# RAPID DEVELOPMENT OF MOBILE APPS USING APP INVENTOR AND AGCO API

by

SPENCER KEPLEY

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Biological and Agricultural Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2014

Approved by:

Major Professor
Naiqian Zhang

# Copyright

SPENCER KEPLEY

2014

# Abstract

Mobile apps are useful tools for many different purposes. In agriculture, apps can be used to check the weather and markets, control irrigation, and monitor machine activity among other uses. This research project is a collaboration between Kansas State University and AGCO and includes the development of two apps, using MIT Application Inventor and Google App Engine. Kansas State University was responsible for developing the apps user interface and functionality while AGCO provide the data needs for the apps through Google App Engine. The first app is called Crop Maturity App and measures Growing Degree Days from a crops planting date. The second app is called Combine Efficiency App and determines the performance of a combine harvesting based on its speed. AGCO provided the server support for these apps from a weather service and their own combines that are connected. This project demonstrates the possibility of an open-source development environment with AGCO machine data.

# Table of Contents

List of Figures ........ vi
List of Tables ........ ix
Acknowledgements ........ x
Chapter 1 - Introduction ........ 1
Chapter 2 - Literature Review ........ 3
  Developing Apps ........ 3
  Potential in Agriculture ........ 5
Chapter 3 - Crop Maturity App ........ 9
  Introduction ........ 9
  Use Case ........ 10
  User Interface ........ 11
  Model ........ 12
  AGCO Weather API ........ 12
  Blocks Editor ........ 15
  Results ........ 20
Chapter 4 - Combine Efficiency App ........ 22
  Introduction ........ 22
  Use Case ........ 22

  Model ........ 25
  API ........ 26
  Blocks Editor ........ 28
    Initialization ........ 28
    Registration and Logging In ........ 34
    Requesting, Receiving, and Handling Machine Data ........ 41
    Changing Fuel and Crop Losses ........ 51
    Mashing Machine Data with Model ........ 60
  Results ........ 64

Chapter 5 - Conclusions ........................................................ 65
Chapter 6 - Future Work Possibilities ........................................................ 66
Appendix A - Combine Efficiency Spreadsheet ........................................................ 69
Appendix B - Template App Sources ........................................................ 70

# List of Figures

Figure 3.1. Home page of Crop Maturity App, displaying a ListPicker (Pick Field), field name label (Field Label), labels for the field location, and labels for the date the crop was planted. ........ 10

Figure 3.2. (a) Display after ListPicker has been clicked. <Add Field> is the only choice initially. (b)Add Field page. (c)Home page with a field selected. ........ 11

Figure 3.3. Procedure for making a request to the AGCO server. ........ 15

Figure 3.4. Event block that occurs when the app has received a response from the AGCO server. ........ 16

Figure 3.5. The parsing procedure places the response in the IncomingValuesList and removes the IncomingTag. ........ 17

Figure 3.6. Procedure to fill column data, perform calculations, and display information. ........ 18

Figure 3.7. Procedures for storing column data. ........ 19

Figure 3.8. Blocks performing GDD calculations. ........ 20

Figure 4.1. The “Login View” contains nine entries: three for user information, three for account information, and three for time factors that can be left alone. ........ 23

Figure 4.2. Views from the Home and Machine sections of the Combine Performance App. Views a, b, and c show the “Home View” while d, e, and f show the “Machine View.” ........ 24

Figure 4.3. Speed vs. Operating expenses for a combine harvester ........ 25

Figure 4.4. Event block for the app initialization. ........ 29

Figure 4.5. Procedure block that is called upon the app initialization and registration. ........ 31

Figure 4.6. Procedure block controlling the view displayed to the user. Three argument entries are possible in this figure to return a true. These are “register_retry, register, and login.” ........ 32

Figure 4.7. Continuation of the “WindowToDisplay” block. Argument entries include “expenses, and configLosses.” ........ 33

Figure 4.8. Final part of “WindowToDisplay” block. “MachineStatus” is an argument entry. ........ 34

Figure 4.9. Event block occurring with the click of the register button. ........ 34

Figure 4.10. Procedure block executed during registration. This procedure checks the time values and stores the textbox entries on the “Login View” to a UserAccount. ........ 35

Figure 4.11. Procedure block that Posts request to Hesston server for a key. ............................ 36
Figure 4.12. Event block that handles web responses. ............................................................ 38
Figure 4.13. Procedure block called in the web handler that handles code parsing. .................. 39
Figure 4.14. Procedure block that stores the register key received by the Hesston server response. .................................................................................................................... 40
Figure 4.15. Procedure block that requests the Hesston server to sync machine data for a fleet. 41
Figure 4.16. Procedure block that posts request to Hesston server for stored machine data for a fleet. ......................................................................................................................... 42
Figure 4.17. Procedure block that handles response from Hesston server that confirms the success or failure of the request to sync machine data for a fleet. ...................................... 43
Figure 4.18. Procedure block handling the Hesston server's response containing machine data for a fleet. ................................................................................................................ 46
Figure 4.19. Procedure block that clears the fleet lists for machine data. ................................. 47
Figure 4.20. Procedure block that loads machine data into lists. The data for a machine has an index and is organized accordingly in the lists. .................................................................. 48
Figure 4.21. Continuation of block loading machine data into lists. ......................................... 49
Figure 4.22. Continuation of block loading machine data into lists. ......................................... 50
Figure 4.23. Continuation of block loading machine data into lists. ......................................... 51
Figure 4.24. Clicking the "Adjust Losses" button in the Home View allows the user to change values for the Crop Losses and Fuel Consumption. .......................................................... 52
Figure 4.25. Event block occurring when the button labeled "Adjust Losses" is clicked in the Home View. This allows the user to change values for the Crop Losses and Fuel Consumption. ........................................................................................................... 52
Figure 4.26. Event blocks occurring when the textbox has the focus or is selected. This event clears the textbox contents. .................................................................................. 53
Figure 4.27. Event blocks occurring when the textbox has lost focus. ...................................... 54
Figure 4.28. Event blocks occurring when the "Refresh" button is clicked for either the Crop Loss list or the Fuel Consumption list. ............................................................................ 54
Figure 4.29. Procedure blocks that check for a valid entries in the Crop Loss and the Fuel Consumption lists. ........................................................................................................ 55
Figure 4.30. Procedure blocks that update the Crop Loss and Fuel Consumption lists. ............. 56

Figure 4.31. Procedure blocks that update the textbox text for the Crop Loss and Fuel Consumption entries. ........................................................................................... 59
Figure 4.32. Event block occurring after a machine has been picked. ....................................... 60
Figure 4.33. Event block occurring at the click of the Solver button in the Machine View. ...... 60
Figure 4.34. Procedure block that updates lists for the model.................................................... 61
Figure 4.35. Procedure block that determines the optimal speed and cost. ................................ 62
Figure 4.37. Procedure block that sets the Home View labels based on the machine status....... 63

# List of Tables


Table 3.1 AGCO Weather API call to retrieve high and low temperatures for a period of days. 13

Table 3.2 AGCO Weather API response to Crop Maturity App request. .................................. 14

Table 4.1. HTTP Post Request for Registration Key.................................................................. 26

Table 4.2. HTTP Post Request for Stored Data. ......................................................................... 27

Table 4.3. HTTP Post Request to Sync Machine Data………………………………………….28

Table 4.4. AGCO Machine Data ................................................................................................ 44

# Acknowledgements

Many people were influential in the completion of this work. I would like to thank Dr. Naiqian Zhang for his advice and persistence, Chris "Buzz" O'Neil at AGCO Corporation for his partnership on the apps development, and my family and friends for their support. I would also like to acknowledge the INSIGHT program at Kansas State University for funding my graduate education.

# Chapter 1 - Introduction

Mobile applications (apps) are popular among users of smartphones and tablets. Mobile users interact with apps through a mobile device's tactile, audio, and visual inputs and outputs. Apps can integrate data from embedded or external sensors, the Internet, or any of the functions native to smart devices and have strong computation power. Due to their small size and light energy consumption, mobile devices are portable tools and are finding uses out of the office.

This research explores developing apps for agricultural purposes using the MIT Application Inventor (AI) integrated development environment (IDE) and data provided by the Hesston application program interface (API).

AI is an IDE still in its beta stage and is designed to allow people without development experience to develop apps for Android devices. It has been taught as in introductory computer information courses at the University of San Francisco, Harvard, and Massachusetts Institute of Technology as well in other places. Developing happens in three windows. The first window is on a browser and allows the developer to select and arrange components on the apps screen. The second window is a Java window where the developer can connect blocks with the purpose of controlling the apps features. The third window is an emulator or a device that allows real-time testing and debugging.

AGCOMMAND is the telemetry system provided by AGCO. This system records data from tractors with AGCOMMAND and sends the data to an AGCO server. The server can further transmit the tractor data to a computer or device with Internet access. The Hesston API is a noncommercial project set up by AGCO employees that communicates with AGCOMMAND to provide tractor data to an app making requests. It is hosted on Google's servers through Google App Engine.

The market penetration of mobile devices is widespread. Smartphones and tablets are taking computing out to remote locations and connecting their users with vast amounts of information. They are highly functional with their complex sensors, Internet connection, portability, and

applications (apps). The market penetration, portability, and high functionality contribute to their overall usefulness.

Android is a mobile operating system that has a large share of the mobile market. It is open-sourced and very easy to develop apps for this platform with App Inventor (AI). AI is an integrated development environment (IDE) that is designed for beginner programmers that are developing apps for Android mobile devices. AI has code in the form of blocks that snap together, similar to puzzle pieces, in order to alleviate programming issues with syntax and correct spelling.

# Chapter 2 - Literature Review

## Developing Apps

Smartphone apps have several different operating systems that they can be included on. Apple iOS, Google Android, RIM BlackBerry 10, and Microsoft Windows Phone are just a few versions available for smartphone users, a group that makes up 42% of mobile phone users (Smith, 2011). Of this group, Android and iOS are the frontrunners. 900 million devices use Android as the operating system with more than 48 billion apps downloaded (Panzarino, 2013). IOS is on 600 million devices and 50 billion apps have been downloaded by Apple devices (Ingraham, 2013). Market share for Android is around 51% while iOS is around 41% (Jones, 2013). Developing apps for both of these platforms would reach a large majority of the population with smartphones or tablets. The number of apps available for each operating system is an indicator of the strength of the ecosystem. Android has 1,000,000 apps, Apple has 900,000 apps, Windows has 145,000 apps, and Blackberry has 120,000 apps.

Apps have marketplaces suited for specific operating systems. These are Apple App Store, Google Play Store, Windows Store, and Blackberry Store. A large difference in the platforms is the rules to publishing an app on a marketplace. Android allows developers to publish apps with little interference. A Google account and a one-time $25 registration fee to validate a Developer Account are the only requirements to distribute apps on the Play Store (Google, 2013). Apple requires a $99/year enrollment fee to even test the app on an actual device and also reviews apps prior to distribution on the App Store (Apple, 2013). Both of these app markets charge a 30% transaction fee for paid apps and in-app purchases (Transaction Fees, 2012 and iOS Developer Programs, 2013). Windows has a $99 yearly registration fee except for DreamSpark students who are given a free registration.

All of the operating system providers have toolkits for developers to use when developing apps for their devices. Developing with these toolkits is specific to each operating system and not compatible with others and is a process known as "native development". Android has several methods to deploy its Software Development Kit (SDK) while Apple has one method for deployment.

When developing with Android SDK, a developer must decide on a development environment. The options listed by Android Developers are the Android Developer Tools (ADT) Bundle, Android Studio, or an existing IDE customized with the Android SDK (Android 2013). The ADT Bundle includes a version of the Eclipse IDE with built-in Android SDK and is the preferred environment. Android Studio is the newest environment promoted by Android. It uses the IntelliJ IDEA environment, a simpler Java environment than Eclipse, includes built-in Android SDK tools, and will be the "first dedicated IDE" for Android. However, it is incomplete at the time of this research and is only available in "early access preview." Customizing an existing IDE can be done with Eclipse Helios or greater, Java Development Kit, and Apache Ant. The steps to set up an environment by this method are more extensive.

For Android this means developing with the Android Software Development Kit (SDK), programming in the Java language and usually in Eclipse IDE. Native development for iOS this means using Xcode IDE, the iOS SDK and the Objective-C language. Apps developed natively operate quick and smoothly as they are purposed specifically for an operating system. However, developing natively and trying to reach all potential customers is very time consuming.

With all of the platform choices available to consumers, app developers have a difficult time reaching everyone. Developing for the two largest operating systems are essential for reaching the majority of potential customers but with the many other available operating systems it becomes extremely difficult and costly to reach all markets by using native development tools. A solution to this problem is to develop using "cross-platform development." This development technique reaches customers with less development time, but it does not always have all the functions that are native to the device and can be slower or only work when an Internet connection is available.

Android has an additional method to develop apps. MIT Application Inventor (AI) started as a project under the supervision of Hal Abelsen while on sabbatical at Google, and was given to MIT to be maintained as an educational tool to developing apps. It is currently in beta stage and has been used in classrooms of all ages. Instead of using written code that is typical for other

developing techniques, AI uses blocks that snap together like puzzle pieces. This eliminates the need for developers to memorize code and focus on how the app will interact with the user. Designing the user interface with AI is as simple as dragging a component onto a design screen. This style of designing is known as "what you see is what you get" (WYSIWYG). AI also enables developers to test their app while programming with an emulator or an Android device instead of waiting to compile. While not containing the complete Android SDK, future plans are to make block code convertible to Java and thereby a complete Android developing toolkit. Apps developed with AI can still be put on the Play Store, however, a gallery for AI programs also exists where people can download the source code as well as the actual app. This is a unique gallery and could become the first marketplace where an app user could purchase an app and then modify the source code to make it better. With the simplicity of the block coding, most people should be capable of doing this.

Developers are using application program interfaces (APIs) to incorporate data from online sources into their apps. These apps are known as "mashups" or "Web 2.0" and are creating a new wave of useful information from data that is available but purposed for other reasons. APIs exist for Google Maps, NOAA, Facebook, and Yahoo! Market. An API that is mashed into multiple apps is Google Maps. It is extremely useful when coupled with smartphone GPS capabilities.

## Potential in Agriculture

Potential uses for smartphones in agriculture in North America are just beginning to be explored. However, mobile phone use in agriculture or mAgriculture has been studied extensively as a means of increasing income for small farmers in Africa. The only explanation for why this is the case is that farmers in North America have access to other methods of communication and information whereas farmers in Africa may only have access to a mobile phone for computation power. Vodafone has cited a potential $138 billion lift to farmers incomes through mobile services. This study cited twelve solutions to benefit farmers that were categorized four ways. These four categories are improving access to financial services, provision of agricultural information, improving data visibility for supply chain efficiency, and enhancing access to markets.

