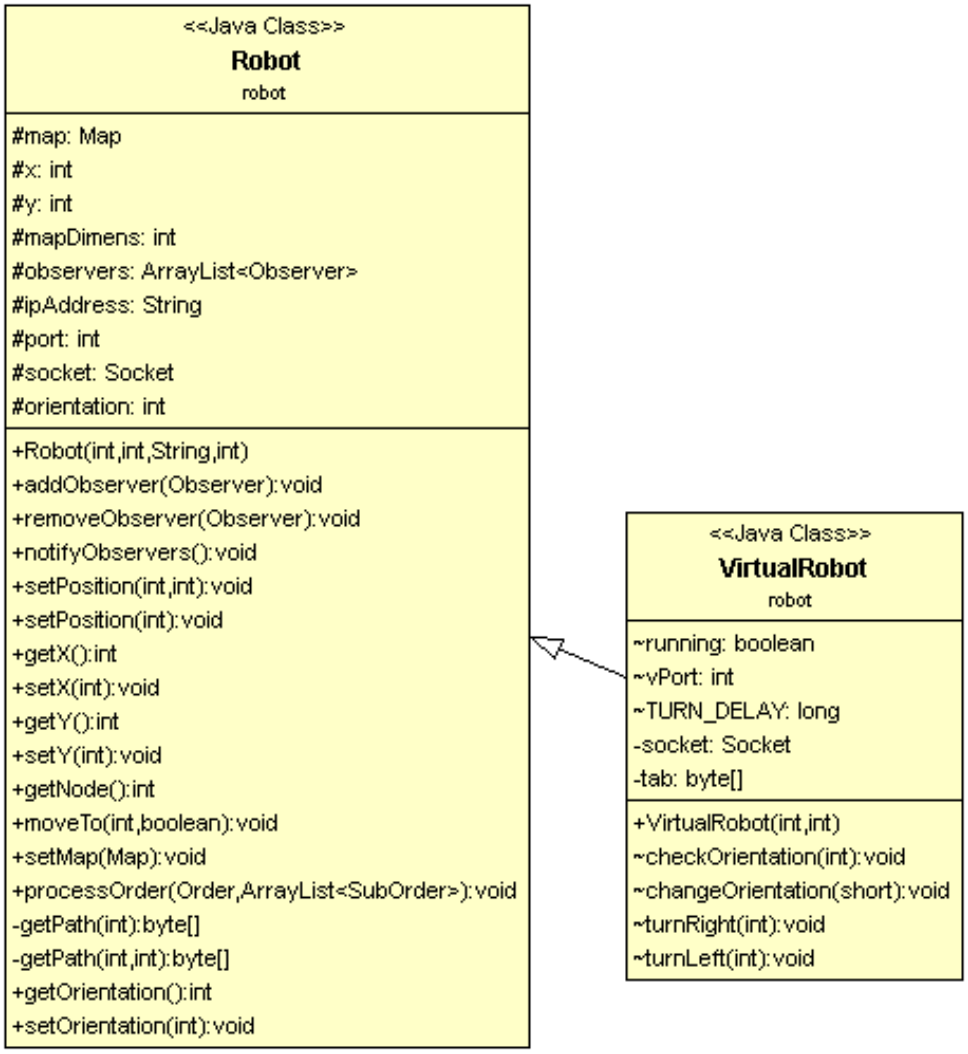

Figure 4.11 robot package UML diagram

- utils : contains some utility methods used in the project
- shared : contains the shared classes between the Java program and the Android application such as the Product class, and the Order class. **Figure 4.12** shows the UML diagram for these classes :

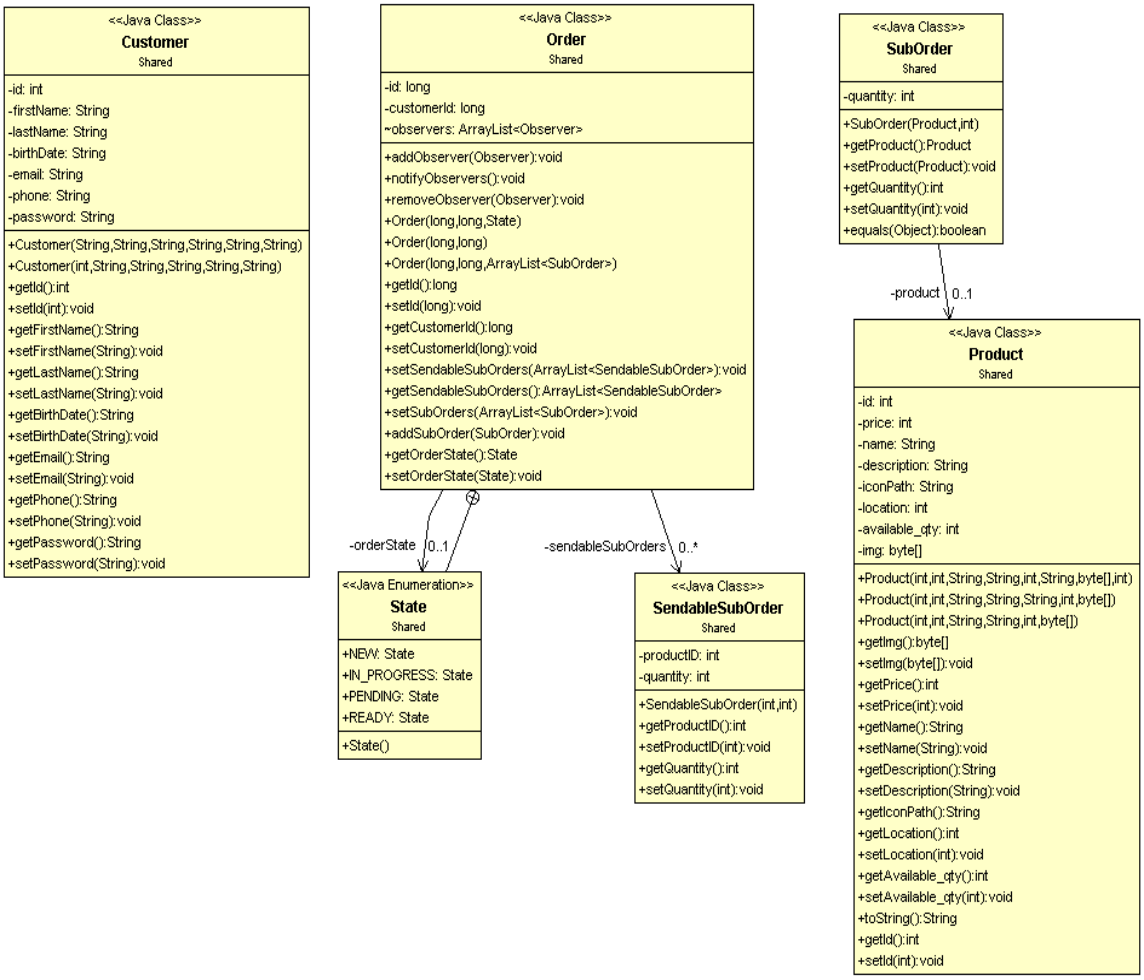

Figure 4.12 shared package UML diagram

4.4.2. Background threads

ProductsServer thread

This thread is responsible of sending the list of products on the database to the android clients. The productsServer thread starts by opening a TCP server socket port 5563 and keeps waiting for clients by calling the accept method on the ServerSocket object. Once a client is connected, a Socket object is returned from the accept method, and a new thread is created to handle the client and allow the ProductsServer thread to accept other clients while the first one is being processed.

