CS3342 : Software Design

Group -27
Written Report

Table of Contents

Group Members:

- MALHOTRA, AVI (55773896) – PROJECT MANAGER
- GUPTA, AARNAV (55990960) – ASSISTANT PROJECT MANAGER
- BANBAH, KUSH (557867405) – SOFTWARE DESIGNER
- JAIN, UTKARSH (55992915) – SOFTWARE DESIGNER
- KASLIWAL, ARYAN GIRISH (55972222) – SOFTWARE DESIGNER
- RAJAGOPALAN, PRATUL (55858290) – SOFTWARE DESIGNER

1. Introduction

1.1 Background

In today's society, one of the main concerns are the effects of overpopulation. In 2020, the world population reached 7.8 billion and it is estimated that by 2030, the population will reach a staggering 8.5 billion. With such an enormous population that is growing rapidly, food consumption across the world will also increase. To meet the needs of an increasing population, smart agriculture should be adopted as it could help bring about sustainable agricultural production.

Smart agriculture is where advanced technologies are used and integrated into traditional farming techniques in order to increase the quality, quantity and efficiency of the yield. Smart farming also helps in automating farming by collecting data for future analysis. This data collected could be used to present users with more information, thereby helping them make better decisions and improve the final product. The system can optimize and examine how processes can be done in different ways by using high-tech solutions.

Most large-scale farmers already have access to the latest technology and are capable of spending large sums of money on buying the best equipment possible. Our app aims to help small-scale and medium-scale farmers who otherwise would not be able to practice smart farming.

The app does not make decisions for the farmer, it gathers the data and presents it to the farmer, making it easy for the farmer to make decisions that would have otherwise taken a lot of time.

1.2 Feasibility

There will be an initial cost to purchase and set up the sensors, and a smaller periodic cost for their maintenance. Apart from that, a constant (small) power supply to keep the sensors powered up would be required. All of this would fit in most medium-scaled farmer’s budgets and will be feasible in most conditions. The sensors would have to be designed keeping in mind that they will undergo natural wear and tear due to water or wind.

The sensors must be cheap enough for small farmers to be able to use them, and the app must be made simple enough to provide clear data to farmers who may be technologically illiterate. Farmers must be convinced that this app will help them before they start using it since the initial setup process (including the cost) may seem daunting to some farmers. Confidence among farmers could be built with the help of many advertisement campaigns and with support from the government to create awareness about the app and the benefits it provides to farmers.

1.3 Downsides/ Risks

The farmer must set up an array of sensors around the crops they are growing to ensure that the system can collect enough data. These sensors are cheap, but farmers may be technologically illiterate and may not know how to set them up correctly. If the farmer setups up the sensors incorrectly they may give incorrect data and damage their farm. Moreover, there is a risk of them damaging the sensors, the connections and or the hardware components while performing setup.

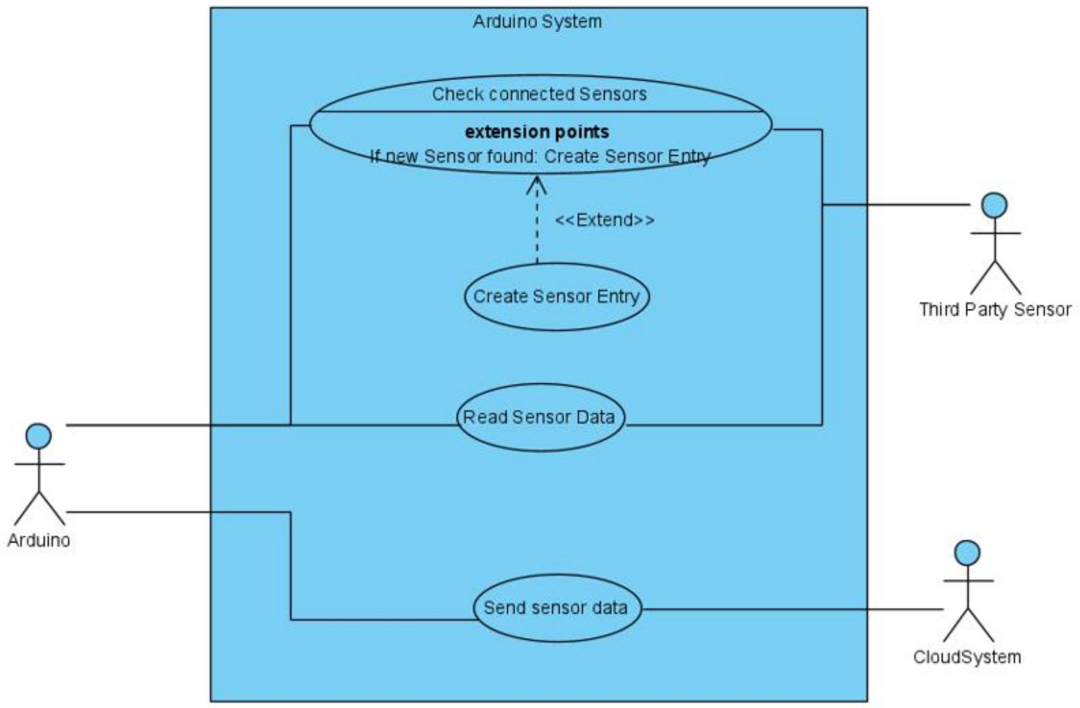
The sensors and hardware components are exposed to the elements (harsh sunlight, rain and wind) and may require maintenance. The farmer may have to devote time to sensor maintenance which takes away from time that can be spent performing other activities.

If there is a bug with the code or if the parts need to be replaced, it can be quite difficult logistically since it must be set up physically and bugs cannot be fixed online for the parts on site. If there is a bug in the on-site hardware, then a major recall and update would have to be done so extensive testing must be done.

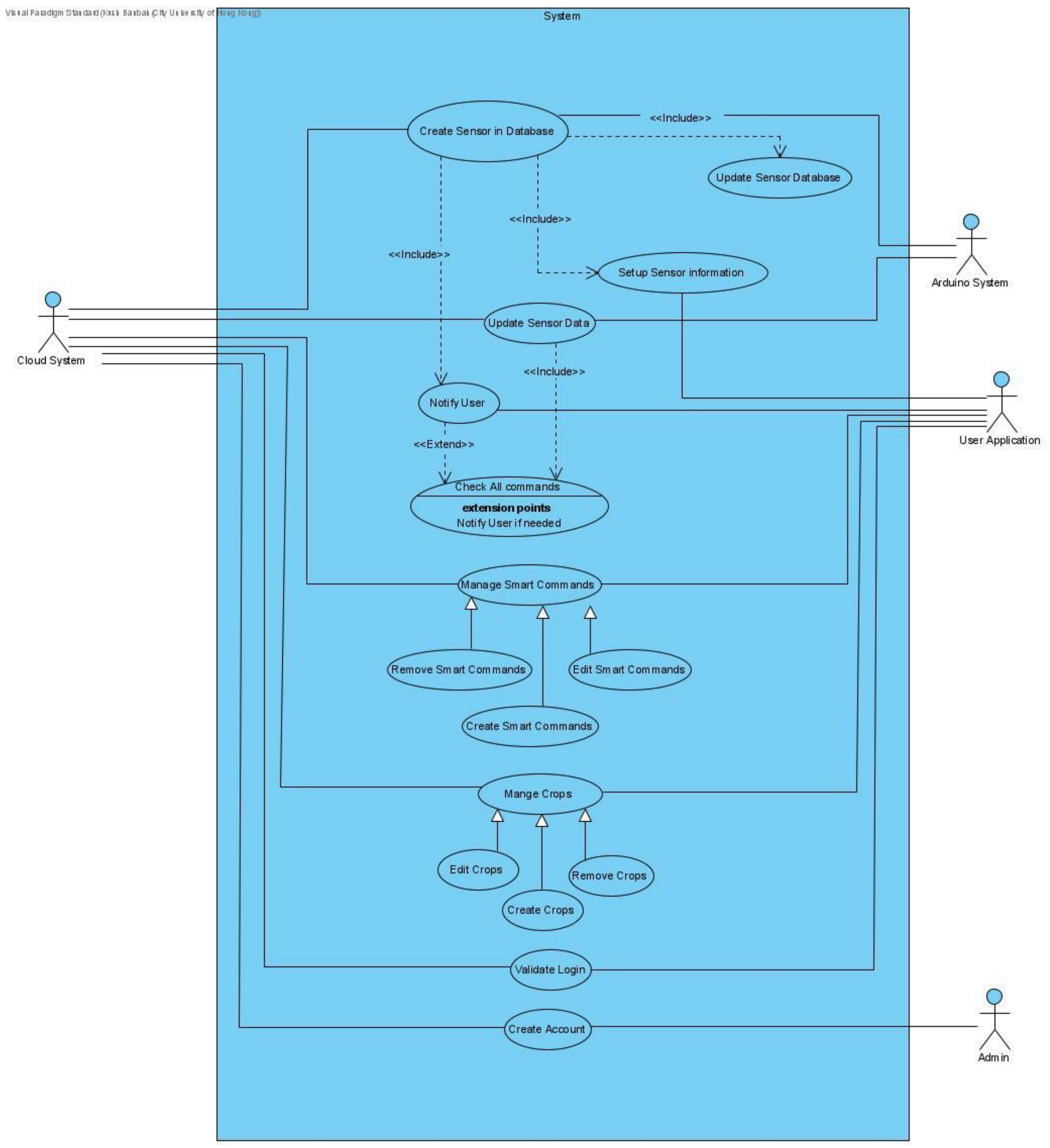
2. Scope and Requirements

2.1 Use case diagrams

This entire solution is essentially three different systems interacting with each other. Beginning with the Arduino system at the lowest level which sends data to the cloud system and then it's communicated to the user application system.