Mobile banking is possible through texting, a website, or an app. Security is an important concern with mobile banking. Security concerns compete with the convenience for accessibility to financial services. The Bank of America Mobile Banking App provides the following financial services depending on device: ability to look up account details and transaction history, make transfers, pay bills, get alerts and BankAmeriDeals™, deposit a check (using the device's camera) and find the nearest Bank of America ATM and banking center.

Apps can provide other financial services. Mint is an app that organizes the user's finances from banking, 401k, loans, credit cards, and other sources. Insurance apps enable users to file a claim. Market apps put stock information in user's hands. Using near field communication (NFC), Google Wallet is an app that allows phones to purchase goods at certain retailers. Payments can be made to friends with Venmo. Square allows a device to accept credit card payments with an attachment card reader. Financial services will continue to expand as consumers gain confidence in their security.

An important source of agricultural information comes from extension services. Extension professionals connect farmers with agricultural research and knowledge. An app, entitled Machinery Sizing, was developed to demonstrate the feasibility of extension reaching clientele through mobile devices. This app development was converted from a spreadsheet and is based from the ASABE Standard D497.4 that estimates the tractor horsepower needed to pull different implements under different conditions. The developers cited benefits for extension professionals to include portability, ease for computations, and automatic updates.

Improving data visibility depends on systems of sensors, networks, displays, and processors. Mobile phones are strongly adapted to the needs of data visibility with their portable screens, strong networks, and computation power. Sensor data from other sources can be sent on phone networks, processed, and displayed on an app or a website. Telemetry and machine-to-mobile communication (M2M) are the terms used for the purpose above. Precision agriculture has made M2M a very achievable means of improving data visibility. Dairy and field crops can be operated nearly autonomously with precision agriculture endeavors.

Lely is an agricultural company with a robotic milking system. This system permits cows to be milked when they desire throughout the day. Equipped with sensors, the Lely T4C is able to detect: milk color, fat/protein indication of the milk, lactose indication of the milk, conductivity of the milk, milk temperature, rumination minutes of the cow, cow activity, cow weight, milk production of the cow, feed intake of the cow, amount of rest feed of the cow, milking time/dead milking time, and milking speed. Lely does not have an app for monitoring the aforementioned traits but does have an app for controlling its robotic feeder. The Lely Vector Control app allows a user to set up a feeding plan, routes for the robot, and allows the user to analyze the feeding data (Lely, 2013).

Tractor data apps are available through OEMs John Deere with JDLink and AGCO with AGCOMMAND. The JDLink app is available on Apple and Android devices. The AGCOMMAND app is only available with Apple devices but a website is also available. Tractor data is tagged with GPS coordinates, timestamps, and includes information relevant to the machine and operation. Every sixty seconds AgCommand™ collects machine performance data and GPS location, which is transmitted via the GSM network and is then viewable via your computer or phone with the AgCommand™ app. Operation information can be used to make prescriptions for fertilizer and seed, insurance claims, and manage machinery and personnel.

Market access is an important revenue component. The more access a person has to a market, the more revenue that person is liable to make. From a farming prospective, market access includes access to transportation of crops, knowledge of prices and futures to hedge against uncertainty. Better transportation allows farmers to find better markets that are farther away. Smart devices help farmers determine the best buyer for their crops and the best seller for their inputs based on price and location. The Growers Edge app displays cash bids at local cooperatives and is available for Android and Apple devices (Growers, 2009).

Besides apps that fit into the four categories mentioned by Vodafone to increase farmers' incomes, other apps purposed for agriculture exist. A study reviewed 60 available agricultural apps and identified them in nine categorizes, some of which correspond with Vodafone's

categories. These app categories are as follows: agriculture information, business, conference, diseases and pests, farm management, learning and reference, location-based, market data, and weather data (Woodill, 2013).

Apps also are available on devices other than smartphones and tablets. They are available on televisions, cars, smartwatches, and heads-up displays. These devices have apps that interface with embedded and external sensors, augment reality, perform tasks formerly accomplished by other technologies, interact with the user through speech,

Apps can be specific to an operating system and also specific to sensor components. For example, the Samsung Galaxy S4 has an infrared blaster that provides the means to control a television. This remote control app is known as WatchOn (Samsung, 2013). This app allows the S4 to perform most functions of a normal television remote including turning on, turning off, changing channels, changing volume, and modifying settings. A new function recommends programs to watch based on viewer preferences and viewing history. Other apps acting as a remote must interface with an external IR source or require a television with WiFi or another wireless connection. WiFi compatible televisions are not normal for older televisions but are more frequent with newer televisions. WatchOn can therefore control old and new televisions but may not be necessary with new televisions equipped with WiFi.

Car manufacturers are adding apps to their vehicles. Ford Sync can pair with a device to interact with compatible apps and phone features. General Motor's vehicles are equipped with OnStar phone capabilities and will perform the functions of a smartphone in future models.

# Chapter 3 - Crop Maturity App

## Introduction

This mobile application mimics the Farm Progress Growing Degree Days™ App (Kansas Farmer, 2011). Growing degree days (GDD) is a heuristic measurement that assigns a value for heat units available for a given day.

The definition of GDD is as follows:

$$GDD = \frac{T_{\max} + T_{\min}}{2} - T_{base} \quad \textbf{(3.1)}$$

where

GDD = Growing degree days

$T_{max}$ = Maximum temperature for a certain day (°C)

$T_{min}$ = Minimum temperature for a certain day (°C)

$T_{base}$ = Constant temperature value that is base for a crop collecting heat units (10°C)

Days = Number of days from a crops plant date to the current date

When accumulated, GDD values can predict the crop stage, a useful tool for crop management. For example, scheduling the application of pesticides or fertilizers is easier when using GDD values than when using a calendar. This is due to the fact that the weather from year to year varies while plants grow in relation to the amount of heat they receive.

The Farm Progress GDD™ App uses weather data provided by the National Weather Service (NWS) API to calculate the number of GDDs accumulated from a certain date. To evaluate the correct temperature date, the app user selects the location with Google Maps or by inputting the zip code. The Crop Maturity App developed here similarly uses weather data from the NWS API and specifies location with the phone's GPS sensor.

The goal of this thesis is to use AGCO's API in developing mashup mobile applications. It therefore was a great fit to develop a mobile application after the GDD™ App. The Crop Maturity App is the product of this work.

## Use Case

The app user interacts with textboxes, labels, and buttons. The user inputs the planting date and field name via textboxes. Buttons are used to trigger screen changes, to affirm dates and field names in textboxes, and to grab information from the phone's clock and LocationSensor.

Upon opening the app, the user is presented with the home screen. From here, the user can view the selected field's GDUs, select a different field to view, or make a new field. As shown in Figure 3.1, the fields are referenced with a location, planting date, and a name .

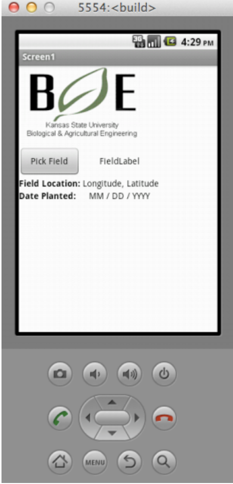



**Figure 3.1. Home page of Crop Maturity App, displaying a ListPicker (Pick Field), field name label (Field Label), labels for the field location, and labels for the date the crop was planted.**

Changing the field being viewed is accomplished through a ListPicker. This looks like a button on the app but behaves differently. When the ListPicker is pressed it displays a list of field names from which the user can choose by touching. This could change the information for the planting date and weather data used as different fields could be planted on different days and use different weather stations.

Creating a new field on the app is initiated with the <Add Field> choice in the ListPicker as shown in Figure 3.2a. Selecting this changes the screen, displaying textboxes for the planting date and for the field name. The field name can be anything the keyboard allows other than the name of a field already entered in the memory. The planting date has a strict pattern that must match in order to be allowed. The user is shown a hint in the text boxes. For example, the

month input has the hint MM to indicate that 2 digits here represent the month. If more or less than 2 digits appear, then the month input is invalidated. If these inputs are valid , as the new field is added to the field list and its GDD values are then displayed on the home page as shown in Figure 3.2c.

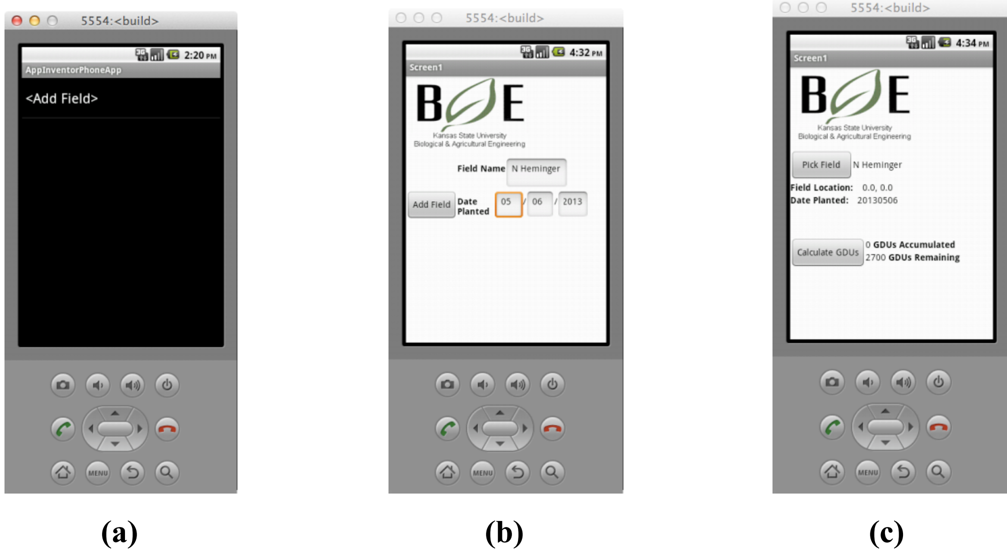



**Figure 3.2. (a) Display after ListPicker has been clicked. <Add Field> is the only choice initially. (b)Add Field page. (c)Home page with a field selected.**

The Crop Maturity App calculates the accumulated GDDs, using temperature data from the AGCO Weather API that in turn receives its information from the NWS API. The user establishes the field location when clicking the 'Add Field' button on the Add Field page. This triggers a call to the phone's LocationSensor that then sends back the latitude and longitude coordinates for the phones position.

## User Interface

The user interface is the end product of the described use case. It has the following components: three textboxes, multiple labels, a listpicker, two buttons, a LocationSensor, a Clock, a TinyDB, and a Web component. The textboxes, labels, listpicker, and buttons are visible to the user whereas the other components work in the background to provide data to the visible components.

The app only has one screen but appears to have two screens. This is accomplished by manipulating its visible components so that they are either hidden or visible. For this app, the

add field view is made visible if the user selects <Add Field> in the listpicker. The home page and the GDU view are visible after the field is added.

## Model

The model uses an adaptation of Equation 1 to determine the accumulated GDDs for a crop. Accumulating the GDDs for a period of time from the date a crop was planted until the current date is simply a summation of the GDDs for each day in that period. This is given in Equation 3.2, below.

$$\sum GDD = \sum_{Day=Planted}^{Current} \left( \frac{T_{\max} + T_{\min}}{2} - T_{base} \right) \qquad \textbf{(3.2)}$$

Using Equation 3.2 as the model would require an iterative procedure. This is feasible, but it would mean an iterative calculation would have to be made. A better model is a rearrangement of the right side of Equation 3.2, and is shown below, as Equation 3.3.

$$\sum GDD = \frac{\sum_{Day=Planted}^{Current} (T_{\max} + T_{\min})}{2} - T_{base} * Days \qquad \textbf{(3.3)}$$

Therefore, Equation 3.3 is used as the model to calculate the accumulated GDDs. This calculation can be broken down in steps and is not iterative. Equation 3.3 requires summations of the temperatures and the number of days from the plant date to the current date. These requirements are met in the delivery of data from the AGCO Weather API, and in the data processing by the blocks for the Crop Maturity app in App Inventor's Blocks Editor.

## AGCO Weather API

Weather data delivery is simplified with the AGCO Weather API. The purpose of the AGCO Weather API is to serve as a mediator between the National Weather Service (NWS) API and the Crop Maturity App. While not currently online, the envisioned AGCO Weather API accepts requests in the form of a URL Post and returns data that is compatible with App Inventor.
An example of the URL post is

http://ag-hes-server.appspot.com/getgrowdayinfo?tag=gettempdata&cmd=gettempdata&fmt=ai&long=95.65344&lat=35.65334&sd=20120522&ed=20120914

Components of the URL are further explained in Table 1.

**Table 3.1 AGCO Weather API call to retrieve high and low temperatures for a period of days.**

| URL Components | Component Explanation |
|---|---|
| http://ag-hes-server.appspot.com/ | This is the base URL for AGCO's API server. |
| getgrowdayinfo? | AGCO Weather API section. This name will likely change to reflect its broader purpose with weather. |
| tag=gettempdata& | Tag that is returned to app after the API has processed the request. |
| cmd=gettempdata& | Command given to API. It has the same value as the tag so could possibly be removed. |
| fmt=ai& | Specifies that App Inventor is making the request. |
| long=95.65344& lat=35.65334& | Field location information. This is a latitude and longitude point in the field and is used to determine what weather station to use. |
| sd=20120522& ed=20120914 | Dates for when the crop was planted and the current date. This has the format yyyymmdd. |

The data delivered from the AGCO Weather API to the requesting app has JSON Text format. The data is transmitted in a single package and has the following form.

"gettempdata,2,100,104,75,77"

Components of the package delivered from the AGCO Weather API are explained in Table 2, shown below.

**Table 3.2 AGCO Weather API response to Crop Maturity App request.**

| Component | Explanation |
|---|---|
| “gettempdata, | Tag for request |
| 2, | Number of days from start date to current date. Should also correspond to the number of entries in the $T_{max}$ list and the $T_{min}$ list. |
| 100,104, | $T_{max}$ list. $T_{max1}$, $T_{max2}$, … |
| 75,77” | $T_{min}$ list. $T_{min1}$, $T_{min2}$, … |

The API always returns the tag as the first item regardless of the call. The rest of the response fits a pattern for the other API calls. The number of days corresponds to the number of items for each list. For example, the 2 given in Table 2 is the number of days returned that have values for both the temperature highs and lows. Returning the number of items in each list permits simple parsing of the data. Grouping the data by temperature type instead of by day is another pattern that is seen in the API responses. The day that they correspond to is referenced by their index in the list.