The new thread starts by querying the products from the database using the querryProducts method of the DBHandler object, the objects are stored in an ArrayList. Then an ObjectOutputStream is used to send the objects one by one to the client, finally the socket is closed and the thread is destroyed.

LoginServer thread

This thread is used to authenticate the user using their phone numbers and passwords. The LoginServer thread creates a ServerSocket that is bound to port 5565, after a client is connected a new thread handles it to allow the other to accept other connections. The handler thread creates an ObjectInputStream and an ObjectOutputStream objects to exchange data with the connected client. First, a string is received from the client, the string is in the form (**phone::password**) where phone is the phone number of the user and password is his password. The **::** is used to separate the two parts of the string, the split method is called for that purpose. A database query is then performed to retrieve the user ID having this phone number and password. The queryCustomerId method of the DBHandler object is called, if an entry on the customer table satisfies the two conditions, then the ID of the user is returned by the method, otherwise -1 is returned. The returned valued is sent to the client through the objectOutputStream, after that the socket is closed and the handler thread is destroyed.

OrdersServer thread

This thread is used to receive orders from the customers. As for the previous threads, the ordersServer thread keeps listening on port 5564, once a client is connected it is handled by a new thread. An Order object is received through the ObjectInputStream, a check operation is performed to ensure that the ordered products are still available and did not get out of stock while the user was selecting his products, if the order is not accepted a boolean value of False is returned to the client to indicate that the order is rejected, otherwise it is inserted in the database using the insertOrder method of the DBHandler object, then the quantity of each ordered product is subtracted from the corresponding available quantity. The order ID returned by the database is assigned to the order, its state is then set to state.PENDING, and a boolean value of True is returned to the connected client to confirm that the order is accepted. Finally the order is inserted in the orders queue to be processed when its turn comes, the socket is closed, and the handler thread is destroyed.

MyOrdersServer thread

This thread is similar to the previous threads, it is used to send the list of orders corresponding to a certain user, such that the user can see his orders and observe their states in real time. The thread starts a TCP serverSocket on port 5566, when a client is connected it is handled by a new thread. First, a value of type long is received, the value represents the user ID. The ID is used to query the list of orders corresponding to that user, the queryOrders(long ID) method of the DBHandler object is used for that. The elements of the returned arrayList are then sent to the client through the ObjectOutputStream. Once finished, the socket is closed and the thread is destroyed.

Robot Controller Thread

The robot controller thread is the thread responsible for the control of the robot. Its main function is to assign the next order to the robot to process it when available. The thread starts first by waiting for an order, once received it checks if the robot is available to process it, otherwise it waits until the robot becomes available and assigns the order to it to process it, as shown in the flowchart in **Figure 4.13**.

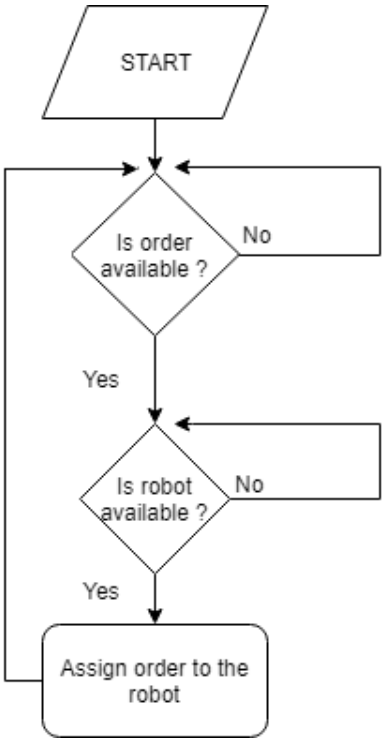

Figure 4.13 RobotController flowchart

This can be seen as a "producer consumer" problem, which is a well known problem in Operating systems and multi-process synchronization. The producer consumer problem can be seen at two levels : The first one concerns the orders, where the OrdersServer thread produces orders and the RobotController thread consumes them. The second one concerns the robot, it is produced by the thread that communicates with the robot, when the robot finishes its work it sends a packet to indicate that it is available again and is added to the queue, and then it is consumed by the RobotController thread when an order is available to process.

In order to solve the problem, a queue is used to store the elements. The queue has two **synchronized** methods : add(T item) and pop( ), which means when a thread calls one of these methods, any other thread trying to call them will block (stop execution) until the first one is done. When the consumer thread calls the pop( ) method and there is no available element, the wait( ) method is called, which causes the thread to be switched to the wait state, and wait until another thread calls the notify method of the same object. When the add method is called by the producer thread, the notify( ) method is called just after the element is added to wake up the consumer thread (if any) that has previously called wait( ) on that same object.

Two queues have been used in our case, one for the orders and another one for the robot, even though there is no real need to use a queue for the robot since we have a single one, however this approach have been chosen so that we could support multiple robots later on.

4.4.3. Database design

The program uses the JDBC java API to interact with an SQLite database. The database contains 4 tables which are :

- products table : this table contains the products of the supermarket, the database schema for this table is the following :

```
CREATE TABLE products
(
id INTEGER PRIMARY KEY AUTOINCREMENT,
product_name VARCHAR(20) NOT NULL,
description VARCHAR(40),
price FLOAT NOT NULL,
icon_loc VARCHAR(150),
location INTEGER NOT NULL UNIQUE,
available_qty INTEGER NOT NULL,
CHECK(location >= 0 AND price >= 0 AND available_qty>0)
);
```

- customers table : this table contains the customers of the supermarket, the database schema for this table is the following :

```
CREATE TABLE customers
(
id INTEGER PRIMARY KEY AUTOINCREMENT,
first_name VARCHAR(25) NOT NULL,
last_name VARCHAR(25) NOT NULL,
phone VARCHAR(13) NOT NULL UNIQUE,
email VARCHAR(50),
birth_date VARCHAR(10),
password VARCHAR(256) NOT NULL
);
```

- orders table : this table contains the orders of each customer, the database schema for this table is the following :

```
CREATE TABLE orders
(
id INTEGER PRIMARY KEY AUTOINCREMENT,
customer_id INTEGER NOT NULL,
order_state INTEGER NOT NULL,
FOREIGN KEY(customer_id) REFERENCES   customers(id) ON
DELETE CASCADE
);
```

- subroders table : this table contains suborders, such that each order is made of one or more suborders, the database schema for this table is the following:

```
CREATE TABLE suborders
(
order_id INTEGER NOT NULL,
product_id INTEGER NOT NULL,
quantity INTEGER NOT NULL,
CHECK(quantity > 0),
FOREIGN KEY(order_id) REFERENCES orders(id) ON
DELETE CASCADE,
FOREIGN KEY(product_id) REFERENCES products(id)
ON DELETE CASCADE
);
```

The UML diagram describing the relations between the 4 tables is shown in **Figure 4.14**.