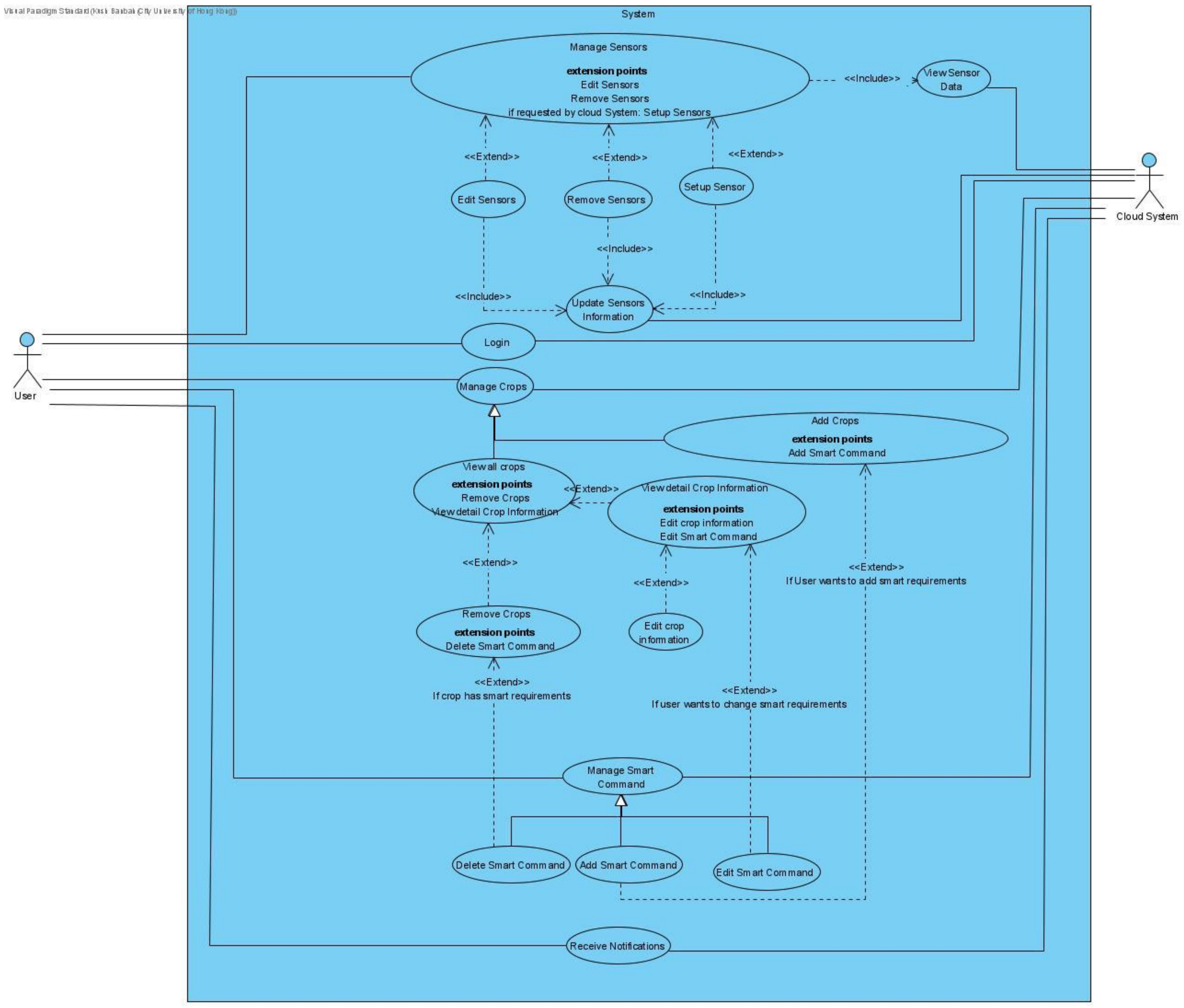
The above figure is the use case diagram for the Arduino. The Arduino's main job is to communicate between the sensors and send that data to the cloud system. It also needs to recognize if any new sensors have been added and include them when sending data to the cloud system.



The next system is the cloud system. This is the focal point for the entire system by acting as a bridge between the Arduino and user application. The cloud system has three main responsibilities. The first responsibility is that it has to store all the data for the user. This data is their crop information, ‘smart commands’, sensor information and the actual raw data of all the sensors as well.

The next main responsibility of the system is that it accomplishes the Arduino use cases of receiving data, it has to connect to the Arduino, parse and store all the data and act as a go between the Arduino and the user application. The Arduino for example will send raw data and sensor information, the cloud system will then create a new sensor and send a notification to the user application to ask the user to name and setup the sensor.

Lastly, the cloud system also manages user’s use cases. It provides all this data to the user application and updates it according to user application. These are described by the Manage Smart commands, manage crops and updating sensor information use cases. ‘Smart commands' are essentially preset conditions by the user. In other words, the user can set a range of values for the sensors, and if the value is not within these bounds, then the user will be notified; hence the notify use case. So, the cloud system, when receiving data, checks if any of the smart commands have been triggered, and if so, notifies the user. It also provides functionality for the user application to login. The admin also creates accounts directly into the system when a new user is created.



The cloud system is the interface between user application and the Arduino system, the Arduino system is the interface between the sensors and cloud, so lastly, the user application is the interface between the user and the cloud. The user application firstly allows user to manage his/her crop and smart command information directly from the cloud. They can add, edit or delete any of these. The user can also manage sensor information, add, edit, or setup any of these, or choose to delete it. The cloud system provides these use cases, so the database is updated concurrently. The user application also receives the notifications sent by the cloud about either setting up new sensors or if a smart command is triggered and notifies the user.

2.2 Use case tables

The project has three core use cases and the detailed stepwise process of all three have been explained below:

2.2.1 Check plant information

The farmer can request the app for the most recent recorded sensor data, the app in turn will request the cloud for this data and then display it to the farmer. The postcondition of this use case is that the farmer can then take an informed decision. It is a simple 3 step process, and it does not involve the Arduino system at all. It is all communicated with the cloud which has the most recently received dataset stored. This use case has one extension point. Instead of viewing the current sensor data, the farmer can choose to view the sensors condition, change the sensor, add new sensors, or configure the current sensor to a different threshold.

Use case Name:	Check sensor data	
Actor(s)	User and Cloud system	
Description:	This use case describes the process the farmer can adapt to timely provide nutrients and water to plants.	
Typical course of steps:	**App (system) action**	**Cloud system action**
	Step 1: The system can access all sensor related data. It receives the “view sensor data” request from the user and in turn sends view data requests to the cloud.	**Step 2:** The cloud receives the “view sensor data” request and sends the most recently received data to the system.
	Step 3: The system displays the data received from the cloud on the app.	
Alternate course of events	**Step 1a:** The system manages the sensor data instead of accessing it. This allows the user to “check sensor condition”, “remove sensor”, “add new sensor” or “configure sensor” in a different way. All changes are also reflected on the cloud.	
Precondition:	The farmer wants to check the app for any updates from the sensors.	
Postcondition:	The farmer receives all the relevant information.	

2.2.2 *System updates the plant information*

The Arduino system automatically updates all sensors information to the cloud every 15 minutes. This is done in order for the complete system to store the most recent sensory data which gets updated every 15 minutes. We chose a time frame of 15 minutes because the conditions of plants does not change drastically within 15 minutes, whereas a very short time frame would burden the system and a longer time frame might lose some soil information.

When the cloud receives the sensor information, it processes it and checks every single data point against the threshold set by the farmer. At this point the post condition is that if sensors have values above the set threshold, then the dataset is stored in the cloud as the most recently received dataset and keeps it ready for the next time farmer requests to view data.

Use case Name:	Update sensor information	
Actor(s)	Arduino and Cloud system	
Description:	This use case describes the method by which the arduino constantly updates the sensory data to the cloud.	
Typical course of steps:	**Arduino (system) action System response**	**Cloud system action**
	Step1: The arduino system has a 15 minute clock cycle and it "updates sensor data" to the cloud periodically.	**Step2:** Everytime the cloud receives the sensory data from the system, it stores it as the most recent data point. Keeps it ready to send to the app upon request. Furthermore, the system processes the data and checks it with the users' set requirements.
Alternate course of events	**Step 1a [Extension Point]:** The farmer requests the app to "view sensory data". Then the 15 minute clock is reset to 0 and the same cycle restarts. **Step 2a [Extension Point]:** If any sensors have a reading below the "user set readings" then an alert is sent to the app.	
Precondition:	The arduino systematically updates all information to the cloud.	
Postcondition:	The cloud systematically stores the latest dataset and keeps it ready to be sent to the app.	

2.2.3 Send alert notification to the farmer

Every time the cloud receives a data set, it checks it against the threshold value set by the farmer. If at any point the sensor value is less than the threshold value, then a notification is sent to the farmer. The postcondition is this notification is an alert to notify the farmer than a plant is lacking a particular nutrient. Along with the notification, the complete data set is sent to the farmer so that the farmer can look at all the details of the alert.

Use case Name:	Send alert to the farmer	
Actor(s)	App and Cloud system	
Description:	This use case describes the method by which the app processes the dataset from sensors.	
Typical course of steps:	**App (system) action System response**	**Cloud system action**
	Step1: Data points from the sensors are sent to the cloud, either by the farmer's request from the app or because of the 15 minute clock interrupt. (Both use cases have been explained above)	
		Step2: Cloud processes the received data. If any sensors have a reading below the "user set readings" then an alert is sent to the app. Additionally, the complete data set is also sent to the app for detailed display.
	Step 3: The app shows an alert in the notifications as well as receives the complete data set, which the farmer can see via "view sensory data".	
Alternate course of events	**Step 2a [Extension Point]:** If any sensors do not have a reading below the user set readings then no alert is sent to the app.	
Precondition:	The sensors record a lack of nutrients or water for the plants.	
Postcondition:	The farmer receives all the relevant information and satisfies the plants' needs.	