Another way to order the response would be to group the data according to day instead of temperature type. This is also feasible but it was decided to group by type.

## Blocks Editor

The blocks shown in this section are concerned with how the app works with the AGCO server to receive data. These blocks implement what is defined in the “AGCO Weather API” and in the Model sections above. The blocks are arranged in a logical order in explaining how data is handled starting from the request for data from API call and ending at calculating the accumulated GDDs.

The API call that requests weather data is shown below in Figure 4. Figure 4 is specifically a procedure block named “getTempsDigest” that posts the request via the Web component whenever it is called. The variable “FieldLongitude,” “FieldLatitude,” “PlantDate,” and “CurrentDate” are passed into the procedure to specify the field.

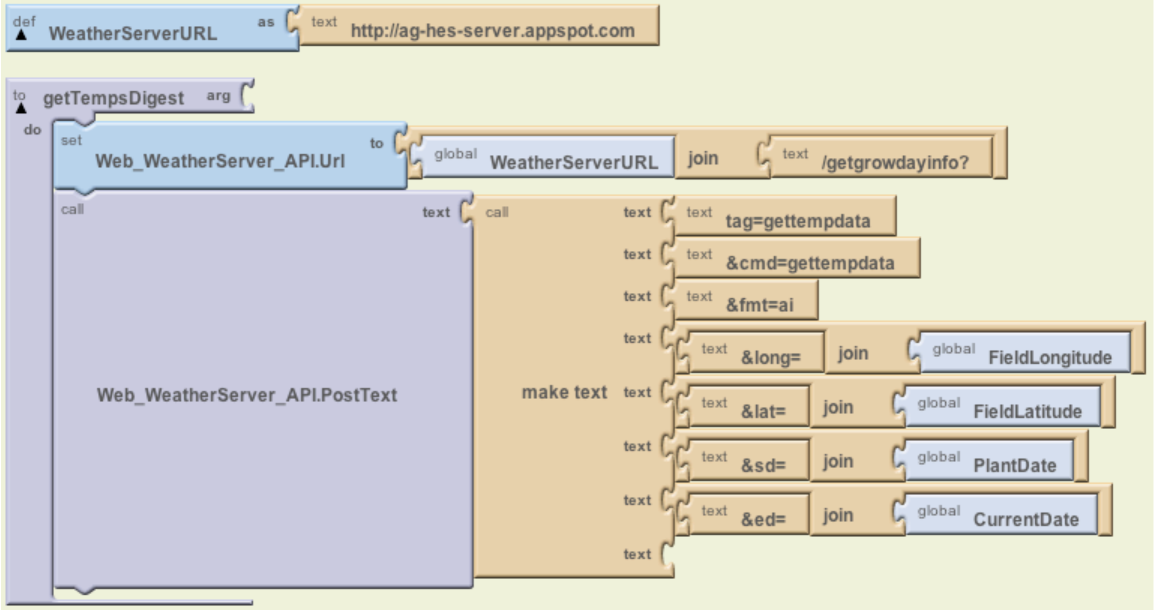



**Figure 3.3. Procedure for making a request to the AGCO server.**

The request for data is normally followed by a short delay. An event block in AI is able to handle waiting for the response and categorize it based on url, responseCode, responseType, and responseContent as shown in Figure 5. For the purposes of this app, only responseCode and responseContent are necessary. The responseCode is used to ensure a valid response and responseContent contains all of the useful data. The event block contains a procedure called parseResponse described later and a conditional block that checks the variable IncomingTag with

text “gettempdata”.  If a match is detected, the procedure inside this conditional block is called, otherwise nothing happens here.

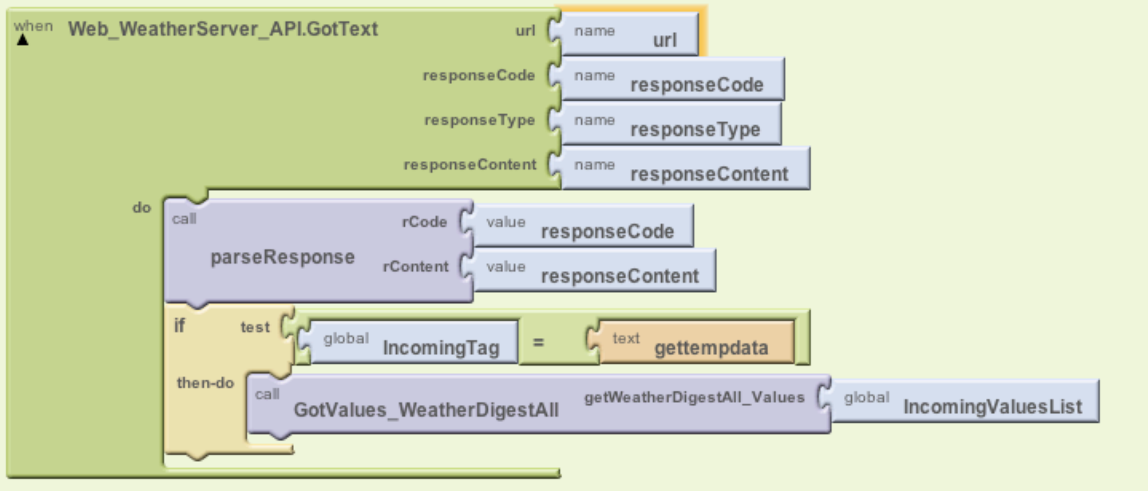



**Figure 3.4.  Event block that occurs when the app has received a response from the AGCO server.**

The first procedure called in the event block above is parseResponse that is shown below in Figure 6.  This block is passed the responseCode and responseContent as arguments.  It checks the responseCode and if equal to 200 it is validated and the responseContent is handled according to procedures.  Otherwise, the user is shown that an error has occurred and the accumulated GDD were not able to be determined.

Given a responseCode of 200, the responseContent is separated at commas and split into a list. The first item is then taken from the list and given to the IncomingTag.  This completes the parseResponse procedure and the path continues with the conditional block of the Event Handler. Assuming a match with “gettempdata,” the procedure GotValues_WeatherDigestAll is called and passed the list IncomingValuesList.

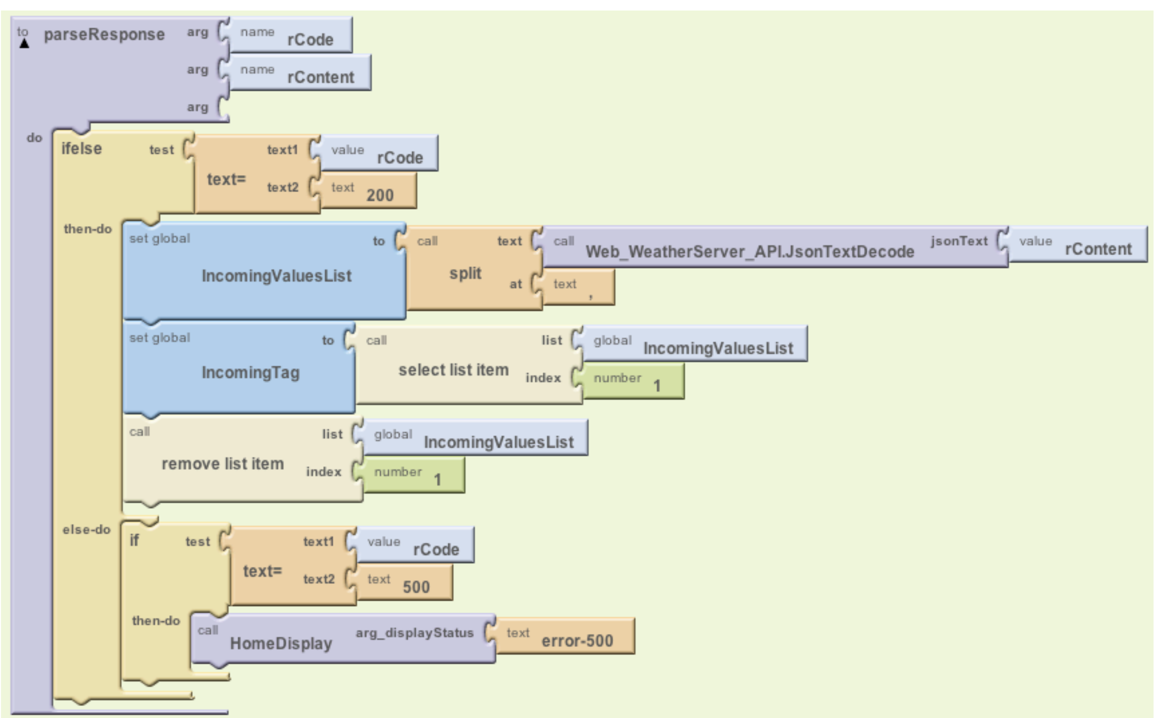



**Figure 3.5. The parsing procedure places the response in the IncomingValuesList and removes the IncomingTag.**

Figure 7 shows the block procedure GotValues_WeatherDigestAll. This procedure finishes the task of sorting the data according the API and then calls other procedures to perform the GDD calculations. Sorting the data involves storing the number of days from the planted date to the current date in a variable and the temperature highs and lows into lists. Setting the variable Days is simple since it has a single value. Setting lists requires a slightly more difficult process. The first part is to clear the list values. This eliminates the possibility of retaining values not coming from the current API response. The second part is sorting the values from the Data list and setting them in their respective lists.

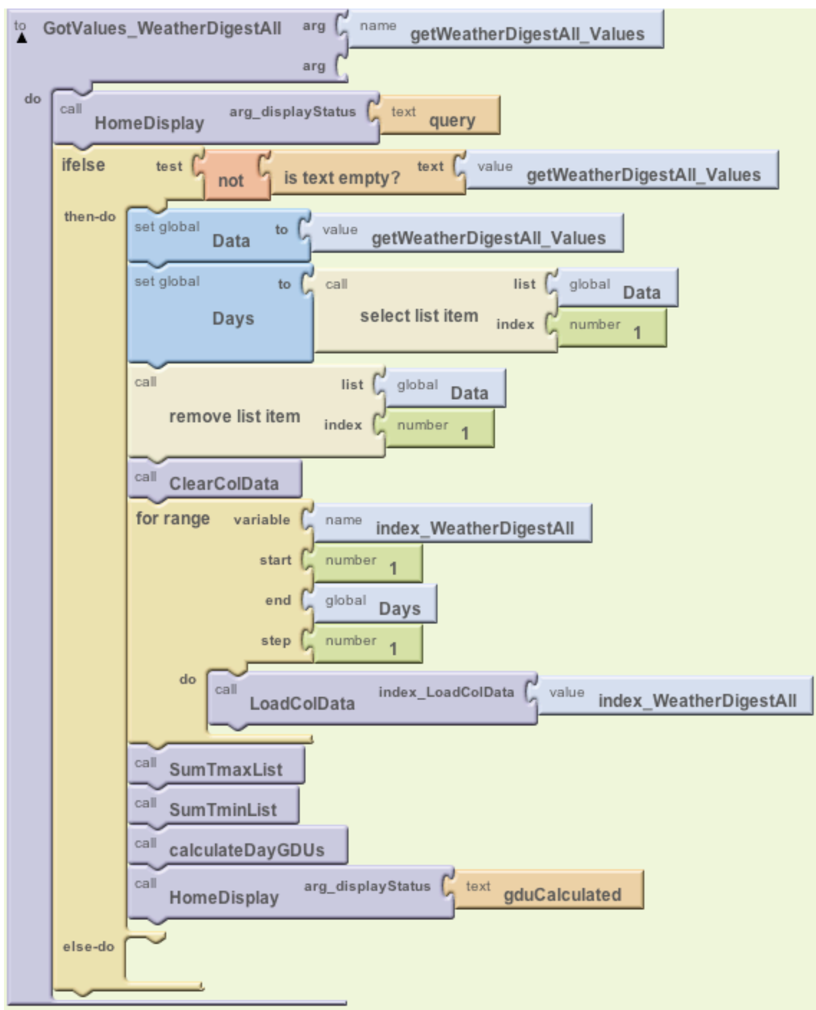



**Figure 3.6. Procedure to fill column data, perform calculations, and display information.**

The procedure to clear and load the TmaxList and TminList is shown in Figure 8 below and are called ClearColData and LoadColData, respectively. ClearColData simply empties any current data that may be held the lists. LoadColData sorts data into the TmaxList and TminList according to the API response. AI provides multiple blocks for handling list items.

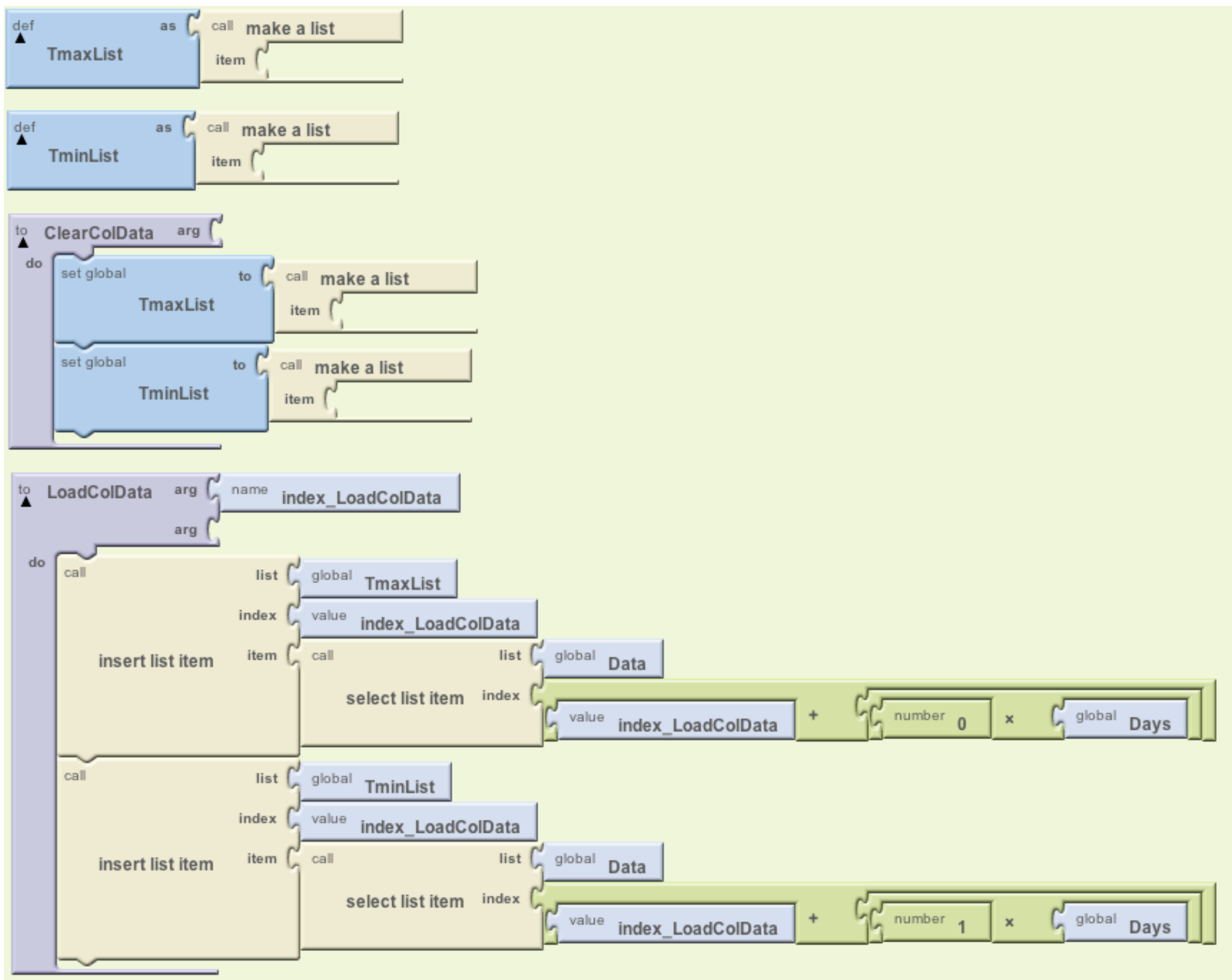



**Figure 3.7. Procedures for storing column data.**

The calculations are performed by three blocks shown in Figure 3.8. Two of the blocks are summations of the items in TmaxList and TminList. The final block is a representation of Equation 3.3.