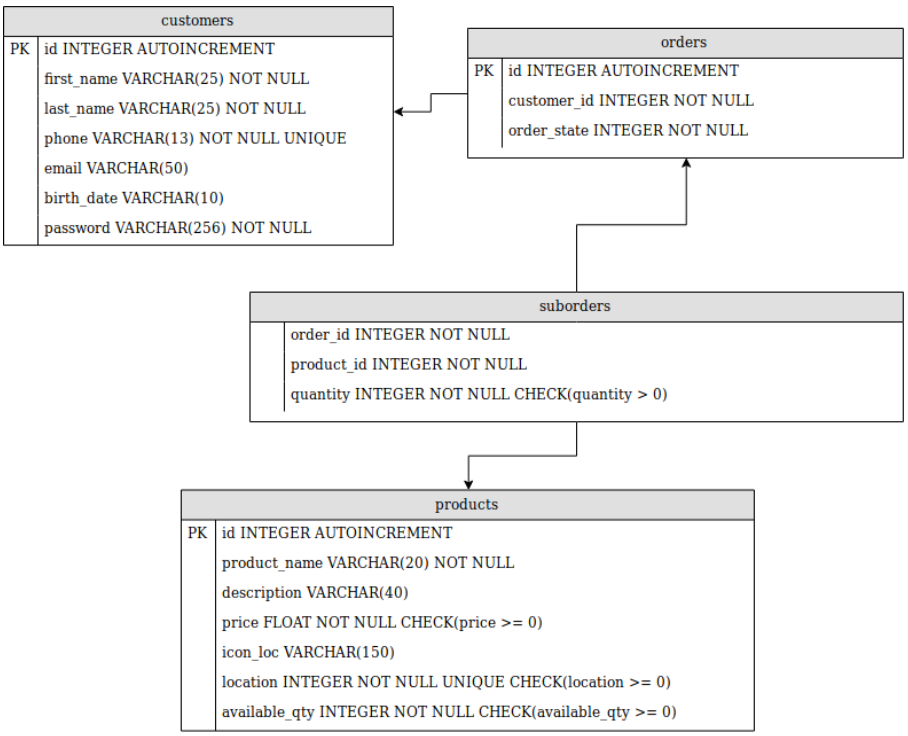

Figure 4.14 Database UML diagram

4.4.4. Graphical User Interface Design

The Graphical user interface (GUI) allows the user to manage the database (insert, update, delete products or customers), and to monitor the current activity of the robot (displacement and position on the map) as well as the list of orders received in real time with their respective states.

The GUI is developed using JavaFx and scene builder, it is divided into three tabs: one for the orders and the robot map, one for the products, and one for customers.

Orders tab

This tab contains the list of orders, the orders are added to the list as they arrive in real time, each cell of the list contains the order ID and its current state.

To the right of the orders list, the map of the super market is displayed as a grid of 20 x 20 numbered square cells, the robot is represented with a robot icon, and its real time position and orientation are shown on the map as shown in **Figure 4.15**. The map can be used to control the robot in manual mode, when a map cell is clicked, a packet is sent to the robot to move from its current position to the desired cell.

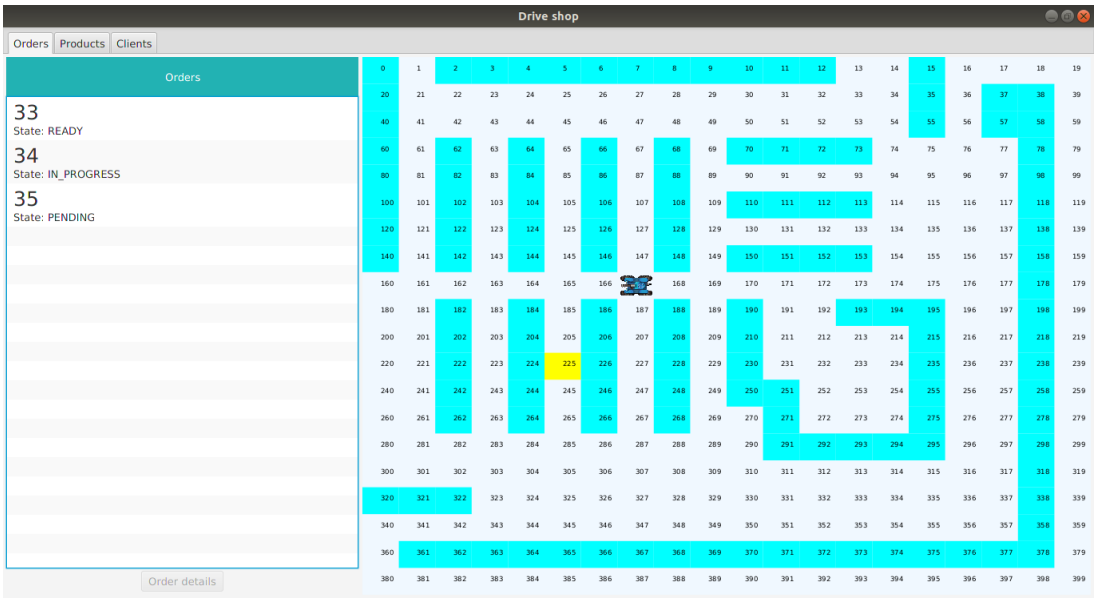

Figure 4.15 Orders tab

When an order is selected, the "order details" button is enabled and when it is clicked, a new window shows the details of that order : the customer name, customer ID, order ID, the total price, and a list of the ordered products with the corresponding quantity. **Figure 4.16** illustrates the order details window.

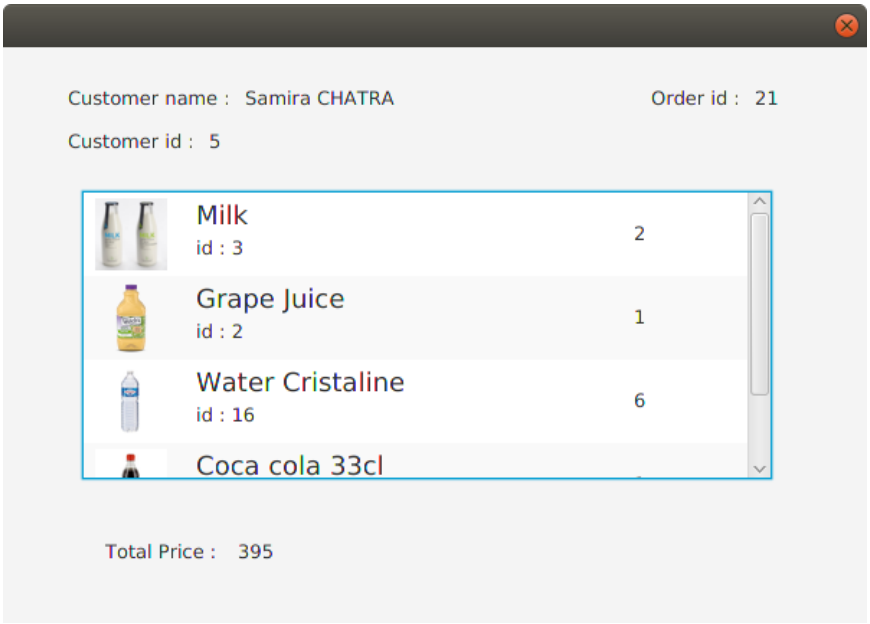

Figure 4.16 Order details window

Products tab

The products tab contains, on the left, the list of all the products in the database. Each list cell contains the name, the ID, and an icon of the product. When a product is selected (clicked), the details about that product (name, icon, ID, price, quantity and description) are shown on the right of the screen, an "Edit" and "Delete" buttons can be used to edit/delete the selected product. A search bar is placed on the top of the list, it can be used to filter the list when searching for a specific product, the search can be done using the product name or ID, by typing a set of characters on the search bar, the list content will be reduced to products having their name or ID starting by this set of characters. A screenshot of the products tab is shown in **Figure 4.17**.