3. Prototype

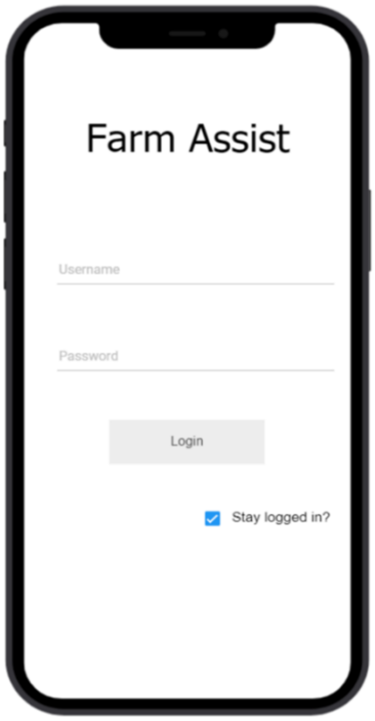


Fig. 1

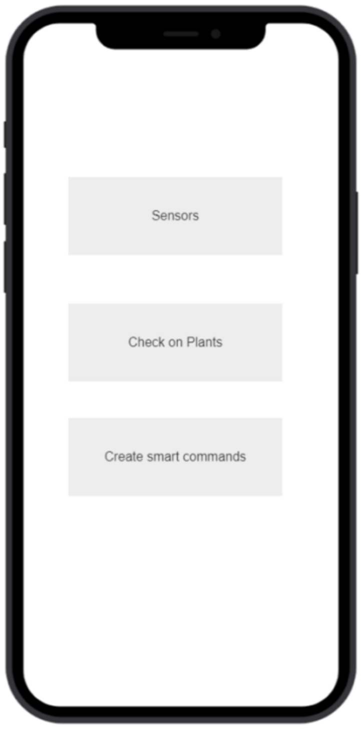


Fig. 2

Fig. 1 is our login screen. When a user successfully logs in, they are greeted by our main menu as shown in Fig2. Our main menu gives the user 3 options to choose from. First, Sensors to view information directly from sensors: Second, Check on Plants, to check information regarding crops; third, create smart commands, to view and edit smart commands.

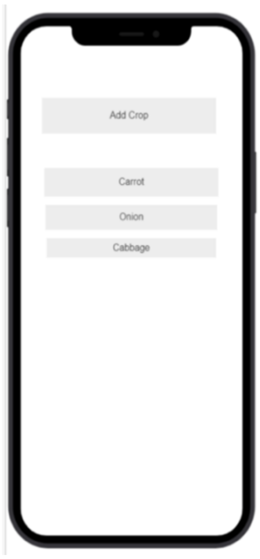
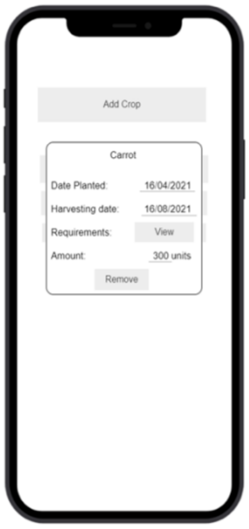
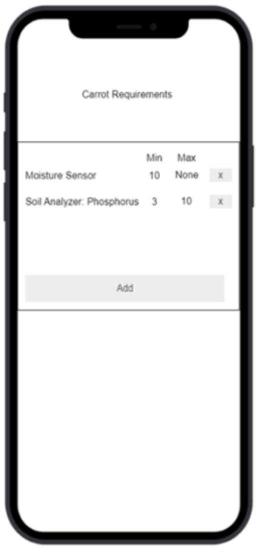


Fig. 3



Date Planted:	16/04/2021
Harvesting date:	16/08/2021
Requirements:	View
Amount:	300 units

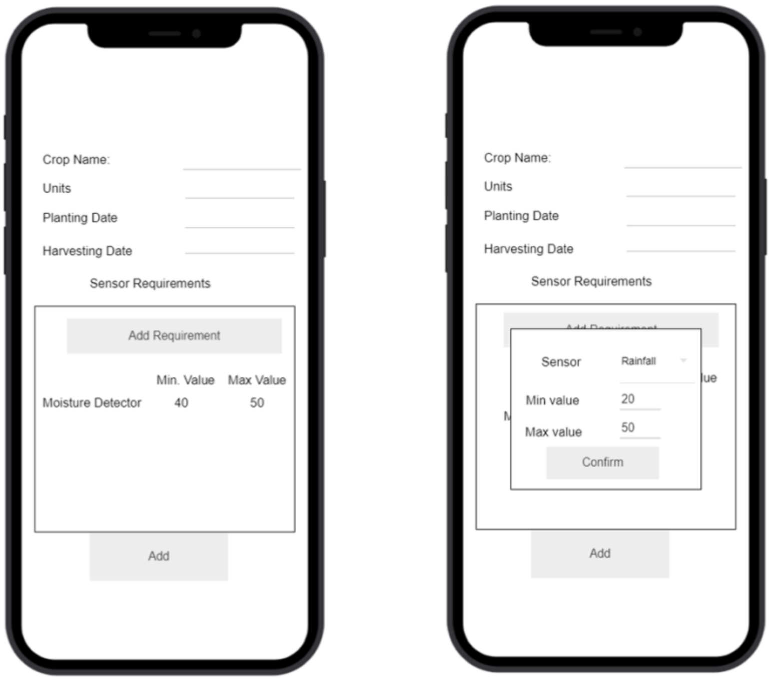
Fig. 4



	Min	Max	
Moisture Sensor	10	None	x
Soil Analyzer: Phosphorus	3	10	x

Fig. 5

When a user clicks on check on plants they come to the internal menu regarding crops. If they click on an already planted crop, for e.g., Carrot then they see information related to the crop named as carrot. Information such as Date Planted, Date of Harvest and size of plantation. If requested users can also view requirements of a crop. This request can be shown in Fig. 5. It shows values and sensory requirements set by the user. Requirements can also be added by users.



Crop Name:
Units
Planting Date
Harvesting Date
Sensor Requirements
Add Requirement

	Min. Value	Max Value
Moisture Detector	40	50

Add

Sensor: Rainfall
Min value: 20
Max value: 50
Confirm

Fig. 6

If the user decides to add a crop they land on screen as shown in Fig. 6. Users can input information like, Crop name, size of plantation, date of plantation, date of harvesting and also crop specific requirements. This includes choosing a sensor, setting range of acceptable values. If only lower limit or upper limit will suffice then user can also input ‘X’.

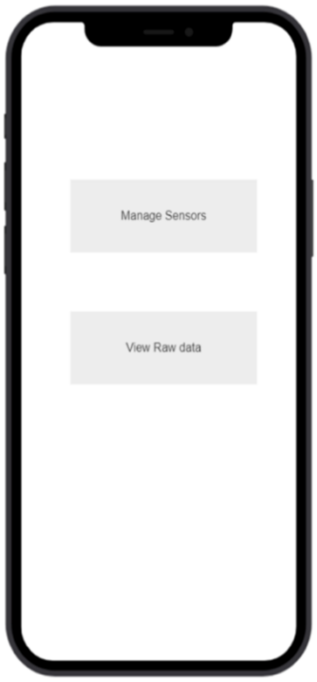
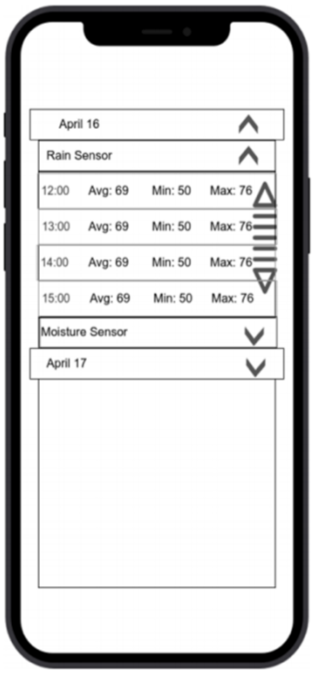


Fig. 7



April 16
Rain Sensor

12:00	Avg: 69	Min: 50	Max: 76
13:00	Avg: 69	Min: 50	Max: 76
14:00	Avg: 69	Min: 50	Max: 76
15:00	Avg: 69	Min: 50	Max: 76

Moisture Sensor
April 17

Fig. 8

If a user chooses Sensors in Fig 2. They land on the screen shown in Fig. 7. Then users can choose if they wish to manage their sensor list or view data from sensors. If a user wishes to view raw sensor data, user lands on the screen as shown in Fig. 8. It shows raw data from the sensors in a mildly processed form. User can see when the data was last processed, average value, minimum value and maximum value.

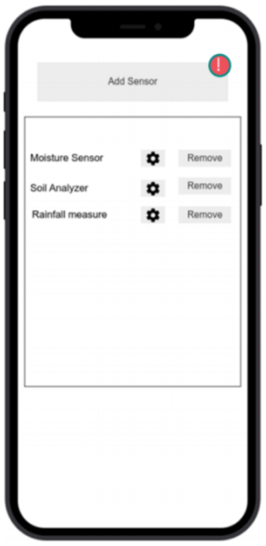


Fig. 9

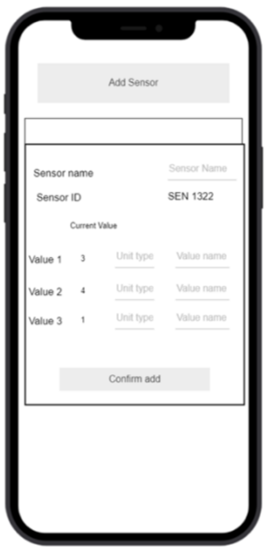


Fig. 10

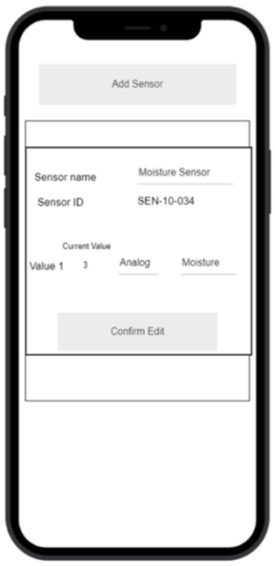


Fig. 11

If the user chooses to manage sensors in Fig. 8, they are directed to the screen as shown in Fig. 9. Here the users can see all the installed sensors or choose to add sensor. If there are any sensors not yet configured by the user but were added to the network, then there is a alerting icon as shown in Fig. 9. If the user has a sensor left to configure, they click on Add Sensor and land on screen as shown in Fig 10. User can add sensor name and other data related to the sensor. If the user chooses to edit an already installed sensor, they can do so by clicking on the gear icon next to the sensor name. As shown in fig. 11, user can edit name, value type and name.