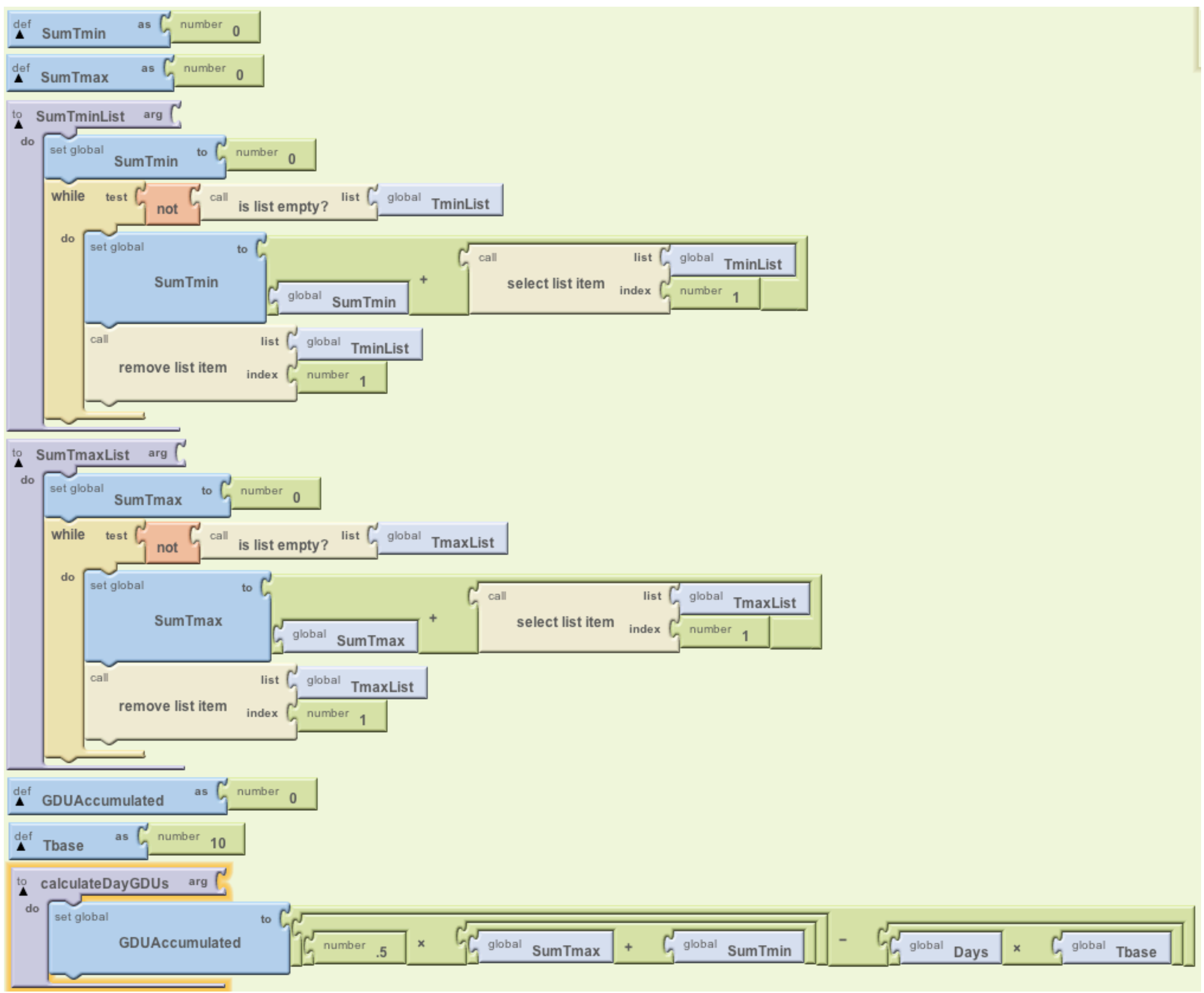



**Figure 3.8. Blocks performing GDD calculations.**

## Results

The concept for this app was simple to establish, however, a problem existed with the AGCO servers retrieving historical weather data. This problem remains unresolved with the app implementation. An app implementation that would solve this problem would require the user to set the planting date on or before the planting date so that weather data would be current. A

solution from the server side could possibly exist where historical weather data would be retrieved.  This would be ideal as the user could set the planting date for a past date.

# Chapter 4 - Combine Efficiency App

## Introduction

The Combine Efficiency App is discussed in this chapter. This app was repurposed from an Excel spreadsheet that a farmer used to determine the optimal harvest speed for a combine. The advancement from a spreadsheet to an app lets the farmer monitor his combine fleet in near real-time and from a remote location.

Optimal harvest speed is defined as the speed of a combine at which the operating expense is the lowest. The model for this app assumes that the speed of the combine is the only factor that determines the operating expenses. This is a simplified model that does not account for variability in fan speed or concavity, both of which are important factors in crop losses.

The data used in the model come from two sources: a farmer and AGCOMMAND. The farmer provides knowledge concerning operation factors and costs to the model. All machine data is provided by AGCOMMAND, AGCO's telemetry system and is transferred by the Hesston server.

## Use Case

The end user of this app is a farmer or a farm manager who typically has multiple operating combines that need to be monitored. Upon opening this app for the first time, the login page is displayed. This page has the user information, account information, and timing rates. The user information is the user's name, email, and phone number. The account information is a username, password, and fleet. This is the most essential information necessary for obtaining machine data from AGCOMMAND. The timing rates are not necessary to change and can be left alone. The timing rates are time zone, app sync interval, and machine sync rate. The time zone is relative to Coordinated Universal Time and is in units of hours. The app sync interval and machine sync rate are the time between calls the app will make to the server to refresh its data and are in units of milliseconds.

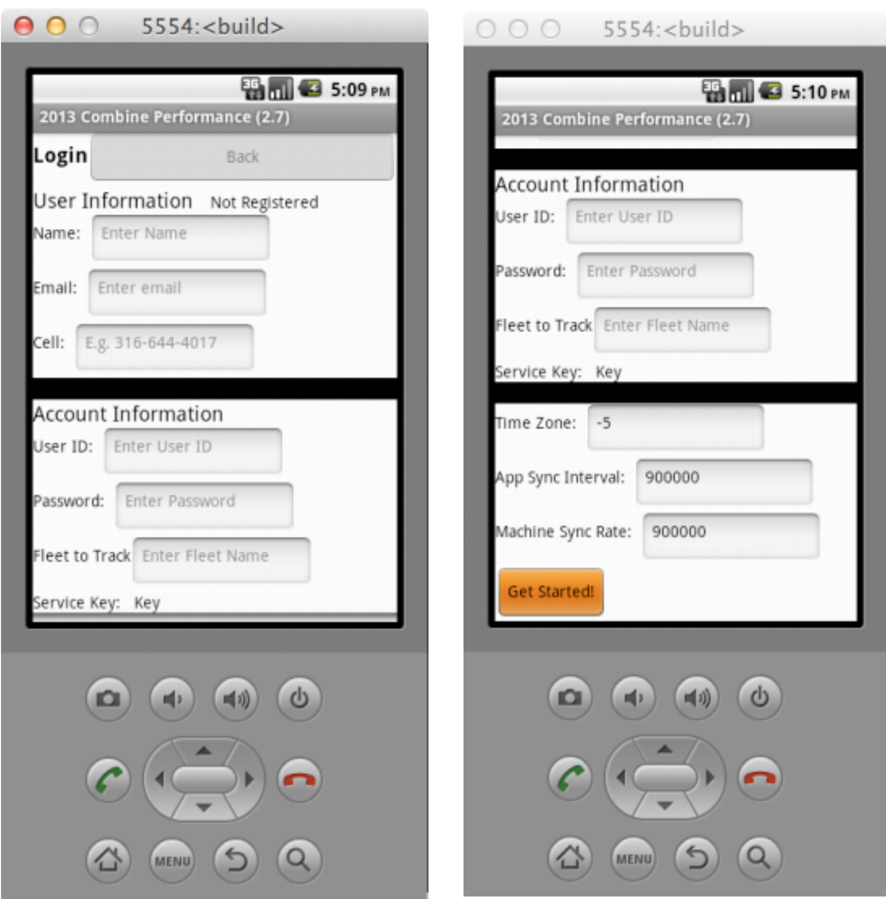
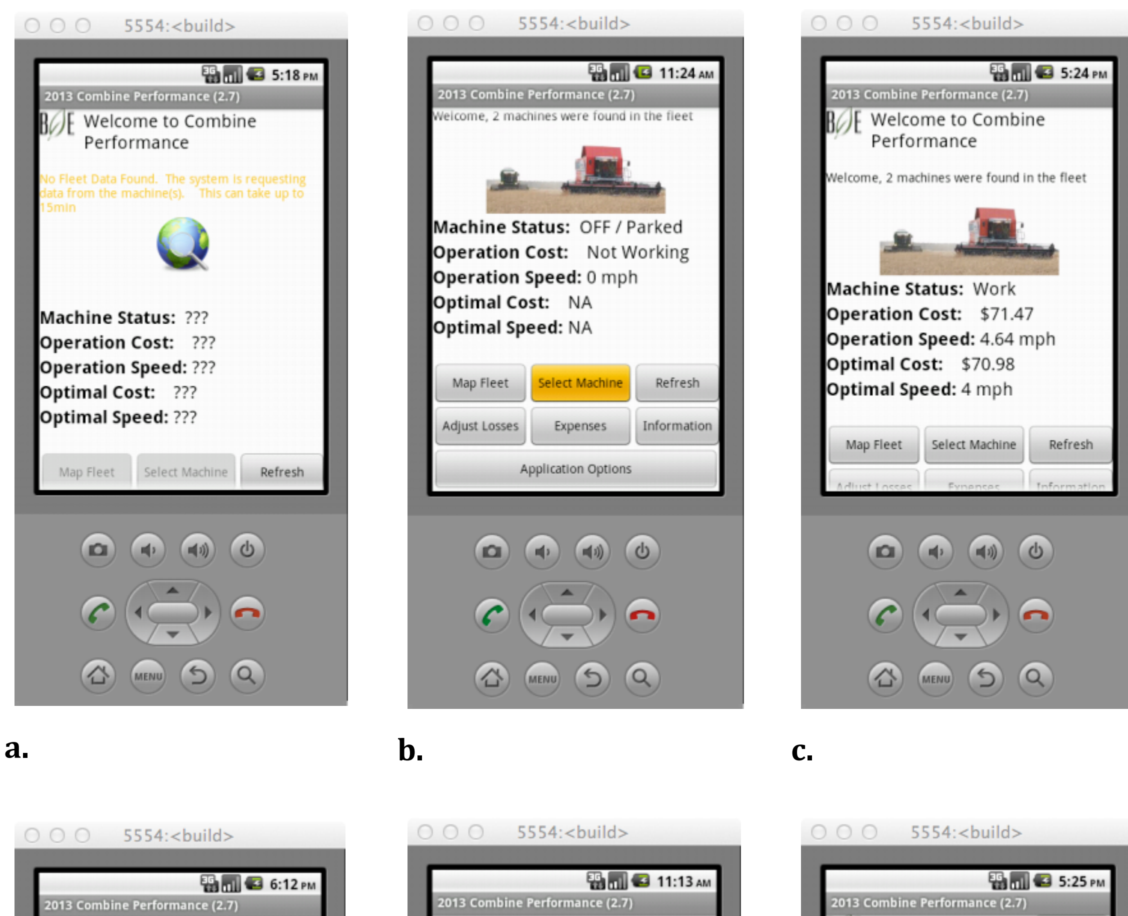



**Figure 4.1. The "Login View" contains nine entries: three for user information, three for account information, and three for time factors that can be left alone.**

After a successful login, the app displays the home page with the first machine in the fleet selected. The information for this machine is its status, actual speed, actual cost, optimal speed, and optimal cost. The app user can then choose between several buttons and a listpicker for changing inputs to the model or changing the machine.