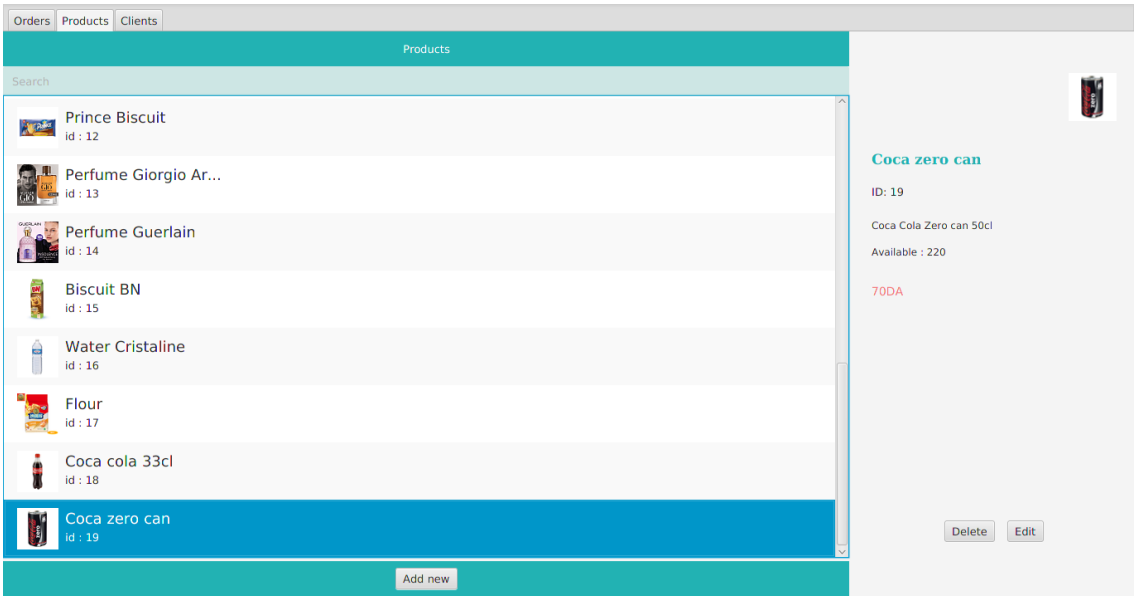

Figure 4.17 Products tab

An "Add new" button is on the bottom of the list, it is used to insert new products to the database, when clicked a new window is shown to prompt the user to enter the new product's information. In that window the user should enter the product's name, description, price, and available quantity, the format is checked using regular expressions, then he is asked to select the icon file by browsing the computer's hard drive files. Finally the location of the product is set by clicking on "search" button then clicking on the cell on the map that is displayed. The add new product window and the product location selection window are shown in **Figure 4.18.**

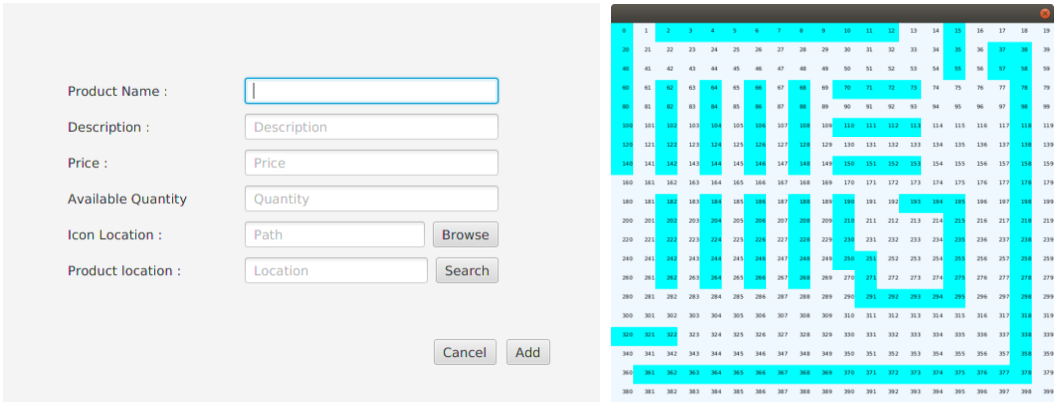

Figure 4.18 Add new product window (left), selection of product position (right)

Customers tab

In the customer tab, a list containing all the customers of the database is displayed on the left. Each list element contains the customer's full name and ID. A search bar on the top of the list allows filtering the list elements and showing only the ones having their first name, last name, or ID starting by the entered text. When a customer is selected from the list, the details about him is shown on the right, the details contain the customer full name, ID, phone number, email, and birth date, as shown in **Figure 4.19**.

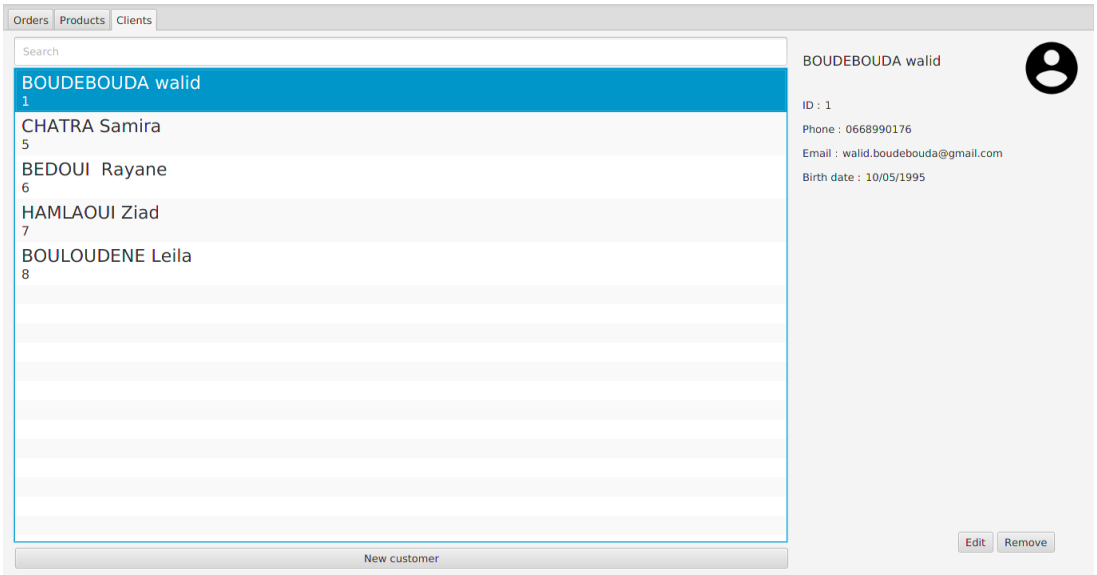

Figure 4.19 Customers tab

A "New customer" button on the bottom is used to add new customers to the database. When clicked, a new window appears to enter the information about the new customer as shown in **Figure 4.20**.

The input format is checked to avoid having corrupted or incorrect data on the database, regular expressions are used to check for the first name, last name, email, and phone number. For the birth date, a date picker is used to ensure that it is saved in the correct date format. The password field must be entered twice to ensure that the user does not make a mistake when typing it, if the two does not match, an error message is displayed.