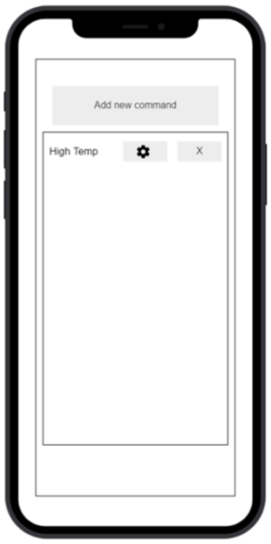


Fig. 12

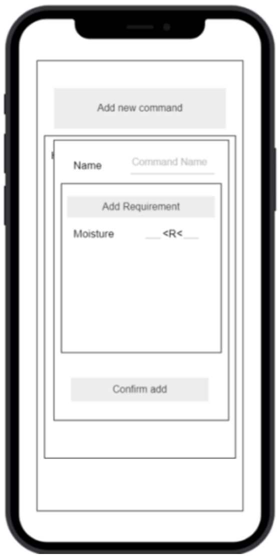


Fig. 13

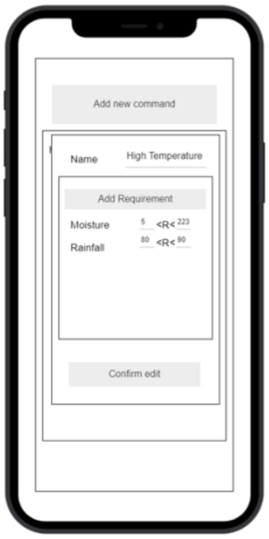
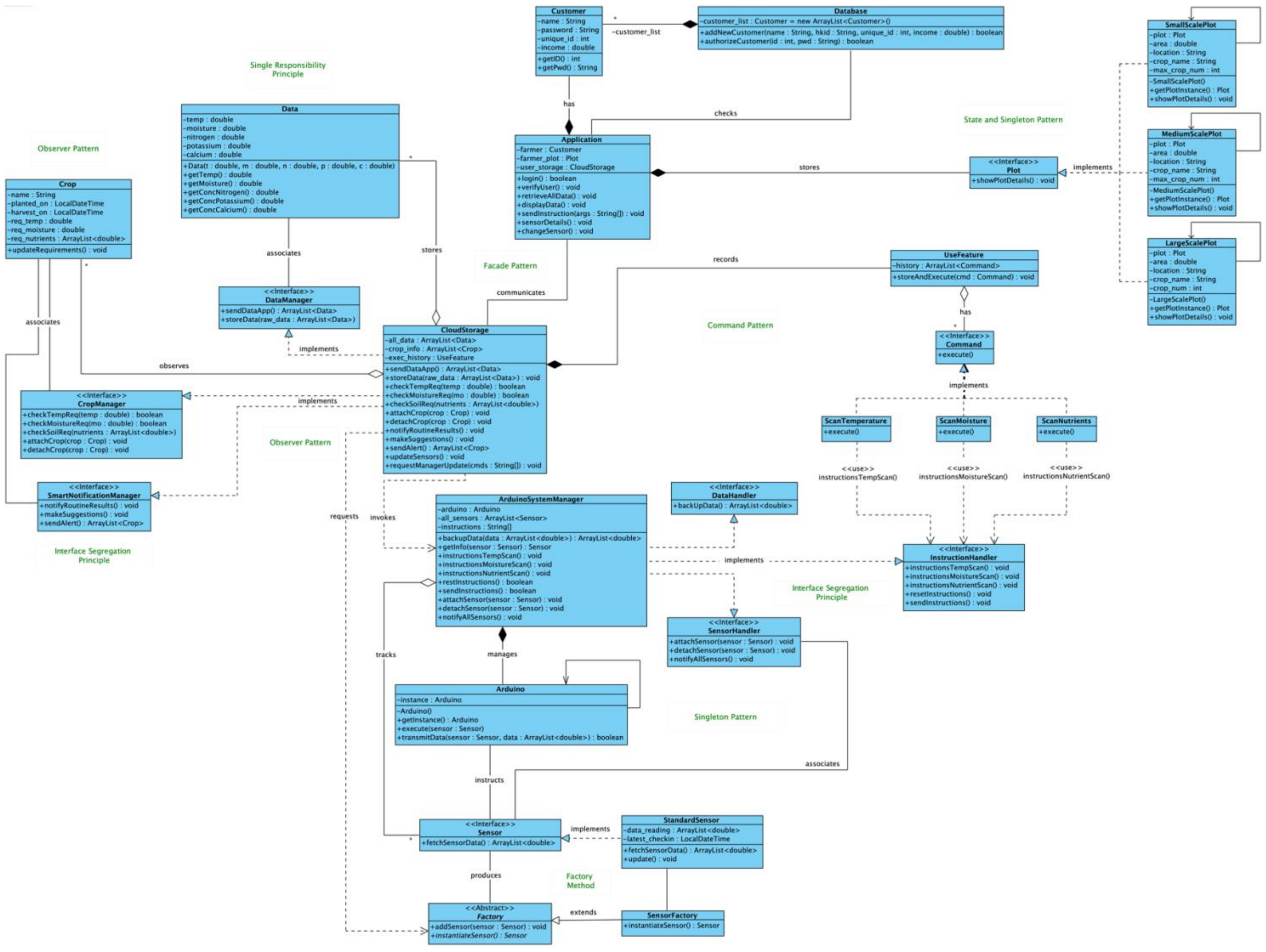


Fig. 14

If the user chooses to manage smart commands in Fig. 2, they are directed to the screen as shown in Fig. 12. Here the users can see all the smart commands in place or choose to add a smart command. If the user clicks on Add Smart Command, then they land on screen as shown in Fig 13. User can add smart command name and set a range. If the user chooses to edit an already in place smart command, they can do so by clicking on the gear icon next to the smart command name. As shown in fig. 14, user can edit name and range of values.

4. Software Design with UML

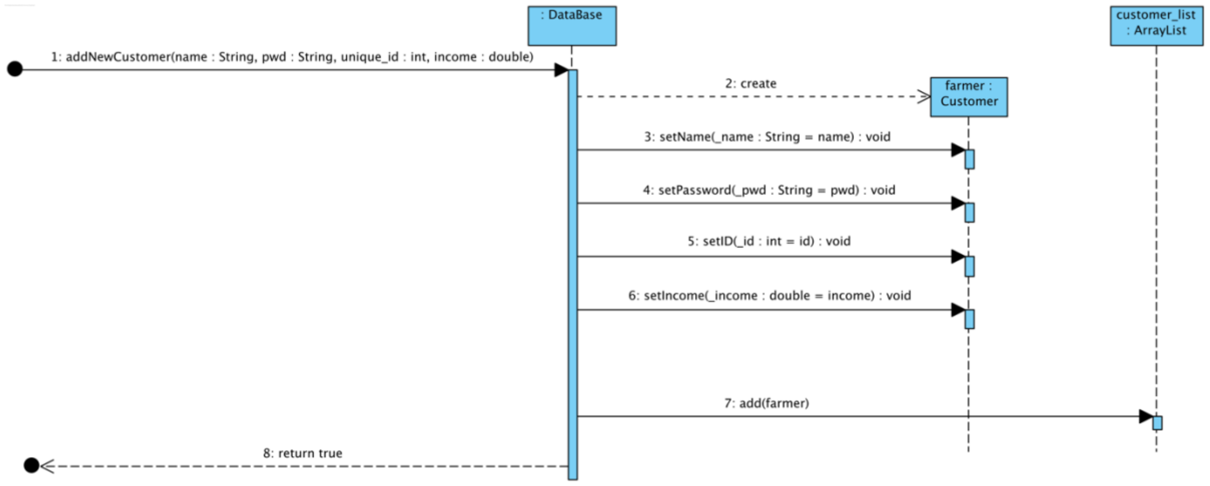
4.1 Class Diagram



Note: Please refer to section 4.3 for a detailed evaluation of the above class diagram.