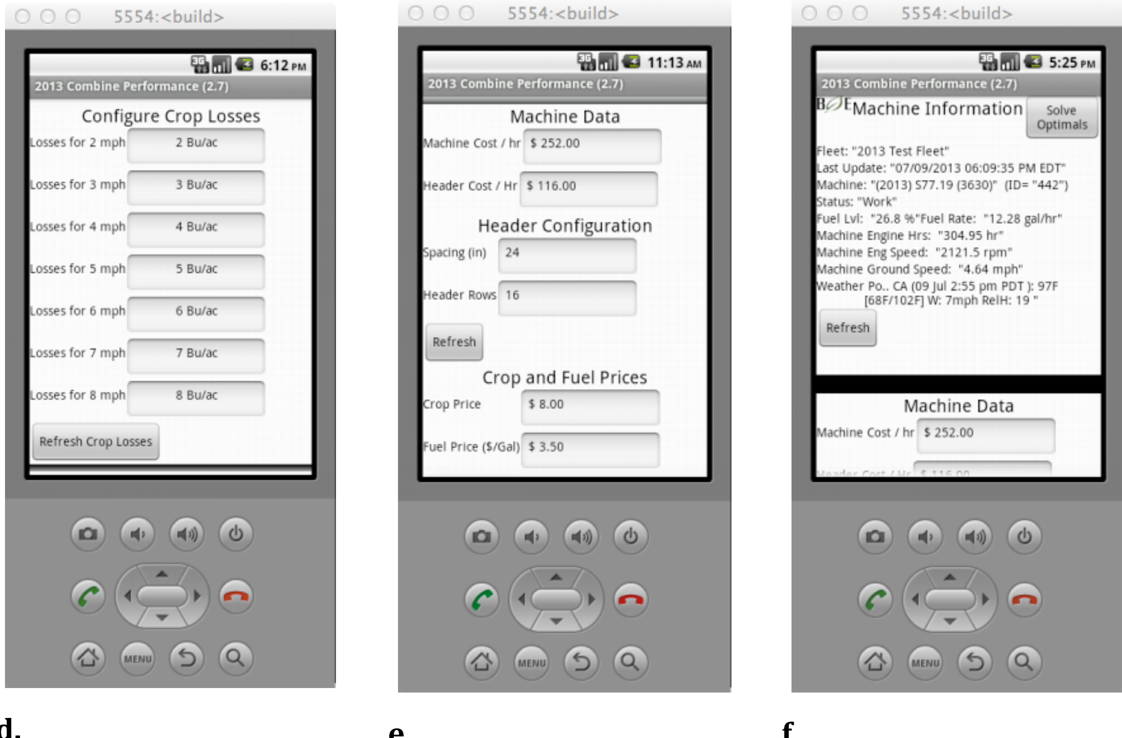




**Figure 4.2. Views from the Home and Machine sections of the Combine Performance App. Views a, b, and c show the "Home View" while d, e, and f show the "Machine View."**

## Model

The model determines the optimal speed and the optimal cost of operating a combine with certain factors known by the farmer. These factors include machine cost per hour, fuel price, crop price, header length, fuel consumption, and crop losses. The operating expenses are categorized as machine cost, losses cost, and fuel cost and are calculated in units of dollars per acre ($/ac). The losses cost typically increases with increasing speed, while the machine and fuel cost typically decrease with increasing speed.

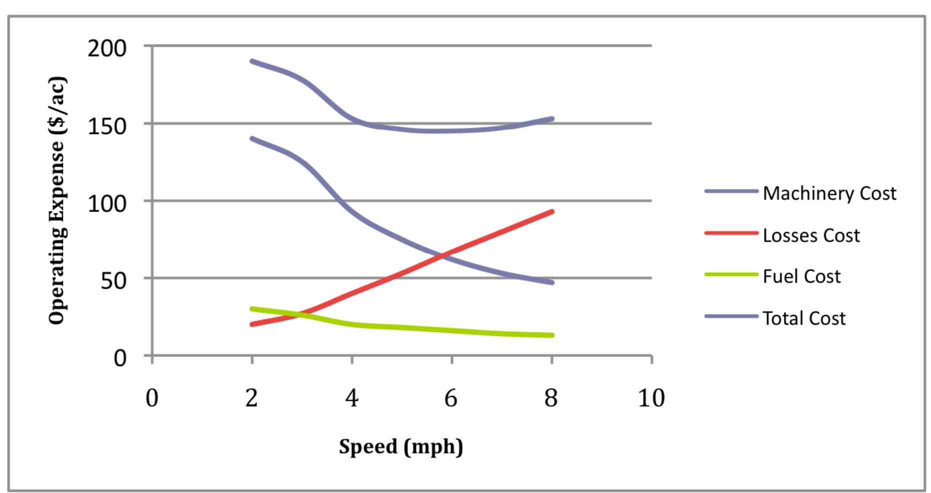



**Figure 4.3. Speed vs. Operating expenses for a combine harvester**

The farmer enters the factors fuel consumption and losses. Both of these factors vary according to speed. Integers from two to eight are used to represent the speed of a combine in mph. The expenses are then determined at those speeds with Equations 4.1, 4.2, 4.3, 4.4, and 4.5.

$$P = S * R * N * 5280 / 43560 \quad (4.1)$$

$$Cm = H / P \quad (4.2)$$

$$Cl = L / Pc \quad (4.3)$$

$$Cf = Pf * Fc / P \quad (4.4)$$

$$Ct = Cm + Cl + Cf \quad (4.5)$$

where

P = performance (acre/hr)

S = speed (mph)

R = row spacing (in)

N = number of rows

Cm = machine cost ($)

H = machine cost per hour ($/hr)

Cl = grain losses cost ($)

L = grain losses (bushels)

Pc = crop price ($/bushel)

Cf = fuel cost ($)

Pf = fuel price per gallon ($/gal)

Fc = fuel consumption (gal/hr)

Ct = total cost ($)

## API

The login/register request is the first API call. This request retrieves a key. If successful, this key is then used to authorize requests for machine data. Table 3 shows the registration request.

**Table 4.1. HTTP Post Request for Registration Key.**

| URL Components | Component Explanation |
|---|---|
| http://ag-hes-server.appspot.com | The base URL for the Hesston API server |
| getfleetownerinformation | Hesston API section |
| tag=registration | Tag that requests a registration key |
| cmd=reg | Command |
| fmt=ai | Format specifying that App Inventor is making the request. |
| n=”name” | The user’s name |
| e=”email” | The user’s email |
| c=”cell” | The user’s cell phone number |
| u=”username” | The username to Hesston server |
| a=”password” | The password to Hesston server |
| d=”app name” | The app’s name |
| k=“key” | The key is handed to Hesston API server to verify authorization to receive information. |

The fleet data request is the same every time it is called. The key is the component that is given to the app from the Hesston server and is essential to receive information from the server. This requests all the data that the Hesston server has aggregated for a fleet at the time of the request. The URL post is
http://ag-hes-server.appspot.com/getfleetperformance?tag=getFleetDigestAll&cmd=fleetdigestall&fmt=ai

**Table 4.2. HTTP Post Request for Stored Data.**

| URL Components | Component Explanantion |
|---|---|
| http://ag-hes-server.appspot.com | The base URL for AGCO's API server |
| getcbfleetinformation | AGCO Fleet API section. |
| tag=getSelectData | Tag that requests all the information available for the fleet. |
| fmt=ai | Format specifying that App Inventor is making the request. |
| cmd=getselectdata | Command given to the AGCO API server. |
| k="key" | The key is handed to AGCO API server to verify authorization to receive information. |

The Hesston response delivers all the fleet data ordered according to machine index in the fleet and type of data.

An example response from the Hesston server for two machines is given below. The data is held as comma-separated values (csv). The first three values are information about the response and are the tag, number of machines in the fleet, and the key. The remaining values contain machine information grouped by information type and indexed by machine.

"getSelectData,2, ag9zfmFnLWhlcy1zZXJ2ZXJyEAsSCEZsZWV0S01MGOXfHAw, \"2013 Test Fleet\" , \"2013 Test Fleet\", \"(2013) S77.18 (2793)\" , \"(2013) S77.19 (3630)\", \"340\" , \"442\", \"Harvester (Combine)\" , \"Harvester (Combine)\", \"6205042217\" , \"6205042217\", 45.6463432 , 39.9581756, -119.4072952 , -101.1401672, \"07/11/2013 09:06:27 PM EDT\" , \"07/11/2013 11:00:07 PM EDT\", \"0 mph\" , \"0 mph\", \"false\" , \"false\", \"921.35 hr\" , \"318.15 hr\", \"0 rpm\" , \"13.5 rpm\", \"OFF / Parked\" , \"OFF / Parked\", \"585.54 hr\" ,

\"163.67 hr\",  \"66 %\" , \"24.8 %\",  \"0 gal/hr\" , \"0 gal/hr\",  \" Hermiston, Herm.. OR (11 Jul 7:53 pm PDT ): 84F [55F/82F] W: 10mph RelH: 16 \" , \" McCook Municipa.. NE (11 Jul 21:53 pm CDT): 82F [70F/102F] W: 16mph RelH: 56 \",  \"No Data\" , \"No Data\",  \"No Data\" , \"No Data\",  \"No Data\" , \"No Data\",  \"No Data\" , \"No Data\",  \"No Data\" , \"No Data\", \"No Data\" , \"No Data\",  \"No Data\" , \"No Data\",  \"No Data\" , \"No Data\",  \"No Data\" , \"No Data\",  \"No Data\" , \"No Data\""

Before receiving viable data, a request to synchronize machine data must be made.  This request tells the Hesston server that data for a specific fleet needs to be aggregated.  The response from the server signifies the success or failure of the sync request.  The request components are shown in Table 4.3, below.

**Table 4.3.  HTTP Post Request to Sync Machine Data.**

| URL Components | Component Explanation |
|---|---|
| http://ag-hes-server.appspot.com | The base URL for AGCO's API server |
| getcbfleetinformation | AGCO Fleet API section. |
| tag=requestMachineDataSync | Tag that requests all the information available for the fleet. |
| cmd=fleetdigestall | Format specifying that App Inventor is making the request. |
| fmt=ack | Command given to the AGCO API server. |
| k="key" | The key is handed to AGCO API server to verify authorization to receive information. |
| f="fleet name" | The fleet name is handed to the AGCO API server to single out the fleet |

## Blocks Editor

The block code for this app includes blocks that change the page view, blocks that make requests and handle responses from a server, blocks that create a model from data, and blocks that accept data from an app user.  These blocks are explained in the following sections. They may overlap one another at points.

### *Initialization*

The first block executed in the source code is the Screen1.Initialize block, shown in figure 4.4 This block initializes values for UserAccount and Index_Machine, and disables the timers.  The

main function of this event block is to call the procedure_App_Initialization block, shown in figure 4.5.

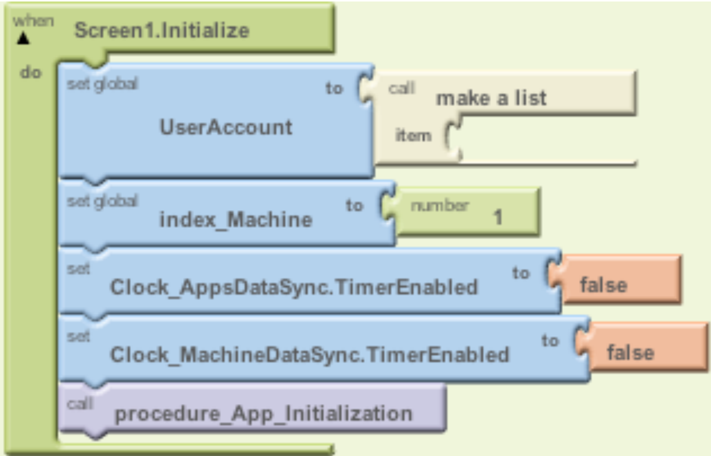



**Figure 4.4. Event block for the app initialization.**

The procedure_App_Initialization chooses what page will be shown.  This procedure is called after the app is initialized and the register button is pressed.  It checks for a value saved in the user account.  This procedure has three pathways that are contained within two if-else tests and are explained below.

The first pathway is a false result of the first “if-else” test.  If no value is found in the user account, the WindowToDisplay procedure is called with the argument “register” signifying the app user will be shown the register page.  This is what initially happens when the app is first opened.  Upon initial opening, the user must insert their information.  This information is used to restrict machine data to people with valid credentials and to specify the fleet.

After the first time a user enters their information, it is saved on the phone’s database and is recalled and set to the UserAccount as seen in the first block.  The procedure then checks the value of UserAccount and finds whether it has content. If the answer is yes, the first if-else test returns a true value and the code proceeds to the second if-else test, which checks whether the information entered was accepted by AGCO.  If the second if-else test returns a true value, the sixth item in the UserAccount is changed from the initial value of “Key” to a new key sent back by AGCO and this new key is then stored in the UserAccount on the phone database.

The second pathway is a false result of the second “if-else” test.  The value for the sixth item in the UserAccount is checked against the string ”key.”  This is understood to be the initial value set in the app but not a value sent by AGCO as a confirmation of a valid entry.  This redirects the user to the registration page.

The third and final pathway is a true result for the second if-else test.  The value for the sixth item of UserAccount does not match “key”, indicating that the app has received a confirmation from AGCO.  At this point, the view is changed to the home page and two requests are sent to AGCO while the timers are set.

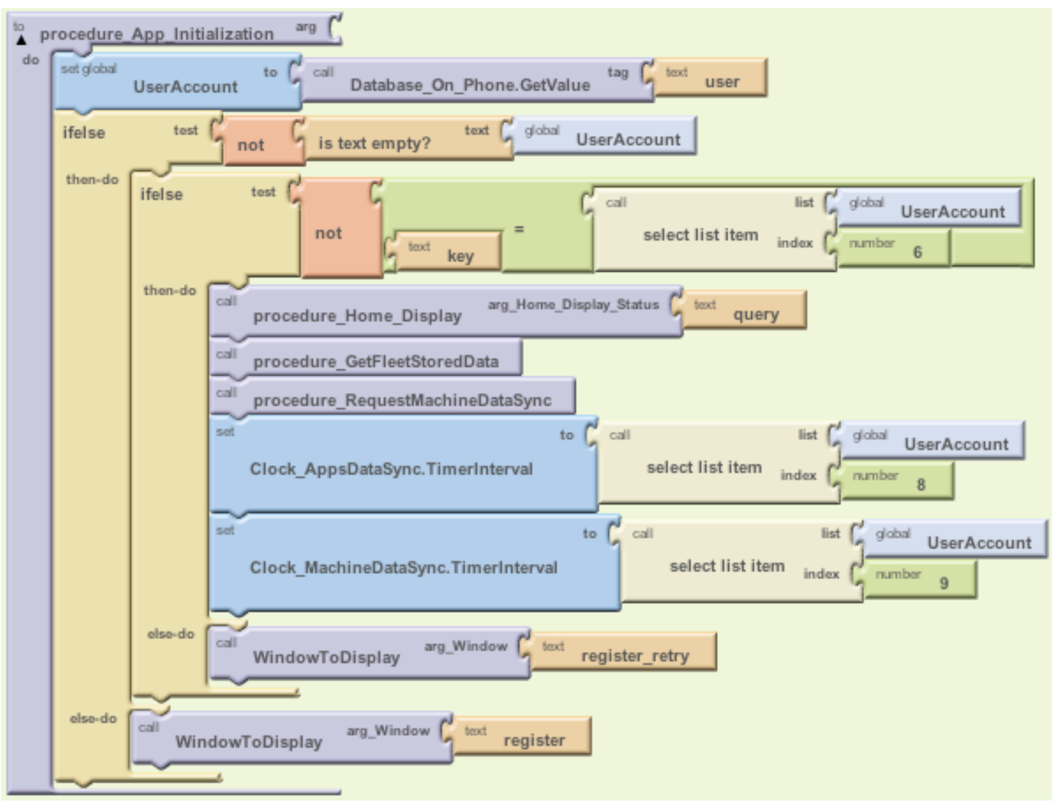



**Figure 4.5. Procedure block that is called upon the app initialization and registration.**

The WindowToDisplay procedure takes an argument that allows the developer to specify which components should be visible to the user. The components are grouped together on the user interface in arrangements made available by App Inventor. These arrangements, as well as most components have a visibility property. It is this visibility property that is being changed by the procedure and altering the page shown to the user. It is very simple to add new views to the procedure by simply adding a new if-then test to the block shown in figures 4.4 to 4.6. If certain components within an arrangement needed to be invisible in one view and then visible in another view, the developer must carefully specify all components and make changes on the visibility accordingly. A visibility error could occur if the developer assumes that an invisible component will be visible. This is due to the visibility property that each component has as well as the arrangements. For instance, in this app, the components in the machine arrangement (VA_Machine) have their visibility changed corresponding with the arguments expenses, configLosses, and machineStatus. If VA_Machine is visible then these components must be managed to specify what is visible. This lengthens the WindowToDisplay block considerably as

seen when comparing the sub block under the argument "register" with the sub block under the argument expenses.