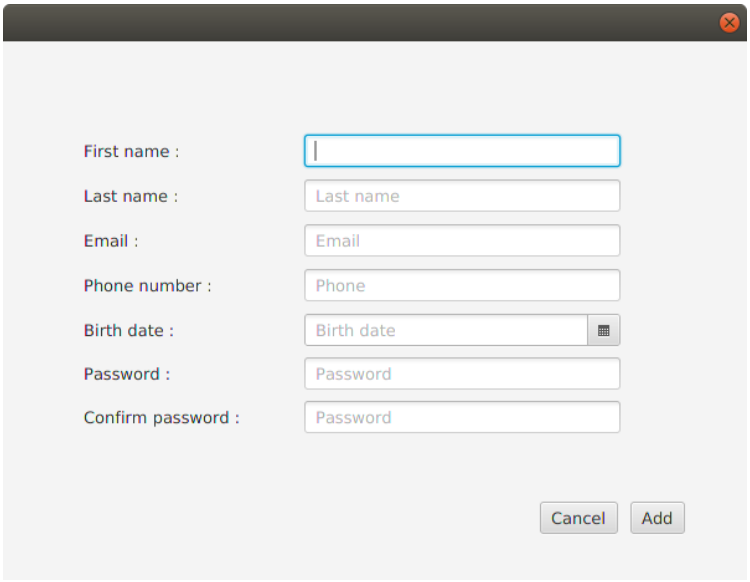

Figure 4.20 New customer window

4.4.5. Map representation

The map is represented as a grid of **d** x **d** square cells numbered from 0 to (d x d - 1), each cell can be either an empty cell or an obstacle as shown in **Figure 4.21**.

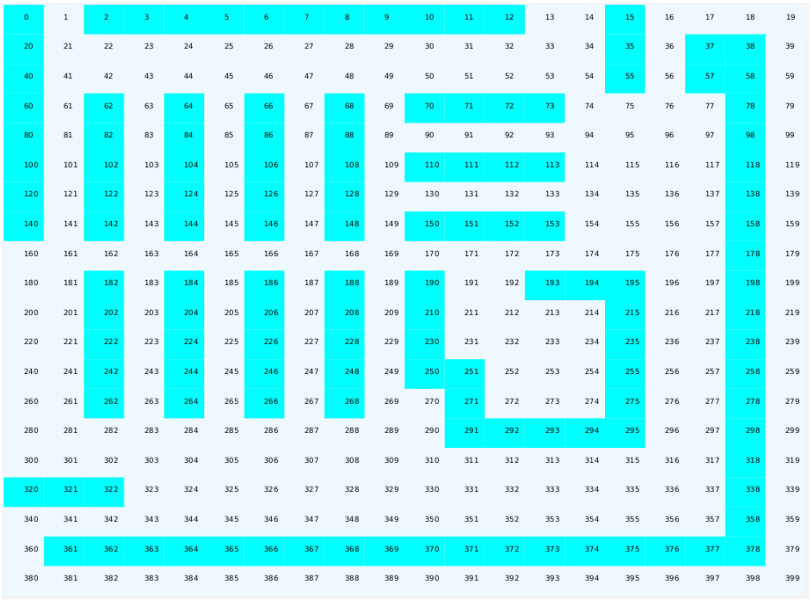

Figure 4.21 Map representation (d = 20)

When the robot is at one of the empty cells, it can move to any of the four adjacent cells if they are not obstacles (up, down, left, right). Diagonal moves between cells are not allowed.

Any cell can be represented in two ways : Either using (x, y) coordinates such that: $0 \leq x, y < d$, or using a single cell number, say **n**, such that : $0 \leq n < (d \times d)$, where **d** is the dimension of the map . The conversion can be done using the following formulas :

$$\mathbf{x} = \mathbf{n} \% \mathbf{d}$$

$$\mathbf{y} = \mathbf{n} / \mathbf{d} \quad \text{(Integer division)}$$

$$\mathbf{n} = \mathbf{y} \times \mathbf{d} + \mathbf{x}$$

The map cells are stored in a 2D array of type Cell, which is an enumeration that can take the values {Cell.OBSTACLE, Cell.EMPTY, Cell.DESTINATION}, the last value is used to colour the destination cell with a different colour when drawing the map. The loadMap() method is called when a Map object is constructed to initialize the array of cells, the values are read from a binary file called "default.map" where the obstacle cells are represented as 1s and the empty as 0s.

4.4.6. Path planning

For the path planning algorithm, two different approaches have been compared:

Single source shortest path approach

Single source shortest path algorithms are algorithms that find the shortest path from a **single** source node in a graph to **all** the other nodes with the sum of the edge costs on each path being minimal. The most famous algorithm for this problem is DIJKSTRA's algorithm.

Dijkstra's algorithm starts from the source node and then it selects the closest adjacent node based on an adjacency matrix, and checks if the distance of the path from the source node to any other node is greater than the path from the source node to the selected node plus the distance from that node to the destination. If the path through the intermediate node is faster, then the path is selected to be a shortest path to reach that destination. After testing all the possible nodes for that selected node, the node is said to be explored and the algorithm takes the next closest node and do the same until all the nodes are explored. If we want to find the path to a single

destination node only, the algorithm can be stopped as soon as the destination node is the next one to be explored.

Another algorithm known as A* algorithm is also used as a single source shortest path algorithm, it is based on Dijkstra's algorithm, and adds an extension to it to achieve better performances. The two algorithms are similar, except that, A* uses a heuristic to guide its search.

The optimization is added when searching for the nearest unvisited node, A* takes another value into account, which is the Euclidean distance (**h**) from any node to the destination, to guide the search algorithm to go faster towards the destination node without exploring unnecessary nodes.

Figure 4.22 shows a comparison between the two algorithms, the green cell with the robot represents the starting point, the yellow one is the destination point, and the red ones are the nodes that have been explored before finding the shortest path. We can clearly see that A* (on the left) is much more efficient than Dijkstra (on the right), and both algorithms generate the same optimal path length.

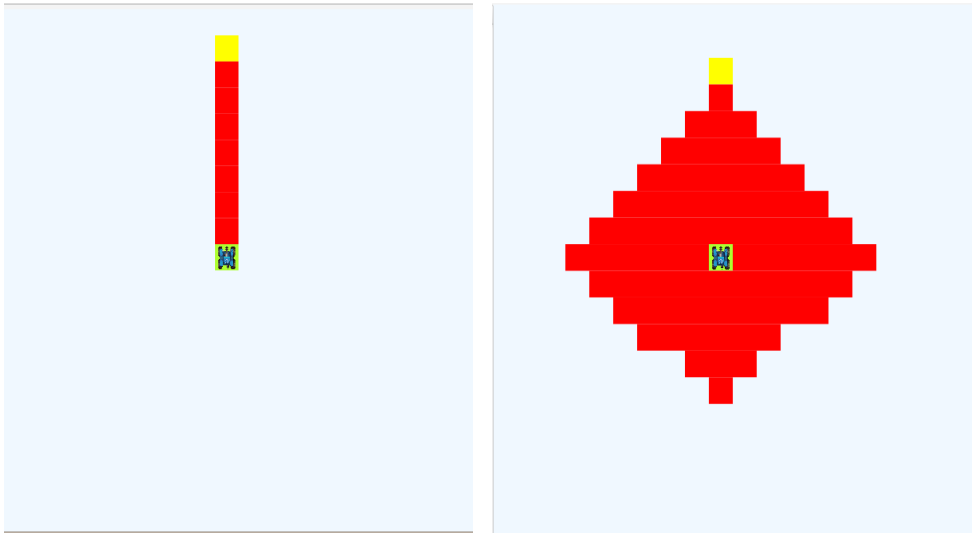

Figure 4.22 A* VS Dijkstra's algorithm

Let h[ ] be an array of size n such that :

$$h[i] = \sqrt{(Xi - Xd)^2 + (Yi - Yd)^2} \quad ; \quad 0 \leq i < n$$

Once all the values of h are calculated, they are used in the algorithm as follows :