4.2 Sequence Diagrams

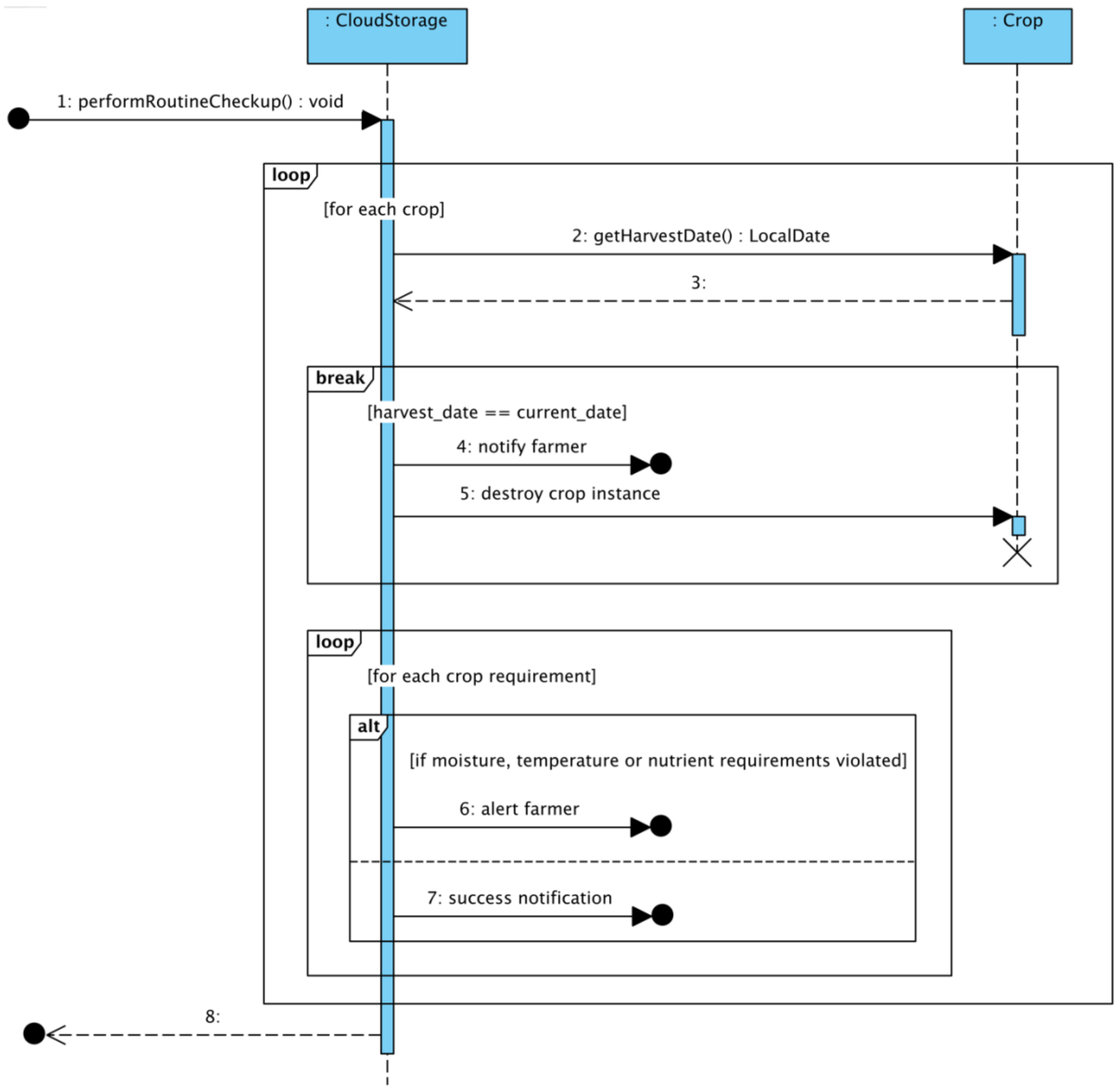
4.2.1 Register Customer


: DataBase
customer_list : ArrayList
1: addNewCustomer(name : String, pwd : String, unique_id : int, income : double)
2: create
farmer : Customer
3: setName(_name : String = name) : void
4: setPassword(_pwd : String = pwd) : void
5: setID(_id : int = id) : void
6: setIncome(_income : double = income) : void
7: add(farmer)
8: return true

The first sequence diagram shows the registration process of a new customer that wishes to use *FarmAssist* services. In the diagram above, one should be able to see a 'found message' that is coming from the application when the user wishes to register himself or herself. The message signature contains registration information, namely the user's name, password, identification, and income. The backend database then consequently creates a new instance of a class type *Customer* called farmer, as shown by the Object creation notation. Then, points 3 to 6 depict setter methods, or mutator methods if you will, that set the values in the *Customer* instance. Finally, we come to point number 7, wherein the customer object is added to the *ArrayList* of Customers and the database lifeline returns a success message as a lost message.

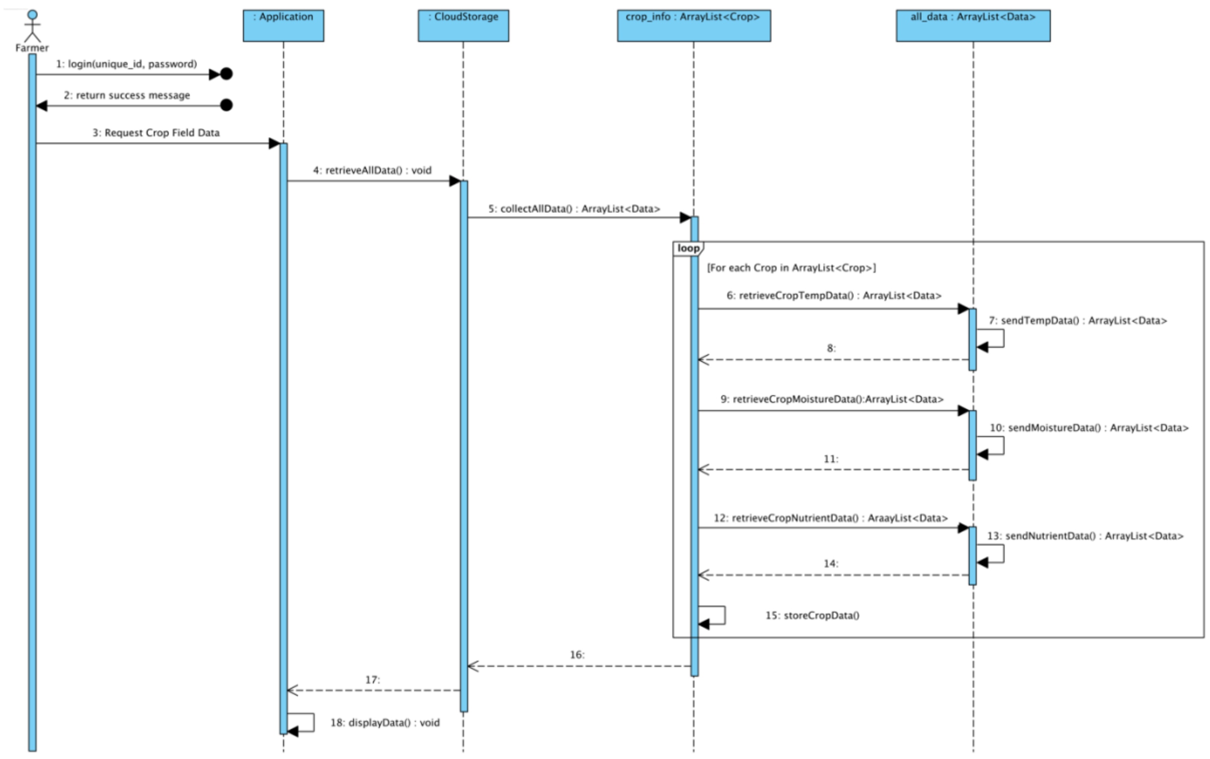
4.2.2 Smart Notification

Moving on, the second sequence diagram – as depicted in the image below – describes the smart notification that is sent to the application. It is a part of the routine checkup conducted by the cloud. The framework of this sequence diagram primarily serves the purpose of checking the information for each crop. Firstly, the *CloudStorage* lifeline retrieves the harvesting date of the crop. After the synchronous message containing the harvesting date of reference type *LocalDate* is received, the cloud then checks whether it is time to harvest the crop and if so, it notifies the farmer via the application by sending a lost message. Subsequently, it proceeds to destroy the crop instance as this project assumes that the farmer immediately harvests the crop.


: CloudStorage
: Crop
1: performRoutineCheckup() : void
loop
[for each crop]
2: getHarvestDate() : LocalDate
3:
break
[harvest_date == current_date]
4: notify farmer
5: destroy crop instance
loop
[for each crop requirement]
alt
[if moisture, temperature or nutrient requirements violated]
6: alert farmer
7: success notification
8:

On the contrary, if the system decides that it is not time for the crop to be harvested yet, the cloud checks the requirements for each crop, and notifies the farmer accordingly. The alert farmer message is invoked if the moisture, temperature of nutrient requests are violated. Such routing is performed for every *Crop* instance in the *ArrayList* of crops contained inside the *CloudStorage* class. The *CloudStorage* class is elaborated upon in the following section (4.3).

4.2.3 Data Retrieval

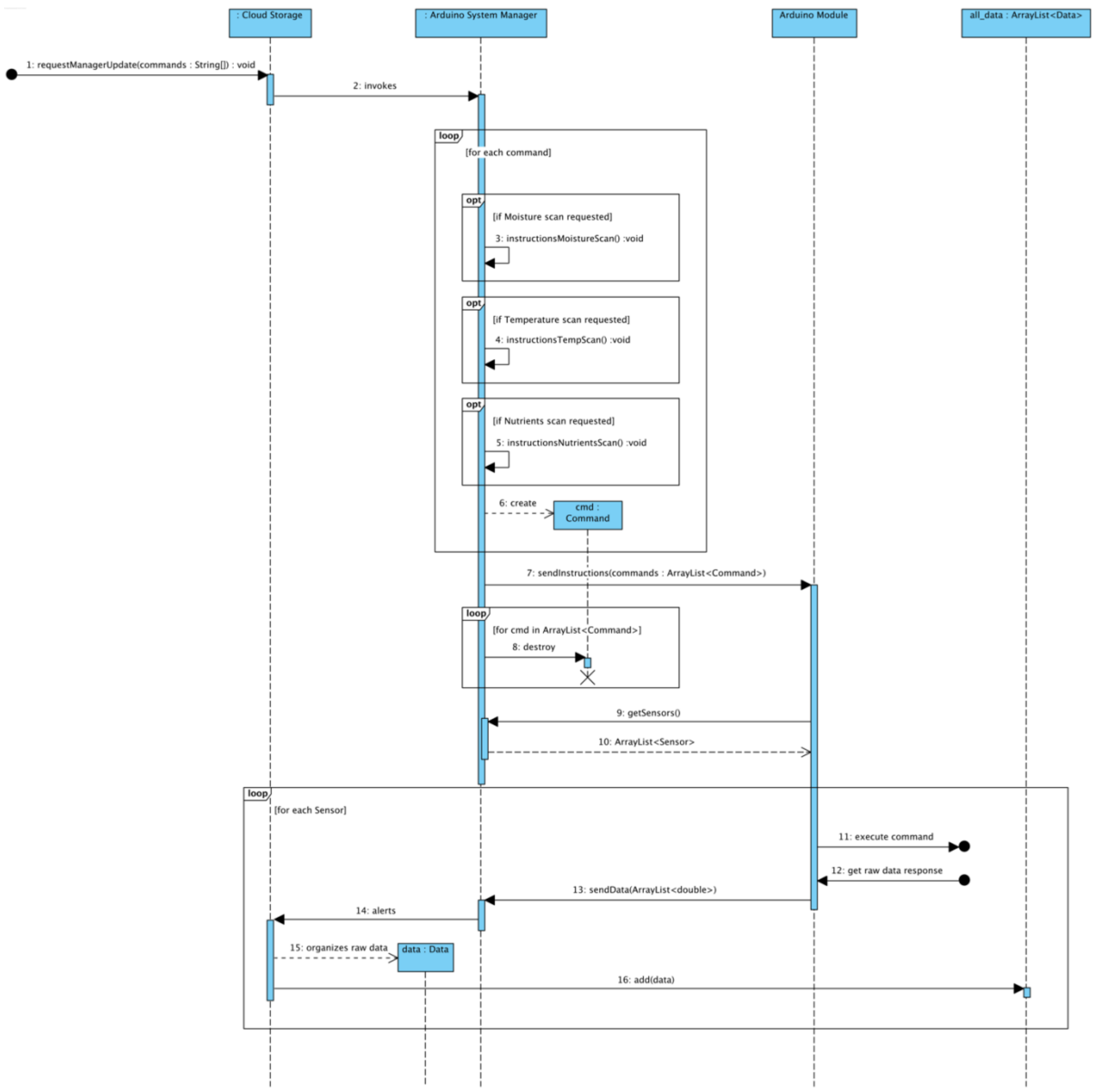

Farmer
: Application
: CloudStorage
crop_info : ArrayList<Crop>
all_data : ArrayList<Data>
1: login(unique_id, password)
2: return success message
3: Request Crop Field Data
4: retrieveAllData() : void
5: collectAllData() : ArrayList<Data>
loop
[For each Crop in ArrayList<Crop>]
6: retrieveCropTempData() : ArrayList<Data>
7: sendTempData() : ArrayList<Data>
8:
9: retrieveCropMoistureData():ArrayList<Data>
10: sendMoistureData() : ArrayList<Data>
11:
12: retrieveCropNutrientData() : AraayList<Data>
13: sendNutrientData() : ArrayList<Data>
14:
15: storeCropData()
16:
17:
18: displayData() : void