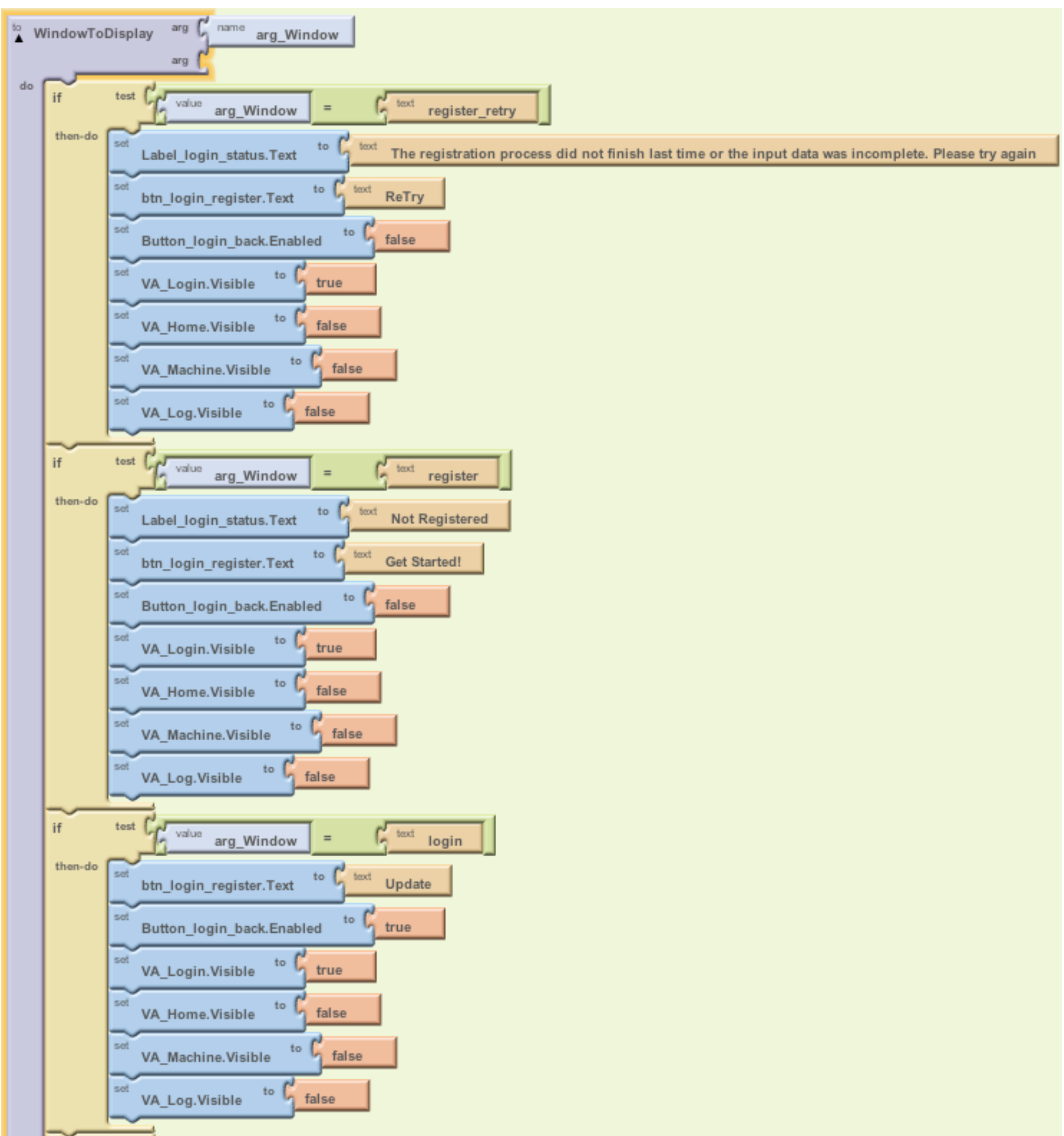



**Figure 4.6. Procedure block controlling the view displayed to the user. Three argument entries are possible in this figure to return a true. These are "register_retry, register, and login."**

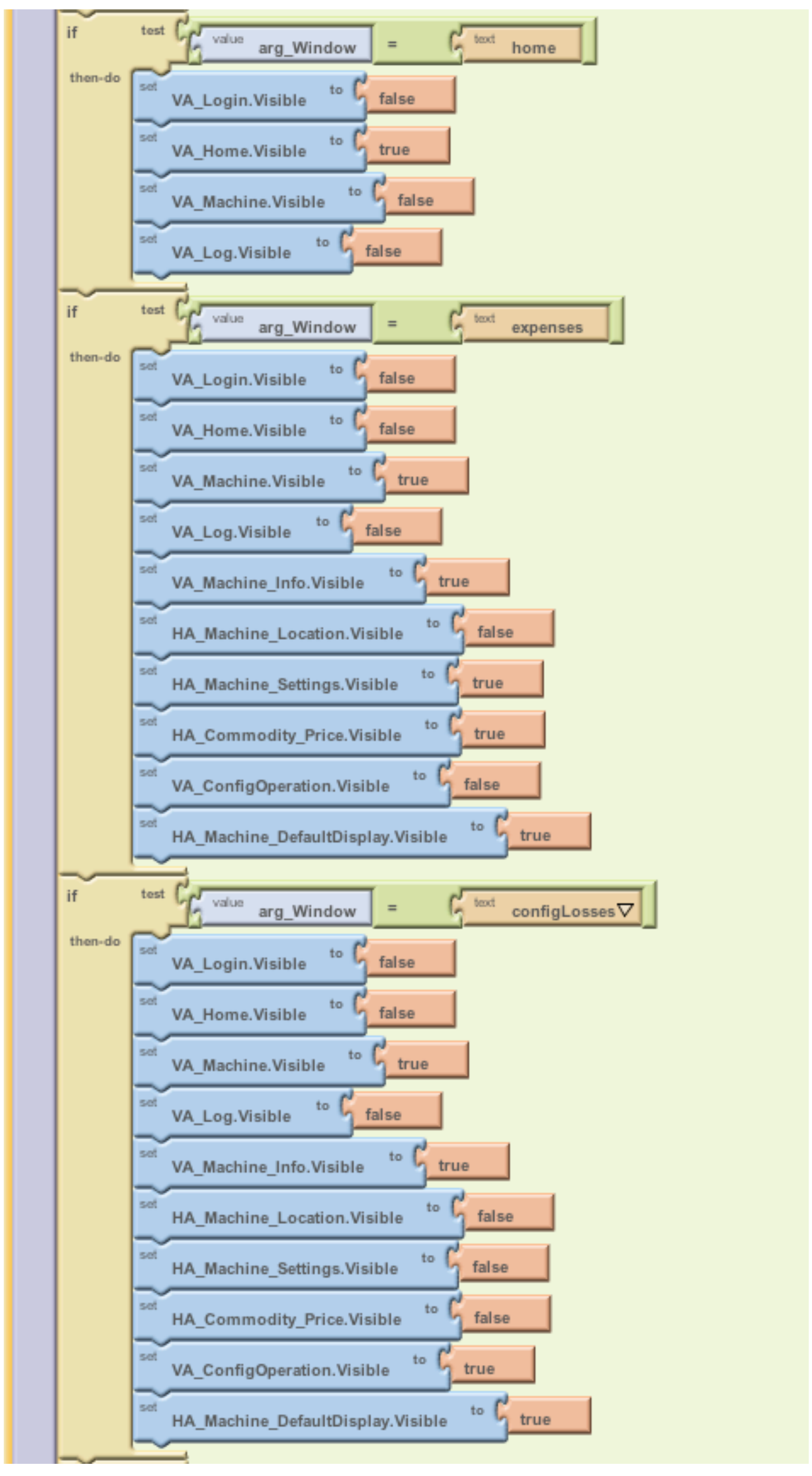



**Figure 4.7. Continuation of the "WindowToDisplay" block. Argument entries include "expenses, and configLosses."**

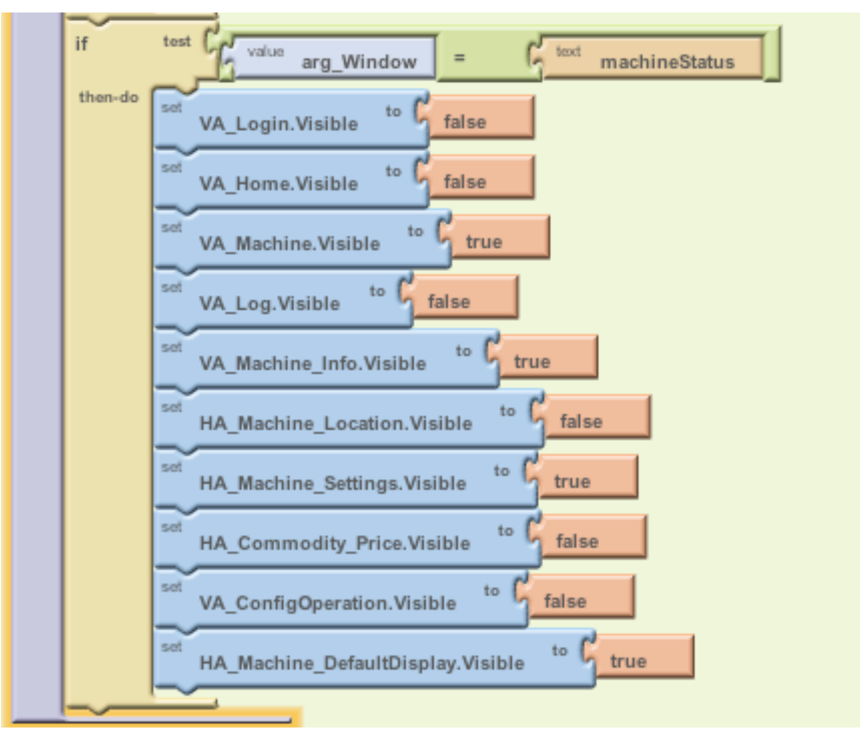



**Figure 4.8. Final part of "WindowToDisplay" block. "MachineStatus" is an argument entry.**

### *Registration and Logging In*

The registration page holds many textboxes that are filled with the user's information. This page also has a button which, when clicked, triggers two procedures, changes the text on a label and on itself to indicate that a process is underway. The Button_login_register.Click block code is shown in figure 4.8. The two procedures are procedure_login_register and procedure_GetRegisterKey and are explained in the following paragraphs.

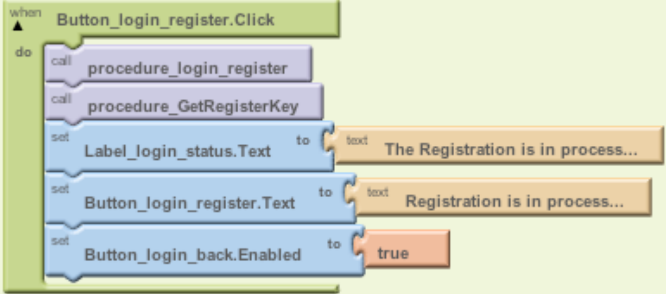



**Figure 4.9. Event block occurring with the click of the register button.**

Figure 4.9 is the procedure procedure_login_register. This procedure checks the values in the textboxes for timezone, app sync rate, and machine sync rate to verify that they aren't empty and

they are numbers. The textboxes from the register page are then filled in to a list and stored as the UserAccount. This list is stored on the phone in the block Database_On_Phone.StoreValue and recalled with the procedure_App_Initialization. This means that a user only needs to register once on their phone and thereafter the home page will always be shown when opening the app. However, the user may also go back and change the information. This is necessary for users that have multiple fleets or for changing users.