1. Let c[ ][ ] be an adjacency matrix of size n x n and **s** our source node.
2. Create four arrays : d[ ] containing the distance from source node to all the other nodes (c[s][ ]), v[ ] array of booleans to mark the visited nodes, initially all the nodes are unvisited, p[ ] to store the last node to go through before reaching the node, initially -1 (no intermediate node), and h[ ] for the Euclidean distance from any node to the destination node **d**.
3. Pick the index **i** of the unvisited node having the smallest value of **(d[i]+h[i])**.
4. Compare the distance from the source node **s** to all the other nodes, with the distance from **s** to **i** plus from **i** to the other nodes (d[j] > d[i]+c[i][j] ; where 0≤j<n, n: number of nodes in the graph)
5. If the distance is smaller, update d[j] by d[i]+c[i][j] and set p[j] = i .
6. Mark **i** as visited (v[i] = true)
7. Repeat for 0 ≤ i < n , until all the nodes are visited.

At the end, we get d[ ] containing the shortest path's length from **s** to all the other nodes, and p[ ] containing the last node to go through before reaching any node. The following recursive function can be used to recover the shortest path from **s** to a target node, say **t :**

```
void path(int t) {
        if (p[t] == -1)
            System.out.print(" -> " + t);
        else {
            path(p[t]);
            System.out.print(" -> " + t);
        }
}
```

All pairs shortest path approach

The second approach that have been tested is using an all pairs shortest path algorithm, which means an algorithm that finds the shortest path between **every pair**

of nodes in the graph. One of the most used all pairs shortest path algorithms is known as FLOYD's algorithm (or Floyd-Warshall algorithm). Floyd's algorithm works as follow for a graph of **n** nodes :

1. Let s[ ][ ] a copy of our adjacency matrix of size n x n, and p[ ][ ] an array of size n x n containing the next node to go through, initially if there is a route from node **i** to **j**, then p[i][j] = j, otherwise p[i][j] = -1
2. Let **i**,**j**, and **k**, be three variables such that $0 \leq i, j, k < n$, initially i=j=k = 0
3. For every pair (i,j), check if p[i][j] > p[i][k] + p[k][j], if true, update s[i][j] by s[i][k] + s[k][j] and set p[i][j] = p[i][k], otherwise continue
4. Increment k and repeat step 3 until k = n-1

At the end of the execution of the algorithm, we get p[i][j] containing all the next nodes to move to, to go from node **i** to **j** for any $0 \leq i, j < n$

The total path from i to j can be recovered from p[ ][ ] using the following function :

```
void getPath(int src, int des) {
    System.out.print(src + " -> ");
    while (p[src][des] != des) {
            System.out.print(p[src][des] + " -> ");
            src = p[src][des];
    }
    System.out.print(des);
}
```

Floyd's algorithm has a time complexity of $O(N^3)$, however some optimization can be added to speed up the algorithm : Since our graph is an undirected graph, it means that the distance from i to j is the same as the distance from j to i. this can be used to reduce the number of iterations such that once we find a new shortest path from i to j, we apply the same to go from j to i, and thus the algorithm can be applied to the upper half of the matrix only and have the lower part being a copy of it. The nested loops used by the algorithm would look like :

```
for (int k = 0; k < n; k++) {
      for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                  if ( s[i][j] > s[i][k] + s[k][j]) {
                        s[i][j] = (s[i][k] + s[k][j]);
                        p[i][j] = p[i][k];
                        s[j][i] = (s[j][k] + s[k][i]);
                        p[j][i] = p[j][k];
                  }
            }
      }
}
```

The number of iterations in this case is reduced from n^3 to :

$$n \times [(n-1) + (n-2) + (n-3) + \ldots\ldots + (1)] = n \times [\frac{n}{2}(1+n-1)] = \frac{n^3}{2}$$

A ragged array could be used to represent the matrix since only one half is used and the other half is a copy of it, this would save half of the memory space used but would require to perform a check on i and j when accessing s[i][j] or p[i][j] such that i should be always greater than j. But since the arrays need to be accessed frequently, it is preferred to minimize the access time at the expense of memory, so the ragged array won't be used in our application.

A last optimization is added to our algorithm to skip some unnecessary iterations as follows :

- If k is a node that represents an obstacle, then it is obvious that no shortest path would be updated if we go through that node (since there is no path to it), skipping this step will avoid $\frac{n^2}{2}$ unnecessary iterations for each obstacle.
- If **i** is an obstacle node, or **i** is equal to **k,** or there is no path from **i** to **k,** then there is no shortest path that would be updated **,** skipping these steps will avoid doing **n-i** unnecessary iterations.

The final code for our Floyd algorithm path finding function is the following :

```
for (int k = 0; k < n; k++) {
     if (getCell(k) == Cell.OBSTACLE)
        continue;
     for (int i = 0; i < n; i++) {
        if(getCell(i)==Cell.OBSTACLE||s[i][k]==Short.MAX_VALUE||i==k)
             continue;
         for (int j = i + 1; j < n; j++) {
              if ( s[i][j] > s[i][k] + s[k][j]) {
                             s[i][j] = (s[i][k] + s[k][j]);
                             p[i][j] = p[i][k];
                             s[j][i] = (s[j][k] + s[k][i]);
                             p[j][i] = p[j][k];
               }
          }
     }
}
```

Comparison between the two approaches

Both approaches can be used in our application. For the first one, once an order is received and the products positions read from the database, the A* algorithm will be executed to calculate the path from the current position to the next product. Once the robot reaches the product, the path from that product to the next one is calculated again, and so on until the robot collects all the products and gets back to the starting position. For the second approach, Floyd's algorithm will be executed only once, after the generation of the map, then the results will be stored in the p[ ][ ] array, and each time a path between two nodes is needed it can be retrieved directly from p[ ][ ]

Since in our application the robot will be frequently moving, the first approach would require to do the same calculations several times because the paths will not be stored, this would add more computational charge to the server (or the robot) and more time delay for the orders processing. Although the second algorithm takes much more time to execute compared to the first one, it has the advantage of being executed only once, at the starting of the program. Since in our application we are considering that the map is fixed and all the obstacles are static, Floyd's algorithm is the one that best suits our application.

Order of collection

For improving the robot displacement, the products are ordered in a way such that the closest one from the starting position is the first one to be collected, then the one which is the closest to that product is the second, and so on until all the products are collected. This is done in the following way :

1. Let L[ ] be a list containing the **n** products that need to be collected, set two variables: **src** initially equal to the starting node, and **i** initially 0.
2. calculate the path lengths from **src** to L[j] (s.t : $i \leq j < n$)
3. swap the position of the product with the smallest path length with the one at position **i**
4. set src = L[i] then increment i
5. repeat from step 2 until i is equal to n.

4.5. Robot software

The robot software is the program that is run by the ESP32 microcontroller to control the robot displacement and communicate with the server. Initially, the robot's position and orientation are saved in two variables : the robot orientation is represented by a byte that can take 4 different values : 1 for North, 2 for East, 3 for South, and 4 for West. The robot position is saved in a variable of type short, it represents the cell number where the robot is at the current moment. As shown previously, the robot communicates with the server through a wireless WIFI connection inside a local area network using TCP protocol, which is a reliable protocol that guarantees the transmission of data with no errors. When starting, the ESP32 connects to the WIFI network and gets an IP address, then it creates a TCP server and keeps waiting for connections on port 1234. When a client is connected, the client sends a packet that is read and stored in a byte array buffer, the processPacket(byte packet[]) method is then called to process the received data.