The above third sequence diagram shows the process whenever the farmer wishes to retrieve data from the system. The first two steps pertaining to the login have been waived over as lost and found messages, and the crux of this sequence diagram starts from step 3. After logging in successfully, the farmer, or the actor in this case, invokes the cloud via the app, which in turn invokes the command to collect all data. Then for every crop, the moisture, temperature, and nutrient data is received synchronously. The decision to send messages synchronously was made to ensure that the caller must wait for the information to be exchanged with the callee before proceeding further with execution.

It is also worthwhile to note that the *sendTempData*, *sendMoistureData*, and *sendNutrientData* methods occur as self-messages because their execution occurs in the same lifeline of *all_data*. After the data has been retrieved, it is simply displayed to the user in the application.

A conscious decision was made to implement this sequence diagram as a stair diagram, as opposed to the fork diagram that was also taught during the lectures. Our group decided to structure this sequence diagram by decentralizing control since there is strong dependency within the messages and moreover, the order of operations cannot be changed and there is no need to insert any new operations in the future during the Software Maintenance phase.

4.2.4 Backend Interaction

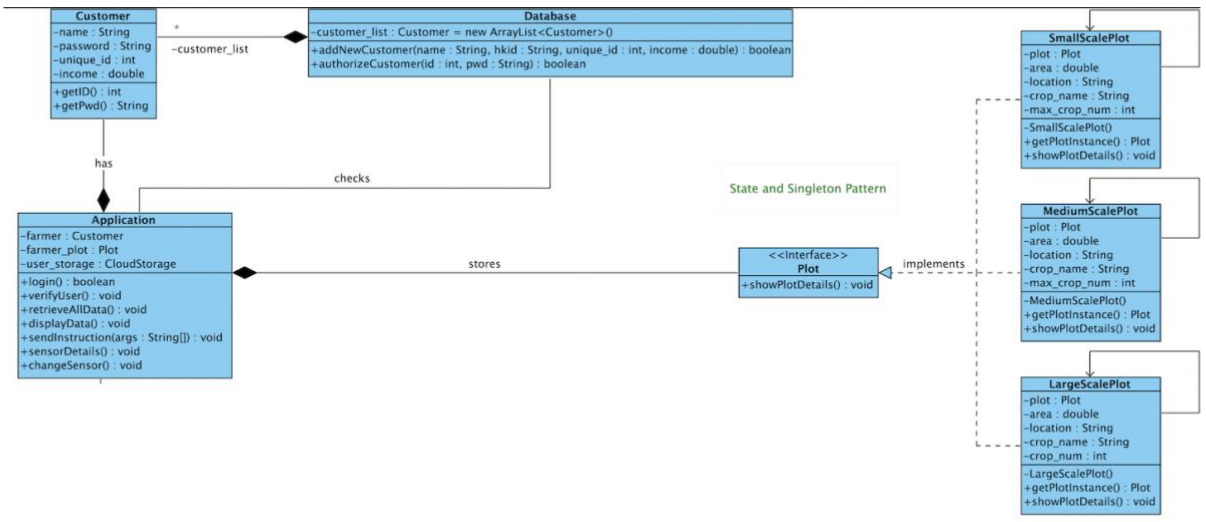

: Cloud Storage
: Arduino System Manager
Arduino Module
all_data : ArrayList<Data>
1: requestManagerUpdate(commands : String[]) : void
2: invokes
loop
[for each command]
opt
[if Moisture scan requested]
3: instructionsMoistureScan() :void
opt
[if Temperature scan requested]
4: instructionsTempScan() :void
opt
[if Nutrients scan requested]
5: instructionsNutrientsScan() :void
6: create
cmd : Command
7: sendInstructions(commands : ArrayList<Command>)
loop
[for cmd in ArrayList<Command>]
8: destroy
9: getSensors()
10: ArrayList<Sensor>
loop
[for each Sensor]
11: execute command
12: get raw data response
13: sendData(ArrayList<double>)
14: alerts
15: organizes raw data
data : Data
16: add(data)

The final sequence diagram describes the backend interaction that takes place and is obscured from the user and the application at large. To begin with, this sequence diagram iterates over the requests (an array of String objects) sent by the user and checks which of the 3 features (moisture, temperature, or nutrients) they pertain to. Then, the *ArduinoSystemManager* class creates a *Command* instance for each request and sends all the commands to the Arduino module to execute. The Arduino then requests the list of sensors that these commands need to be applied to as a recant message. The process of executing these commands and the data they send falls under the realm of hardware design and circuit programming. Since such low-level details are beyond the scope of this project and course, we simply step overlook them as lost and found messages interacting with the sensor. Finally, the raw data is passed back to the cloud, which organizes it as seen in the subsequent section of the report.

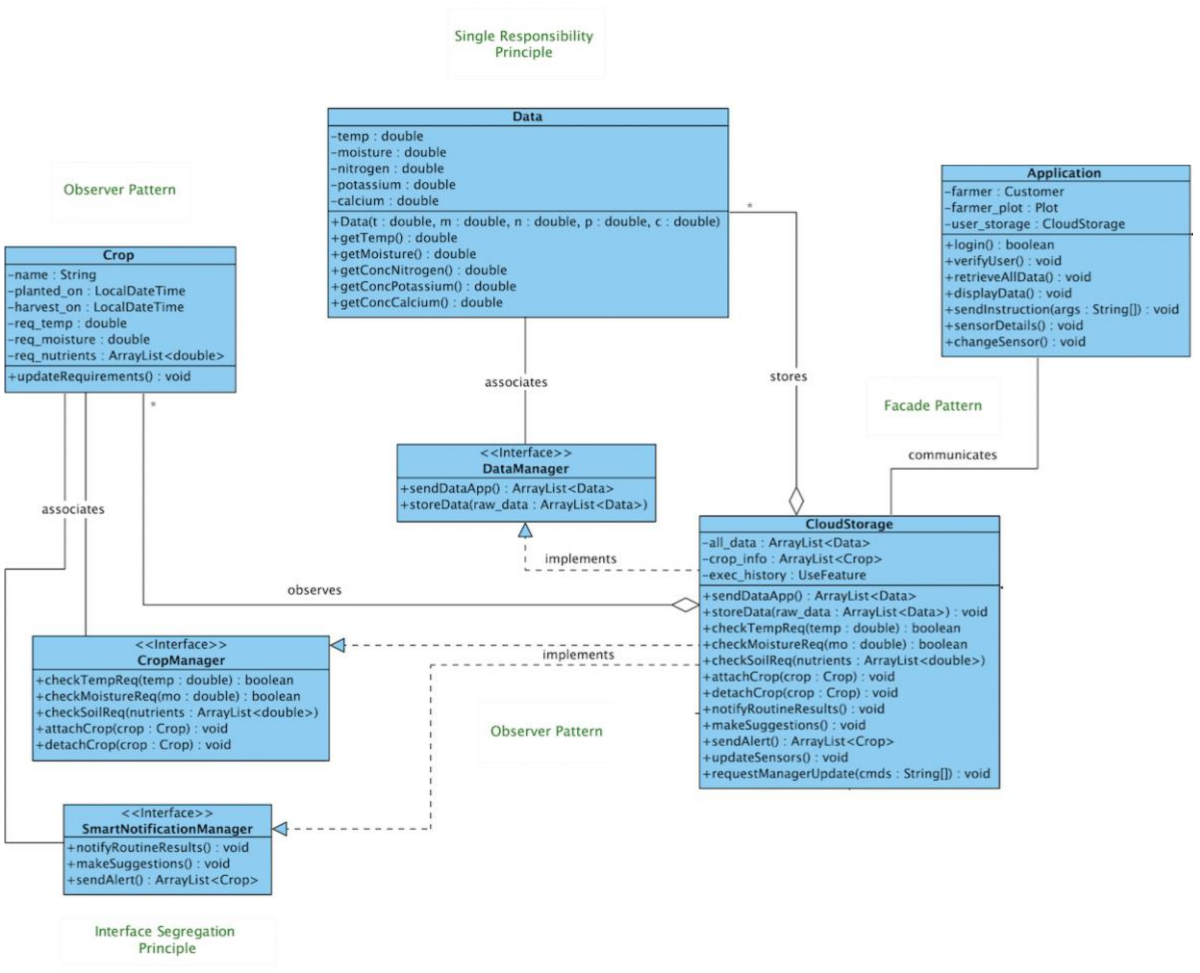
4.3 Software Design Principles, Patterns & Strategies

This section discusses the UML class diagram shown in section 4.1 for *FarmAsisst* in detail and elaborates upon the Software Design principles, patterns and strategies that were utilized while constructing the diagram. Obviously, not all design principles, patterns, and strategies thought in this course were forcefully implemented in the diagram to avoid over-engineering the solution. Nonetheless, they were all consulted and reviewed carefully to design an optimal, efficient class diagram that abides by the best practices of software design and supplements the overarching structure of this project.

Please note that the figures below have been cropped to make content more reader friendly and focused towards highlighting certain concepts. The full class diagram, in its entirety with all class links, is available in section 4.1.