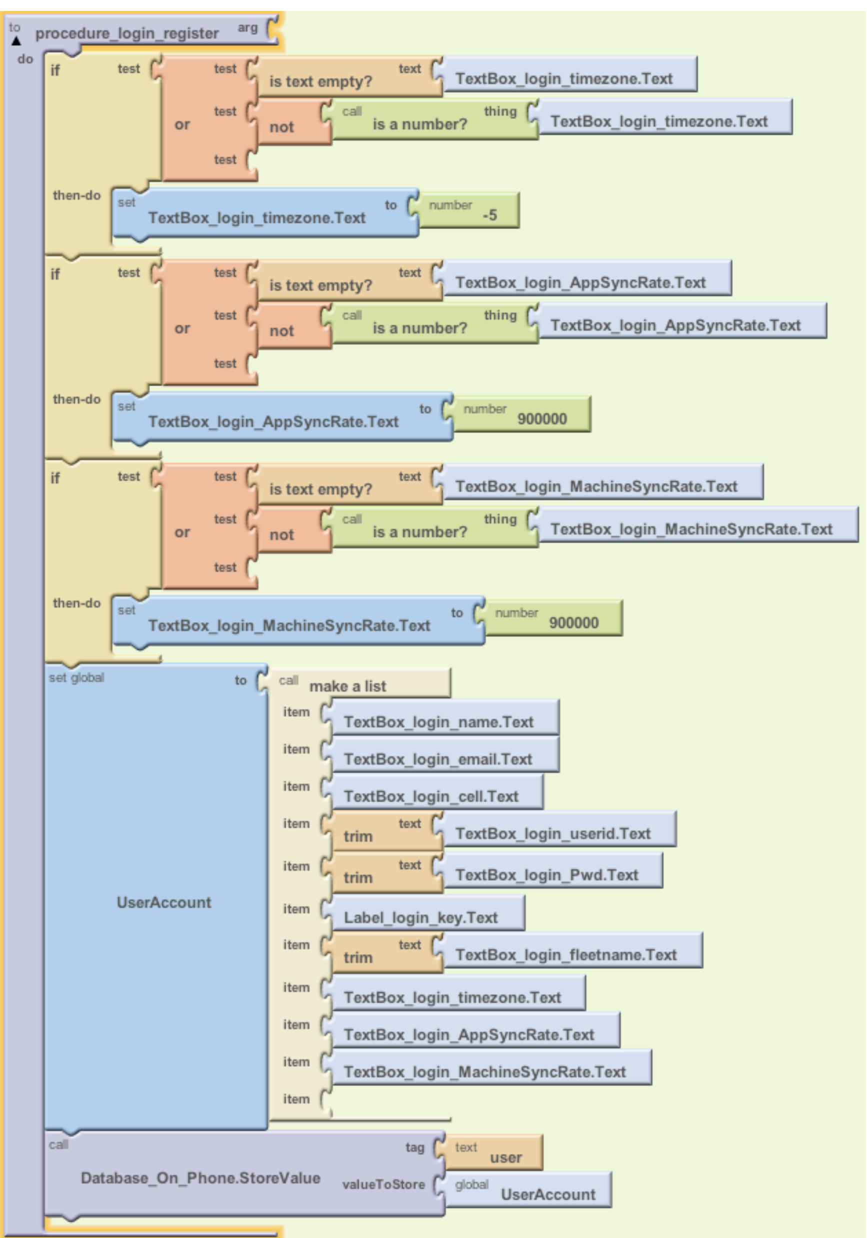



**Figure 4.10. Procedure block executed during registration. This procedure checks the time values and stores the textbox entries on the "Login View" to a UserAccount.**

The procedure_GetRegisterKey, shown in figure 4.10, is called next within the Button_login_register.Click block code. This block posts a request to the AGCO API server with the user information, excluding the final three items that involve time intervals. The response to this request is waited on by the Web_AgServer_API.GotText block shown in figure 4.11.

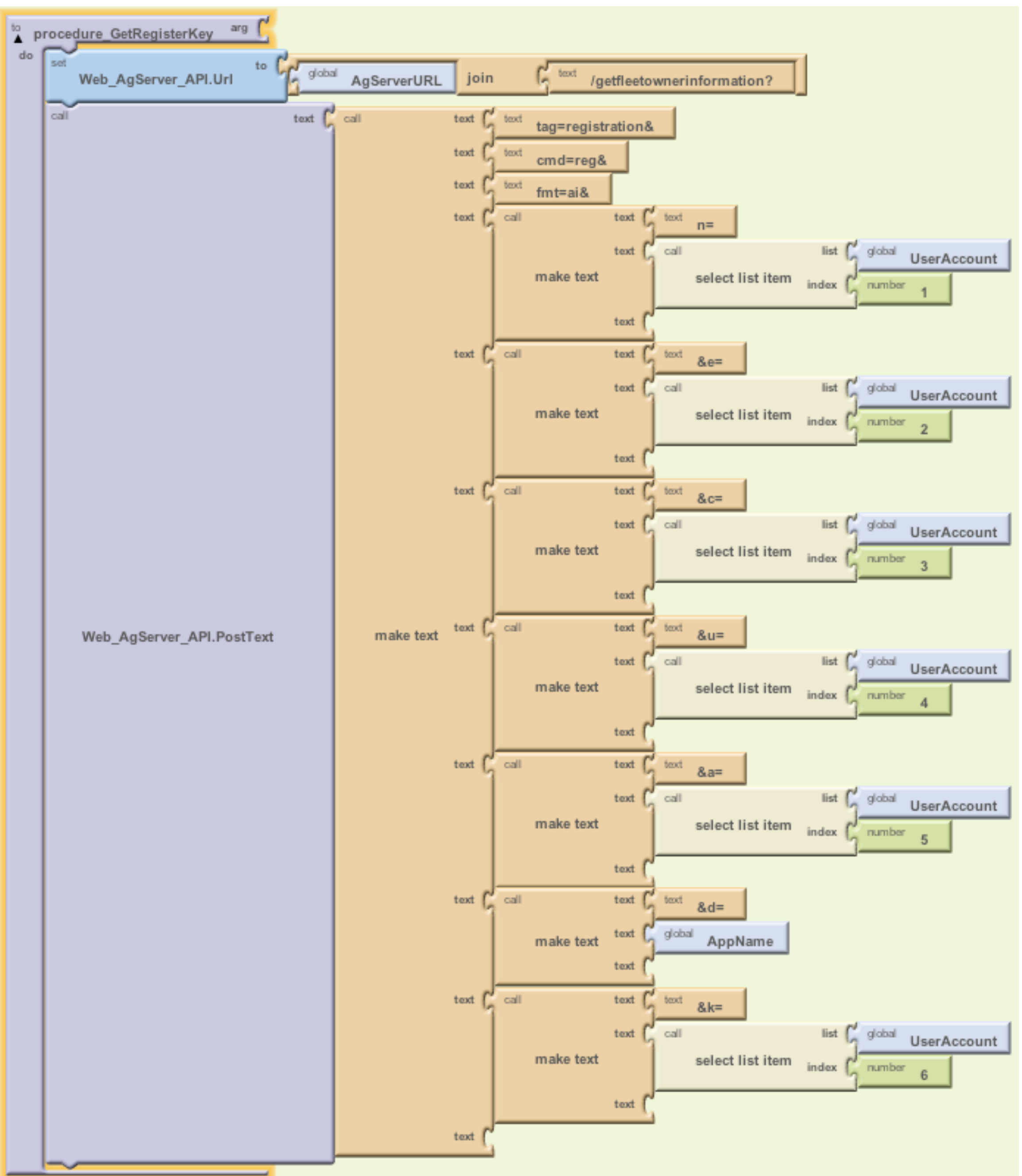



**Figure 4.11. Procedure block that Posts request to Hesston server for a key.**

The Web_AgServer_API.GotText block handles all of the responses from the AGCO server. Every response has four categories: url, responseCode, responseType, and responseContent. The two categories that concern this app are responseCode and responseContent. The responseCode gives the HTTP response status code indicating the success or failure of the transmission. This is handled by the procedureParseResponse. The responseContent is the category of interest because it carries the AGCO API server response. When the server is sending a response for the registration, it sends two items. The first item identifies the response and contains the string "registration." The second item contains the key. This key is then handed to the procedureGotValue_RegisterKey.

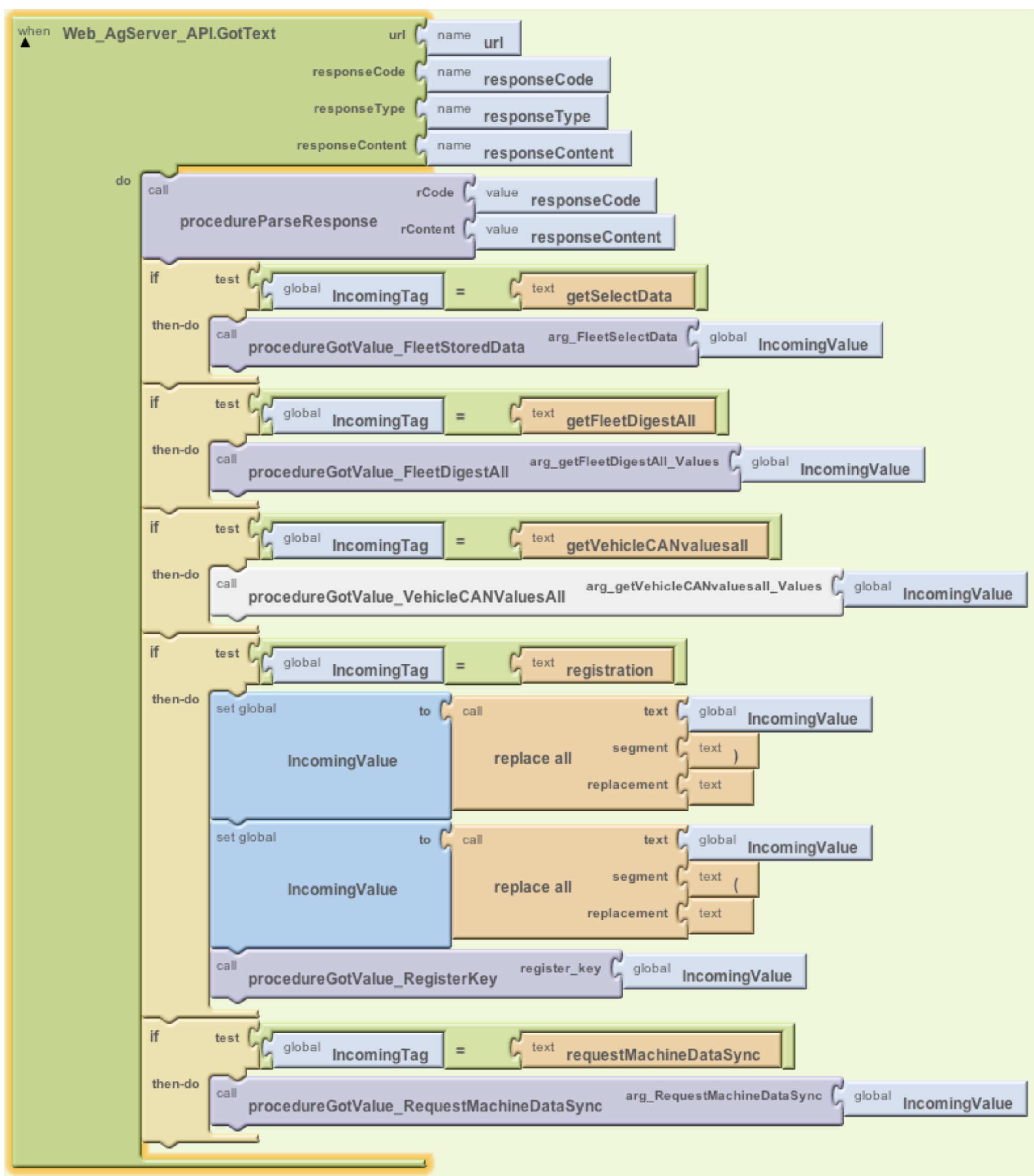



**Figure 4.12. Event block that handles web responses.**

The procedureParseResponse, shown in figure 4.12, tests the value of the responseCode and then decides if the responseContent is valid. Valid responseContent is given if the responseCode has a value of 200. It is invalid otherwise.

Valid responseContent is data separated by commas in JSON format. AI provides a block for decoding JSON text and for splitting the data into a list. After splitting the data, the first item in

the list is designated as the "IncomingTag" and then removed from the list. This "IncomingTag" is later compared in the Web_AgServer_API.GotText block. This determines how the "IncomingValue" list is handled.

A responseCode has a value of 500 when there is a server error. This case is handled by the procedure_Home_Display. If the value is not 500, nothing would happen. In either case the responseContent is not handled.

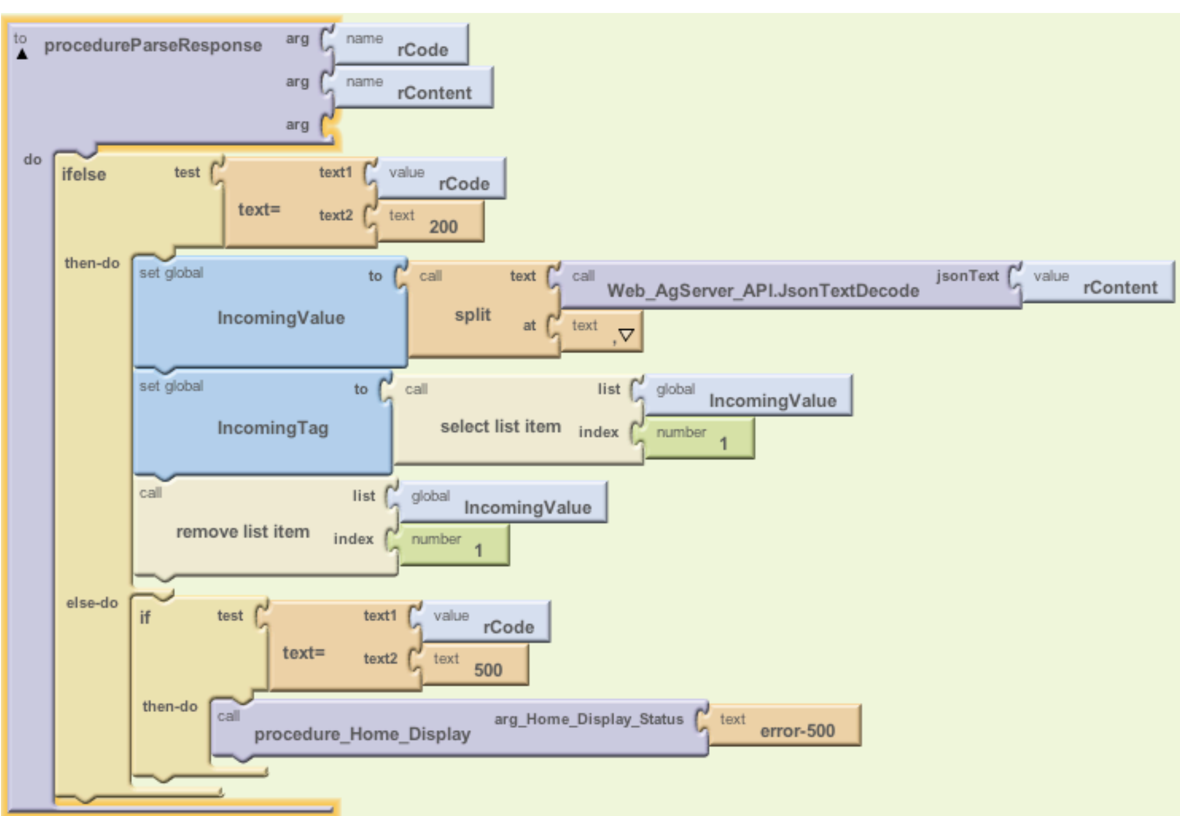



**Figure 4.13. Procedure block called in the web handler that handles code parsing.**

The procedureGotValue_RegisterKey block, shown in figure 4.13, stores the key received from the AGCO API server on the phone's database, and then calls the procedure_App_Initialization. Once the key is stored, the app no longer is limited to staying on the login/register page and can query AGCO for machine data.