The packet contains the list of coordinates of the points where the robot has to go in order to reach the next destination. The first two bytes of the array are reserved : packet[0] contains the length of the received data array, packet[1] determines if the destination point is a product cell (should mark a pause to collect it), or a normal destination point (ex : come back to the starting point).

The rest of the array contains the successive cell numbers that the robot should go through to reach the destination, each cell number is represented by two bytes since a single byte would restrict the cell numbers to be between 0 and 255, which is not sufficient for our 20x20 map. Bitwise operators are used to recover the cell numbers as follows :

```
for (int i = 2 ; i < packet[0]; i += 2) {
    short next = (packet[i] << 8) | (packet[i + 1]);
    //...Move to next ...
  }
```

Once **next**, which is the next point to move to, is determined, the checkOrientation(short next) function is called. This function determines in which orientation the robot should be to move from its current position to **next,** in the following way :

Suppose that the robot is currently at some position **p** on the map as shown in **Figure 4.23.**

P-42	P-41	P-40	P-39	P-38
P-22	P-21	P-20	P-19	P-18
P-2	P-1	P	P+1	P+2
P+18	P+19	P+20	P+21	P+22
P+38	P+39	P+40	P+41	P+42

Figure 4.23 Robot position

Since the robot can only rotate left or right by 90 degrees or go straight, then the robot can move to one of 4 adjacent cells, as shown in grey in the figure. The coordinates of the cells where the robot can move are : {P-20, P-1, P+1, P+20}, a simple calculation can be done, as shown in **Table 4-1**, to determine the orientation in which the robot must be, so that when it goes straight by one cell it moves from **P** to **next** :

Table 4-1 Needed orientation with respect to (current position - next position)

(P - next)	Orientation
-1	2 (E)
1	4 (W)
20	1 (N)
-20	3 (S)

Once the orientation is determined, the setOrientation(byte o) is called to set the robot on that orientation before going straight by one cell. First, the setOrtientation(byte o) method checks if the robot is already on the right orientation (ie : current orientation = o), then if true the method returns directly without doing anything since we don't need to rotate the robot. Otherwise, the function calculates in which direction and how many times the robot needs to rotate by 90 degrees. **Table 4-2** shows how the calculation is done :

Table 4-2 Calculation of number and direction of rotation

(Current orientation - o)	Rotation
-1	Rotate right x1
1	Rotate left x1
-2	Rotate right x2
2	Rotate left x2
-3	Rotate left x1
3	Rotate right x1

Once the direction and the number of times are determined, the corresponding function is called to activate the motors and perform the action.

Two functions are implemented to perform the rotation actions: turnRight(int n) and turnLeft(int n), where **n** is the number of 90 degrees rotations that should be done. Another function stepForward(int d) is used to go straight by d centimetres. The three functions use an algorithm to synchronize the speed of the two motors, this is performed using the encoders discussed previously. The encoders are also useful to determine the distance travelled by the robot, it can be used to make the robot travel for an exact distance in centimetres by calculating the number of ticks corresponding to that distance. Given that the number of holes on the encoder wheel is 20, and the radius of the robot wheel is 3.5 cm :

$$\text{circumference of the wheel} = 2 \times \pi \times 3.5 \approx 21.99 \text{ cm}$$

$$\text{ticks per centimetre} = 20 / 21.99 \approx 0.91 \text{ ticks/cm}$$

Using this value, we can calculate the needed number of ticks to travel an exact distance. The flowchart of **Figure 4.24** describes the algorithm used to turn the motors for a precise distance and keep the speeds synchronized.

The first step is to enable the sensors interrupts, meaning that each time the encoder rotates by one position, an interrupt will be triggered and the ISR for that sensor will be called. Each ISR contains a single instruction, which is to increment a counter, L_count is used for the sensor on the left wheel and R_count is used for the sensor on the right one. Then according to the direction (left, right, or forward), the direction control pins are activated (IN1, IN2, IN3, IN4 of the H-Bridge) to set the direction of rotation of the wheels. After that, an initial speed is set to both motors by setting the PWM duty cycle of the pins connected to EN1 and EN2 of the H-Bridge, the duty cycle for each motor are stored in **L_PWM** and **R_PWM**. Then, as long as the tick counts of one of the motors is smaller than the desired one **t_count**, the speeds of the two motors are compared by counting the actual tick count minus the previous one for each motor.