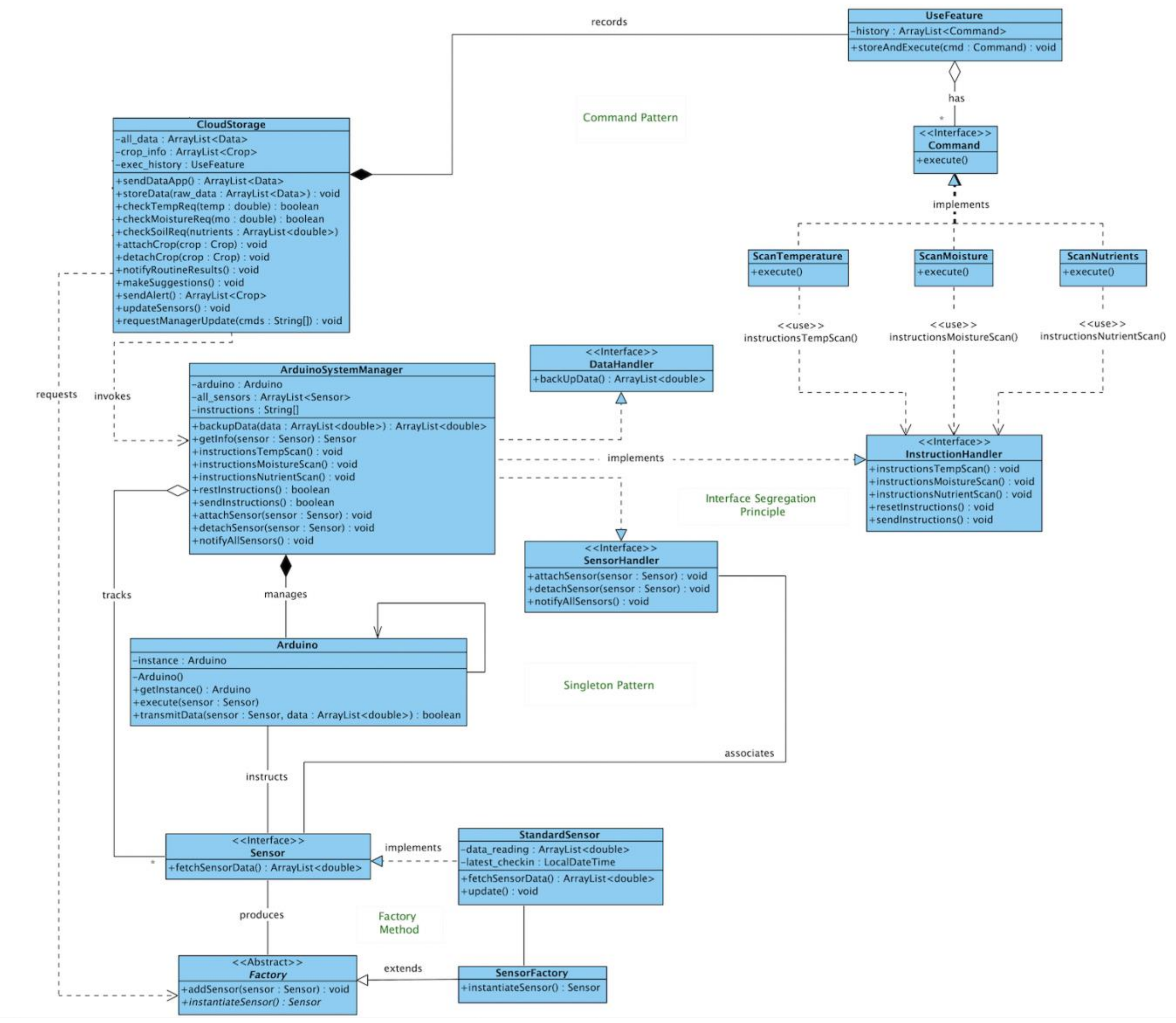


The *Application* class is quintessentially the crux of the diagram from the user's perspective. Obviously, every application is registered to a single customer. It is also linked to a database class, that is invoked for the first time during the customer registration (as seen in the first sequence diagram in section 4.2) and then every subsequent time whenever the user logs into the application after verifying themselves. The *Application* stores the plot details of each farmer as an attribute of type *Plot*. The *Plot* class is itself an interface and depending on the dimensions of the farmer's land, implemented by 3 concrete classes. These 3 classes contain their own implementation of the *showPlotDetails* method which is invoked directly by the user. The motivation to use the **State Pattern** here was to allow the user to alter the *Plot* object classification at runtime. In addition, these three sub-classes are implemented as **Singletons** because each farmer can have only 1 instance of a *Plot*.


Single Responsibility Principle
Data
-temp : double
-moisture : double
-nitrogen : double
-potassium : double
-calcium : double
+Data(t : double, m : double, n : double, p : double, c : double)
+getTemp() : double
+getMoisture() : double
+getConcNitrogen() : double
+getConcPotassium() : double
+getConcCalcium() : double
Observer Pattern
Crop
-name : String
-planted_on : LocalDateTime
-harvest_on : LocalDateTime
-req_temp : double
-req_moisture : double
-req_nutrients : ArrayList<double>
+updateRequirements() : void
Application
-farmer : Customer
-farmer_plot : Plot
-user_storage : CloudStorage
+login() : boolean
+verifyUser() : void
+retrieveAllData() : void
+displayData() : void
+sendInstruction(args : String[]) : void
+sensorDetails() : void
+changeSensor() : void
associates
stores
Facade Pattern
communicates
<<Interface>>
DataManager
+sendDataApp() : ArrayList<Data>
+storeData(raw_data : ArrayList<Data>)
associates
implements
CloudStorage
-all_data : ArrayList<Data>
-crop_info : ArrayList<Crop>
-exec_history : UseFeature
+sendDataApp() : ArrayList<Data>
+storeData(raw_data : ArrayList<Data>) : void
+checkTempReq(temp : double) : boolean
+checkMoistureReq(mo : double) : boolean
+checkSoilReq(nutrients : ArrayList<double>)
+attachCrop(crop : Crop) : void
+detachCrop(crop : Crop) : void
+notifyRoutineResults() : void
+makeSuggestions() : void
+sendAlert() : ArrayList<Crop>
+updateSensors() : void
+requestManagerUpdate(cmds : String[]) : void
observes
implements
<<Interface>>
CropManager
+checkTempReq(temp : double) : boolean
+checkMoistureReq(mo : double) : boolean
+checkSoilReq(nutrients : ArrayList<double>)
+attachCrop(crop : Crop) : void
+detachCrop(crop : Crop) : void
Observer Pattern
<<Interface>>
SmartNotificationManager
+notifyRoutineResults() : void
+makeSuggestions() : void
+sendAlert() : ArrayList<Crop>
Interface Segregation Principle

Moving on, the rest of the functionality shown in the prototype is implemented by the *CloudStorage* class shown above, which acts as the backend to the frontend side of the application class. Evidently, this design is clearly an example of the **Facade Pattern** and it was utilized to avoid overburdening the *Application* class and give farmers direct access in addition to avoiding backend complexity. As shown in the sequence diagrams, the cloud contains an *ArrayList* of crops and manages them by constantly re-evaluating their requirements whenever new data is sent in from the Arduino system. We decided to use the **Observer Pattern** here since the *CloudStorage* class needs to notify and update all its dependent crops in either of the following two cases: when it is time to update the crop requirements or record a harvesting date for the crop.

The CloudStorage manages not only the crops and their data, but it is also responsible for sending notifications back to the application for the farmer to act on. We utilized the **Interface Segregation Principle** here by grouping each set of common tasks as an interface for heuristic purposes. The three interfaces are called *CropManager*, *DataManager*, and *SmartNotificationManager* and are all implemented by the Cloud.

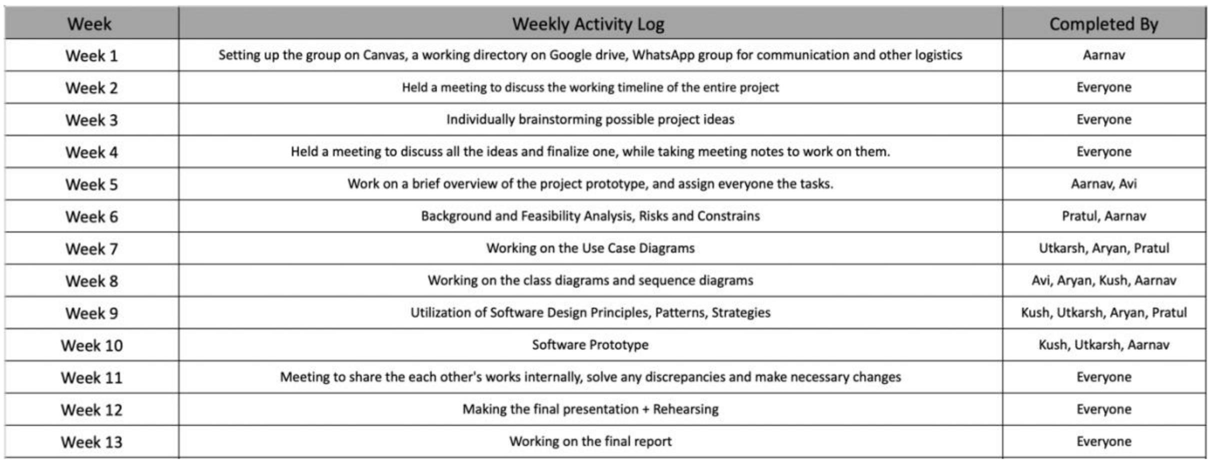

records
UseFeature
-history : ArrayList<Command>
+storeAndExecute(cmd : Command) : void
has
Command Pattern
CloudStorage
-all_data : ArrayList<Data>
-crop_info : ArrayList<Crop>
-exec_history : UseFeature
+sendDataApp() : ArrayList<Data>
+storeData(raw_data : ArrayList<Data>) : void
+checkTempReq(temp : double) : boolean
+checkMoistureReq(mo : double) : boolean
+checkSoilReq(nutrients : ArrayList<double>)
+attachCrop(crop : Crop) : void
+detachCrop(crop : Crop) : void
+notifyRoutineResults() : void
+makeSuggestions() : void
+sendAlert() : ArrayList<Crop>
+updateSensors() : void
+requestManagerUpdate(cmds : String[]) : void
<<Interface>>
Command
+execute()
implements
ScanTemperature
+execute()
ScanMoisture
+execute()
ScanNutrients
+execute()
<<use>>
instructionsTempScan()
<<use>>
instructionsMoistureScan()
<<use>>
instructionsNutrientScan()
requests
invokes
<<Interface>>
DataHandler
+backUpData() : ArrayList<double>
ArduinoSystemManager
-arduino : Arduino
-all_sensors : ArrayList<Sensor>
-instructions : String[]
+backupData(data : ArrayList<double>) : ArrayList<double>
+getInfo(sensor : Sensor) : Sensor
+instructionsTempScan() : void
+instructionsMoistureScan() : void
+instructionsNutrientScan() : void
+restInstructions() : boolean
+sendInstructions() : boolean
+attachSensor(sensor : Sensor) : void
+detachSensor(sensor : Sensor) : void
+notifyAllSensors() : void
implements
<<Interface>>
InstructionHandler
+instructionsTempScan() : void
+instructionsMoistureScan() : void
+instructionsNutrientScan() : void
+resetInstructions() : void
+sendInstructions() : void
Interface Segregation Principle
<<Interface>>
SensorHandler
+attachSensor(sensor : Sensor) : void
+detachSensor(sensor : Sensor) : void
+notifyAllSensors() : void
tracks
manages
Arduino
-instance : Arduino
-Arduino()
+getInstance() : Arduino
+execute(sensor : Sensor)
+transmitData(sensor : Sensor, data : ArrayList<double>) : boolean
Singleton Pattern
associates
instructs
<<Interface>>
Sensor
+fetchSensorData() : ArrayList<double>
implements
StandardSensor
-data_reading : ArrayList<double>
-latest_checkin : LocalDateTime
+fetchSensorData() : ArrayList<double>
+update() : void
produces
Factory Method
<<Abstract>>
Factory
+addSensor(sensor : Sensor) : void
+instantiateSensor() : Sensor
extends
SensorFactory
+instantiateSensor() : Sensor

Furthermore, the *Arduino* is implemented as a **Singleton** as only 1 instance of the class can exist during execution. Since the Arduino is a hardware system, it cannot directly talk to the cloud as its functions are limited to executing commands and sending data. Therefore, we designed a manager class for it called *ArduinoSystemManager* that uses **Integration Segregation Principle** to categorize and handle tasks. It is worthwhile to note that the instruction handler is a part of an overarching **Command Pattern** that allows farmer to issue commands at ease via the *FarmAssist* app and encapsulate method calls in objects. The three types of scans that a sensor can perform are for: temperature, moisture, and nutrients. The history of these commands is stored in the *UseFeature* class, which has a composition relationship with the *CloudStorage*. Each of these 3 commands, moreover, require their own set of programmed, low level instructions that are compiled by the *ArduinoSystemManager* class and passed onto the Arduino module. At the same time, the *ArduinoSystemManager* also notifies all the sensors to make the update and thus the **Observer Pattern** is prevalent here as well given the one-to-many dependency between the *ArduinoSystemManager* and the *ArrayList* of all sensors.

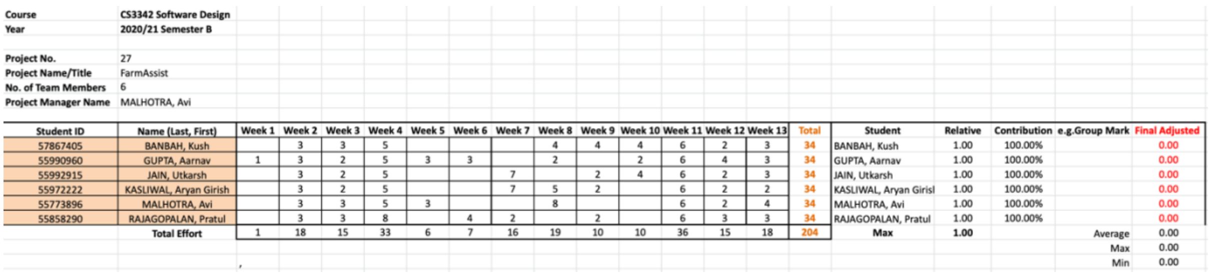
The Arduino then simply collects the raw data from each sensor and transmits it to the *ArduinoSystemManager* class, which then further passes it along to the Cloud. It is here that the raw data is encapsulated and organized. This is done by creating an instance of the Data class to store the array of values. Suffice it to say, the Data class adheres to the standards of the **Single Responsibility Principle** as it is intentionally designed to store instances of raw data.

Lastly, if the user wishes to link more sensors to the software, the cloud requests the *SensorFactory* to produce a new instance of the *StandardSensor* and then proceeds to link it. It is important to note that the relationship type here is of a dependency and not an association because the *Factory* simply returns a newly created instance of an object. The **Factory Method** proved to be an incredibly useful design pattern as the creation of the sensor object was needed to be determined at runtime. Finally, it is worth highlighting that the entire UML class diagram adheres to the standards of the ever-important **Open-Close Principle**. All object data is private, and it may only be accessed to getter or accessor methods by other classes. Moreover, all classes are also open for extension as deemed appropriately during the software maintenance of this project in the long term. For instance, a Software Engineer will easily be able add a new command that implements the *Command* interface or link a new kind of sensor (perhaps called *PremiumSensor*) to the Cloud.

5. Reflection



Week	Weekly Activity Log	Completed By
Week 1	Setting up the group on Canvas, a working directory on Google drive, WhatsApp group for communication and other logistics	Aarnav
Week 2	Held a meeting to discuss the working timeline of the entire project	Everyone
Week 3	Individually brainstorming possible project ideas	Everyone
Week 4	Held a meeting to discuss all the ideas and finalize one, while taking meeting notes to work on them.	Everyone
Week 5	Work on a brief overview of the project prototype, and assign everyone the tasks.	Aarnav, Avi
Week 6	Background and Feasibility Analysis, Risks and Constrains	Pratul, Aarnav
Week 7	Working on the Use Case Diagrams	Utkarsh, Aryan, Pratul
Week 8	Working on the class diagrams and sequence diagrams	Avi, Aryan, Kush, Aarnav
Week 9	Utilization of Software Design Principles, Patterns, Strategies	Kush, Utkarsh, Aryan, Pratul
Week 10	Software Prototype	Kush, Utkarsh, Aarnav
Week 11	Meeting to share the each other's works internally, solve any discrepancies and make necessary changes	Everyone
Week 12	Making the final presentation + Rehearsing	Everyone
Week 13	Working on the final report	Everyone



Course	CS3342 Software Design
Year	2020/21 Semester B
Project No.	27
Project Name/Title	FarmAssist
No. of Team Members	6
Project Manager Name	MALHOTRA, Avi

Student ID	Name (Last, First)	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Total
57867405	BANBAH, Kush		3	3	5				4	4	4	6	2	3	34
55990960	GUPTA, Aarnav	1	3	2	5	3	3		2		2	6	4	3	34
55992915	JAIN, Utkarsh		3	2	5			7		2	4	6	2	3	34
55972222	KASLIWAL, Aryan Girish		3	2	5			7	5	2		6	2	2	34
55773896	MALHOTRA, Avi		3	3	5	3			8			6	2	4	34
55858290	RAJAGOPALAN, Pratul		3	3	8		4	2		2		6	3	3	34
	Total Effort	1	18	15	33	6	7	16	19	10	10	36	15	18	204

Student	Relative	Contribution	e.g.Group Mark	Final Adjusted
BANBAH, Kush	1.00	100.00%		0.00
GUPTA, Aarnav	1.00	100.00%		0.00
JAIN, Utkarsh	1.00	100.00%		0.00
KASLIWAL, Aryan Girisl	1.00	100.00%		0.00
MALHOTRA, Avi	1.00	100.00%		0.00
RAJAGOPALAN, Pratul	1.00	100.00%		0.00
Max	1.00		Average	0.00
			Max	0.00
			Min	0.00

The biggest challenge we faced as a group was that our collaboration and communication was limited to the online tools available. We were in two different time zones which, too, added up in making the discussions a bit of a hurdle. All the meetings we had were through *Zoom* and working together on *Visual Paradigm* also became a bit difficult, as we had to constantly share screen, and only one person could work on it at a time.

With the willingness and punctuality of all the members involved, and a previously planned timeline, the planned work could be finished as per schedule. The group had great harmony and understanding among themselves, making it easier for another person to add on what the person before them did.

6. Conclusion

We were able to achieve all our goals, as planned. Although, given more time and resources, we would like to implement the functionality for the app to make its own decisions, using Machine Learning algorithms, and to be able to carry them out on its own my controlling smart tools. With that functionality, FarmAssist will be able to expand to even the high-scale farmers. Although, we are absolutely thrilled by what the app has the capability to perform as of now, and how it helps the middle scale farmer in making more informed decisions.

To summarize, the mobile app will be connected to a cloud, and so will be multiple sensors. These sensors will periodically measure the information they're designated for, such as the humidity in soil or the temperature etc. and will upload it to the cloud. The user can then see the live updates on his mobile app. If needed, he can request for the current report too. The app allows the user to add new crops, new sensors, and a desirable value for each of the attributes, so the user can get a notification in-case the sensor reports a value too high or too low.

7. References

- First step towards a bright future. (n.d.). Retrieved February 01, 2021, from http://www.isaf-forum.com/related-news-details.html#:~:text=The%20goal%20of%20smart%20agriculture,support%20system%20for%20farm%20management.&text=Smart%20farming%20deems%20it%20necessary,crops%20to%20health%20and%20harvesting.

- Smart Farming Technology - Advanced Agriculture Solution | Cropin. (2019, August 22). CropIn. https://www.cropin.com/smart-farming/

- Why Smart Agriculture is the need of the hour. (2018, September 28). Inter-Regional Smart Agriculture Forum. http://www.isaf-forum.com/related-news-details.html#:%7E:text=The%20goal%20of%20smart%20agriculture,support%20system%20for%20farm%20management.&text=Smart%20farming%20deems%20it%20necessary,crops%20to%20health%20and%20harvesting.

- Visual Paradigm (16.2). (2020). [Software]. Visual Paradigm International Ltd. https://www.visual-paradigm.com