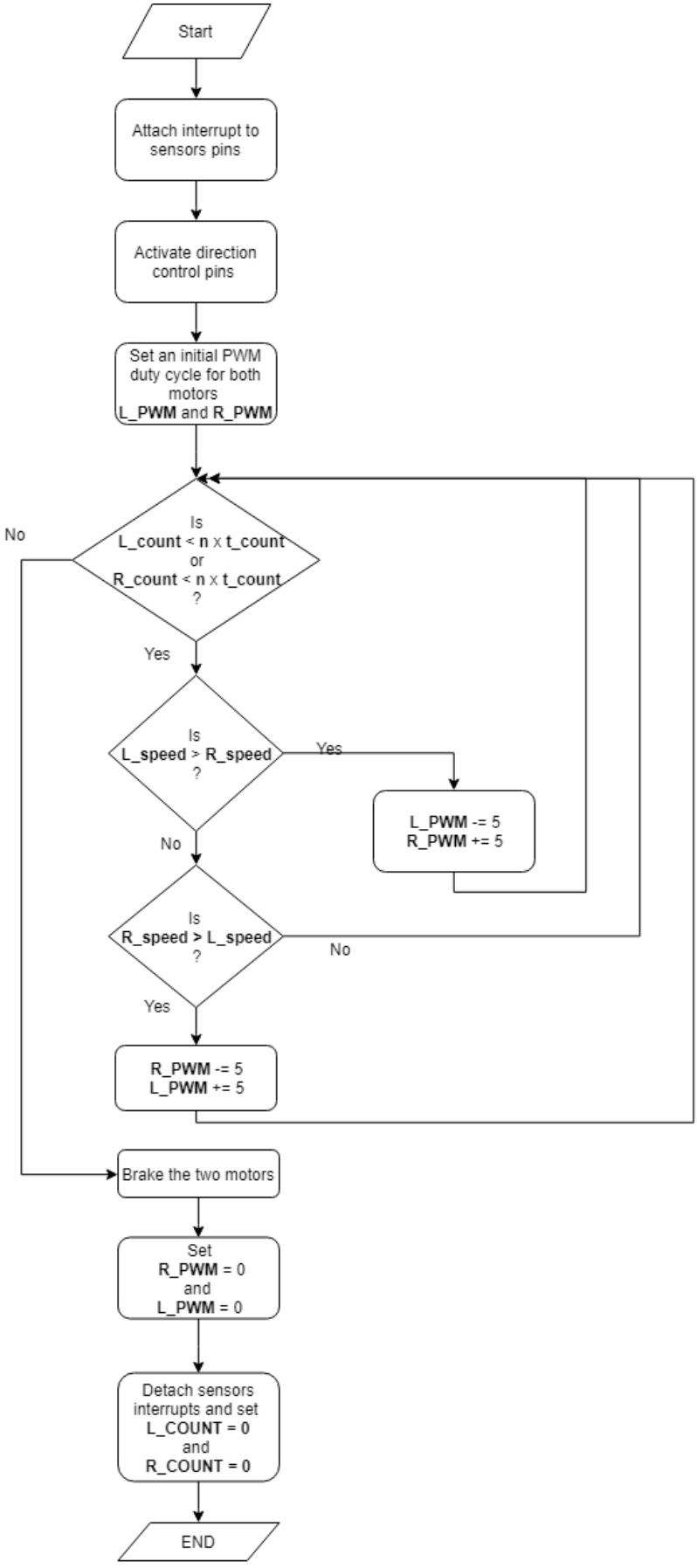

Figure 4.24 Robot motors control algorithm flowchart

If one is greater than the other, the speed of the slower one is incremented and the speed of the other is decremented (using L_PWM and R_PWM). After **t_count** is reached, a brake function is called to prevent the motors form rotating freely, the brake function inverses the direction of rotation of the motors for a brief delay, then stops the two motors. Finally, the PWM duty cycles, L_count, and R_count are reset to 0.

After each move (turn left, turn right, or step forward), the ESP32 sends a packet to the connected client as an array of bytes in order to keep the server updated of the current position and orientation of the robot and display it in real time on the GUI. Three types of update packets are used : when data[0]=1, the packet is a position update, the new position of the robot is sent in the two following bytes data[1] and data[2], when data[0]=2 the packet represents an orientation update, the new orientation of the packet is sent in data[1], when data[0]=0 the packet indicates that the robot reached its destination and is ready to receive a new one.

Figure 4.25 shows a picture of the final prototype of the robot with all the parts assembled.

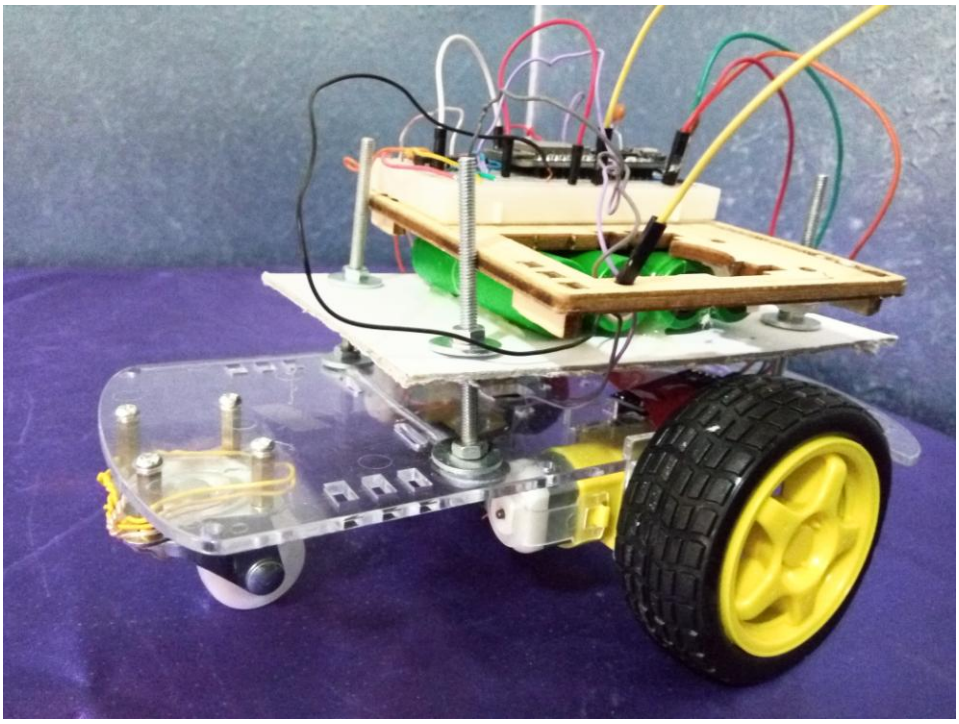

Figure 4.25 Picture of The Robot

Chapter 5

Conclusion

The design and implementation of a robot based drive through system for a super market, consisting of an Android application, a java program, and a mobile robot controlled by an ESP32 microcontroller is described in this report.

The software and hardware parts worked as expected and met successfully all the objectives set at the beginning of the project mainly : products selection and ordering process, the reception and processing of the orders, the products and customers database administration, the orders state and robot position monitoring in real time, and finally the robot displacement and path planning.

Some constraints were encountered during the hardware design. Having the speed of both wheels of the differential drive robot synchronized was the most challenging part, due to the quality of the low cost motors and sensors used, and to some mechanical constraints. Finding batteries that had enough power for the microcontroller and the motors and being light enough to be put on the robot was also a difficult task. For the software part, testing and debugging a code of such a big size was a new experience that required a lot of organization especially for the java program. It was also interesting to explore the networking part and find the different protocols that allowed the three parts of the project to communicate through the network.

This work covered different topics in the field of computer engineering such as: software design, GUI design, databases, multithreading, path planning, network communication, embedded systems, interfacing, as well as some control techniques. It was a good opportunity to put into practice the different skills learned during the past 5 years to build a concrete project with its different parts.

Nevertheless, a technical project is never complete. The action of collecting the products was not covered due to the mechanical complexity and lack of equipments. Further improvements of the project may include :

- The use of a multi-agent path planning and collision avoidance algorithm to support multiple robots.
- The use of image processing or sensors to detect dynamic obstacles.
- Include a robotic arm or some other mechanism to perform the products collection action.

References

[1] Wired, "Amazon robotics - Wikipedia," june 2019. [Online]. Available: https://www.youtube.com/watch?v=8gy5tYVR-28. [Accessed 06 15 2019].

[2] Z. Z. Zhao G., "Design and Implementation of the Online Shopping," *WISM 2012. Lecture Notes in Computer Science*, vol. 7529, 2012.

[3] M. Z. A. Rashid, T. Izzuddin, N. Abas, N. Hasim, F. Azis and M. Aras, "Control of Automatic Food Drive-Through System using," *International Journal of u- and e- Service, Science and Technology*, vol. 6, no. 4, p. 10, 2013.

[4] Espressif Systems, "ESP32 Series datasheet," Espressif Systems, 2019.

[5] Statista, " Statista," 2019. [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. [Accessed 1 June 2019].

[6] Android, "Meet Android Studio," [Online]. Available: https://developer.android.com/studio/intro. [Accessed 03 Juin 2019].

[7] Android, "Projects overview | Android studio," [Online]. Available: https://developer.android.com/studio/projects/index.html. [Accessed 03 June 2019].

[8] Android , "Introduction to Activities | Android developers," [Online]. Available: https://developer.android.com/guide/components/activities/intro-activities. [Accessed 10 06 2019].

[9] Android, "Understand the activity lifecycle | Android developers," [Online]. Available: https://developer.android.com/guide/components/activities/activity-lifecycle. [Accessed 10 06 2019].

[10] Android, "Layouts | Android developers," [Online]. Available: https://developer.android.com/guide/topics/ui/declaring-layout.html. [Accessed 12 06 2019].

[11] Google developer training, "7.1: AsyncTask and AsyncTaskLoader · GitBook," [Online]. Available: https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-3-working-in-the-background/lesson-7-background-tasks/7-1-c-asynctask-and-asynctaskloader/7-1-c-asynctask-and-asynctaskloader.html. [Accessed 13 06 2019].