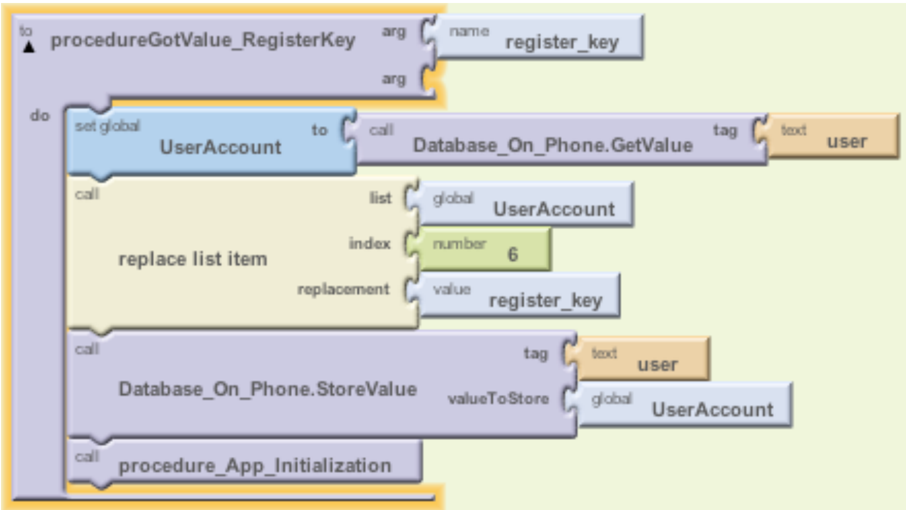



**Figure 4.14. Procedure block that stores the register key received by the Hesston server response.**

### *Requesting, Receiving, and Handling Machine Data*

Requesting machine data from AGCO can be handled in two steps. The first step requires the AGCO API server to aggregate and store data for a certain fleet. The second step requests to retrieve the stored data. These two steps give the AGCO API server enough time to perform both functions. This avoids the server from declaring timeout and aborting the task.

The first request is procedure_RequestMachineDataSync and is shown in figure 4.14. This request implements what is defined in the "AGCO Machine API" for aggregating machine data for a fleet. This request uses the key and the fleet name to designate the user and fleet to store data for. This is the only request that uses the fleet name since later calls for the stored data assume the same fleet.

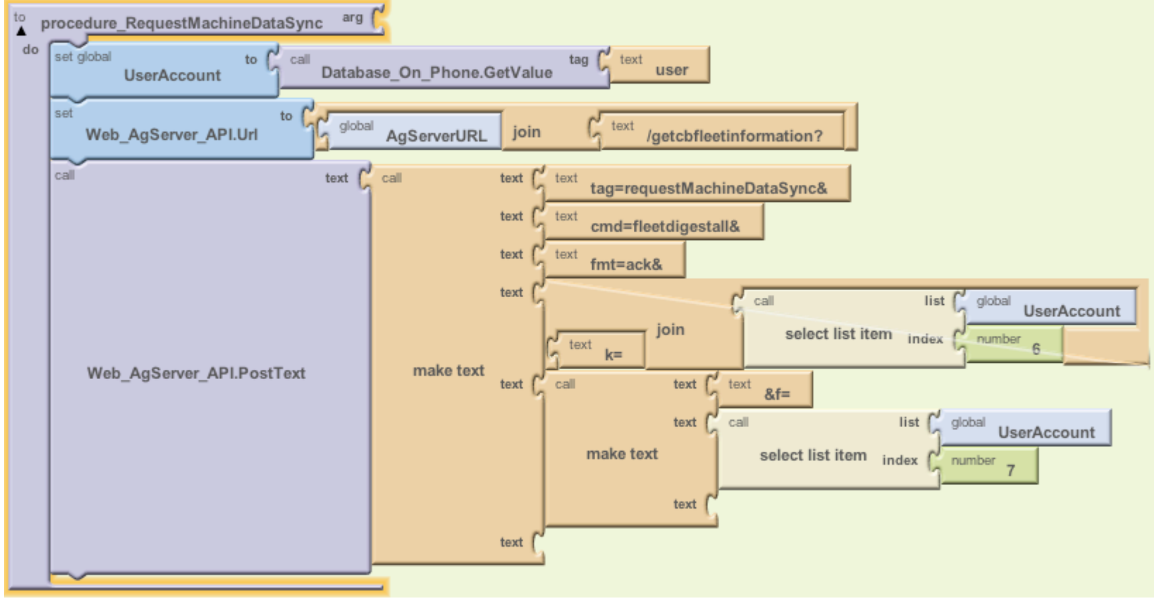



**Figure 4.15. Procedure block that requests the Hesston server to sync machine data for a fleet.**

The request procedure_GetFleetStoredData, shown in figure 4.15, is shorter than the request to aggregate machine data by one parameter. The AGCO API server still needs to know who it is communicating with but already has stored data and is ready to deliver it. That is the reason for the inclusion of the key and the exclusion of the fleet name. Future changes in the API could dictate a need for including the fleet name because a user could potentially request to aggregate data for multiple fleets and then request their retrieval from the same app.

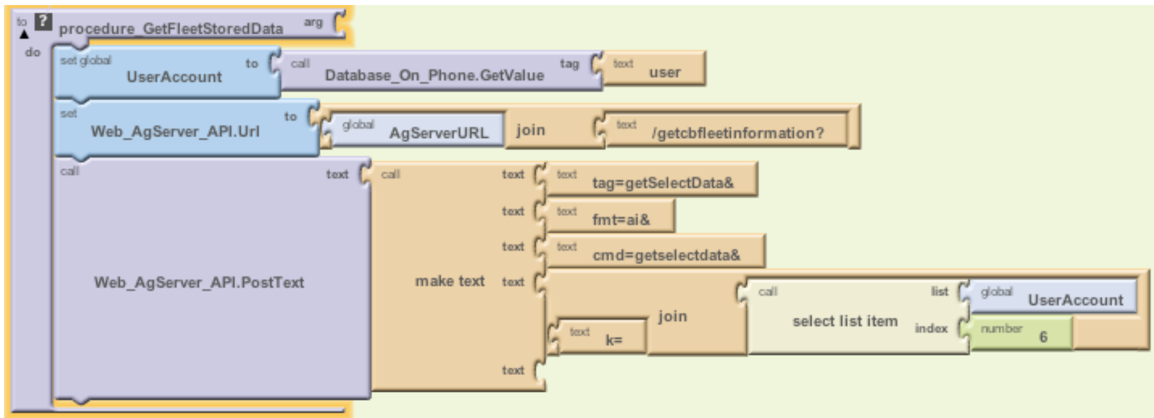



**Figure 4.16. Procedure block that posts request to Hesston server for stored machine data for a fleet.**

The Web_AgServer_API.GotText block handles the responses, from both of the requests above. After the procedureParseResponse, the response is handed to specific functions, based on the "IncomingTag" or first value in the response, that handle the data.

The request from procedure_RequestMachineDataSync is given a response from the Hesston server with the "IncomingTag" having a string value "requestMachineDataSync." When the Web_AgServer_API.GotText block receives this string, it hands the "IncomingValue" to the procedureGotValue_RequestMachineDataSync, shown in figure 4.16. The data contained in the list "IncomingValue" has only one item that describes the success or failure of the server to perform the request. This item can be "LoginFailed," "MachineSyncSuccess," or "NoVehiclesFound." Upon determining what the item is, the procedure alters the text so that the user can see the status of the data request.

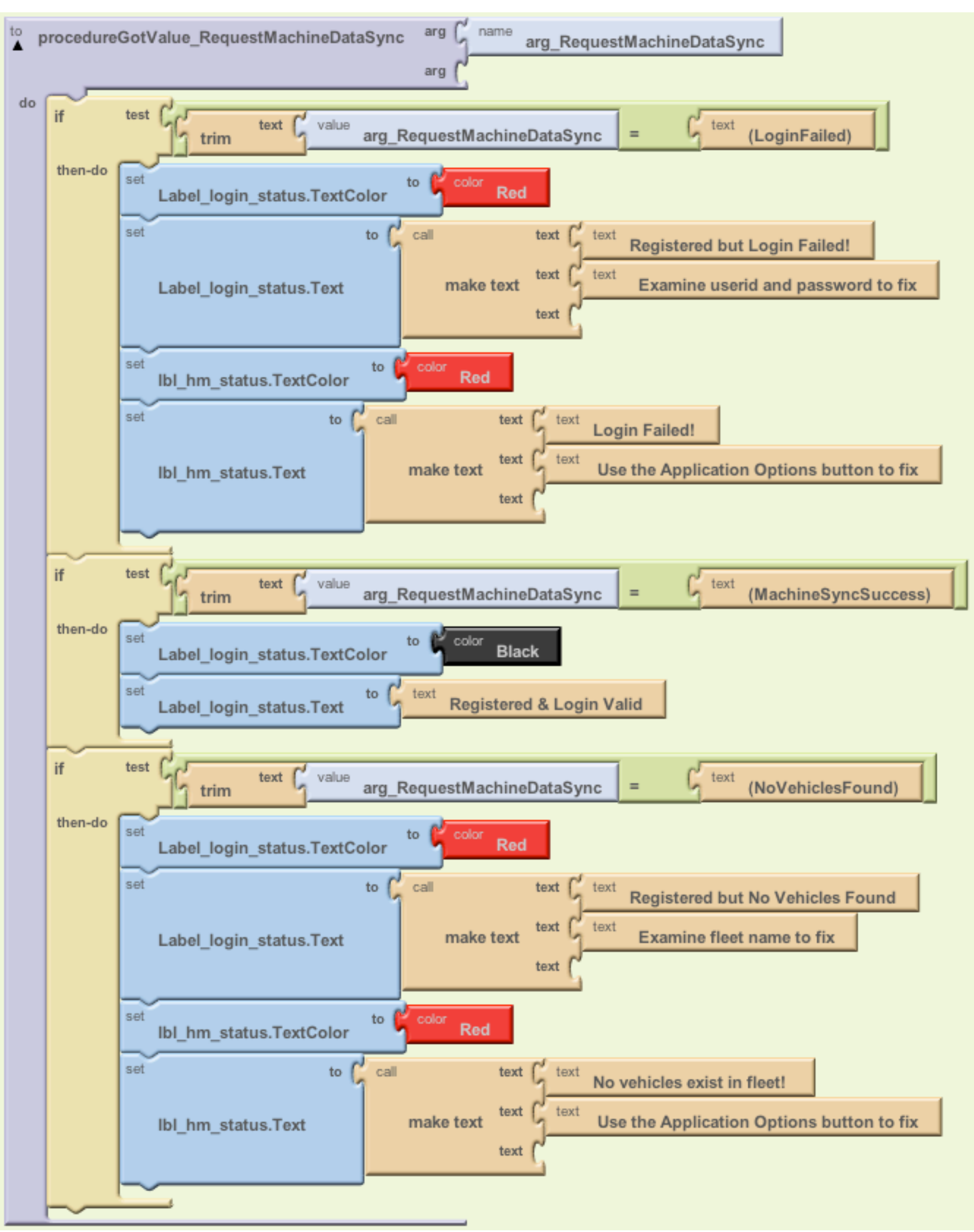



**Figure 4.17. Procedure block that handles response from Hesston server that confirms the success or failure of the request to sync machine data for a fleet.**

The request from the procedure_GetFleetStoredData is given a response from the Hesston server with the "IncomingTag" having a string value "getSelectData." When the

Web_AgServer_API.GotText block receives this string it hands the "IncomingValue" to the procedureGotValue_FleetStoredData block. The "IncomingValue" is a list of items with a variable length. The first item that is always delivered is the item describing the fleet size. When no fleet is available, the shortest list occurs and the fleet size is "-1." This value shows the user that the Hesston server has not yet stored data for the requested fleet. The user is alerted to this by a label change on the home page. The data given for one machine with this request is delivered as a package of 27 items with the fleet size of "1".

After the fleet size, the following items are delivered as shown in Table 4.4, below.

**Table 3.4. AGCO Machine Data**

| | List Name | List Value for Index = 1 |
|---|---|---|
| 1 | unitFleet_Col | 2013 Test Fleet |
| 2 | unitName_Col | (2013) S77.18 (2793) |
| 3 | unitID_Col | 340 |
| 4 | unitType_Col | Harvester (Combine) |
| 5 | unitOpPhone_Col | 6205042217 |
| 6 | unitLat_Col | 45.6463432 |
| 7 | unitLong_Col | -119.4072952 |
| 8 | unitDateTime_Col | 07/11/2013 09:06:27 PM EDT |
| 9 | unitSpeed_Col | 0 mph |
| 10 | unitAlarm_Col | FALSE |
| 11 | unitEngHr_Col | 921.35 hr |
| 12 | unitEngSpeed_Col | 0 rpm |
| 13 | unitStatus_Col | OFF/Parked |
| 14 | unit_ThresherHr_Col | 585.54 hr |
| 15 | unit_FuelLvl_Col | 0.66 |
| 16 | unitFuelRate_Col | 0 gal/hr |
| 17 | unit_Weather_Col | Hermiston, Herm.. OR (11 Jul 7:53 pm PDT ): 84F [55F/82F] W: 10mph RelH: 16 |
| 18 | unit_baleWeight_Col | No Data |
| 19 | unit_yieldAve_Col | No Data |
| 20 | unit_yieldMoisture_Col | No Data |

| | | |
|---|---|---|
| 21 | unit_baleMoisture_Col | No Data |
| 22 | unit_baleTotalCnt_Col | No Data |
| 23 | unit_baleCnt_Col | No Data |
| 24 | unit_engineLoad_Col | No Data |
| 25 | unit_lossRotor_Col | No Data |
| 26 | unit_lossShoe_Col | No Data |
| 27 | unit_capacityAve_Col | No Data |

These items are sorted in the list "IncomingValue" in a manner that places data of a certain type together and indexed to match the machine. This is discussed in fuller detail in the API section.

The apps task of ordering this data into lists is accomplished with the FleetSize block, the procedure_ClearColSelectData block, a "for range" block, and the procedure_LoadColSelectData. The procedure_ClearColSelectData clears all previous data held in these lists. The procedure_LoadColSelectData is ran each time the "for range" block informs it to run. The "for range" block is a for loop, incrementing an index by one each time it runs the procedure_LoadColSelectData block until the index reaches a determined quantity that is the value of FleetSize here.

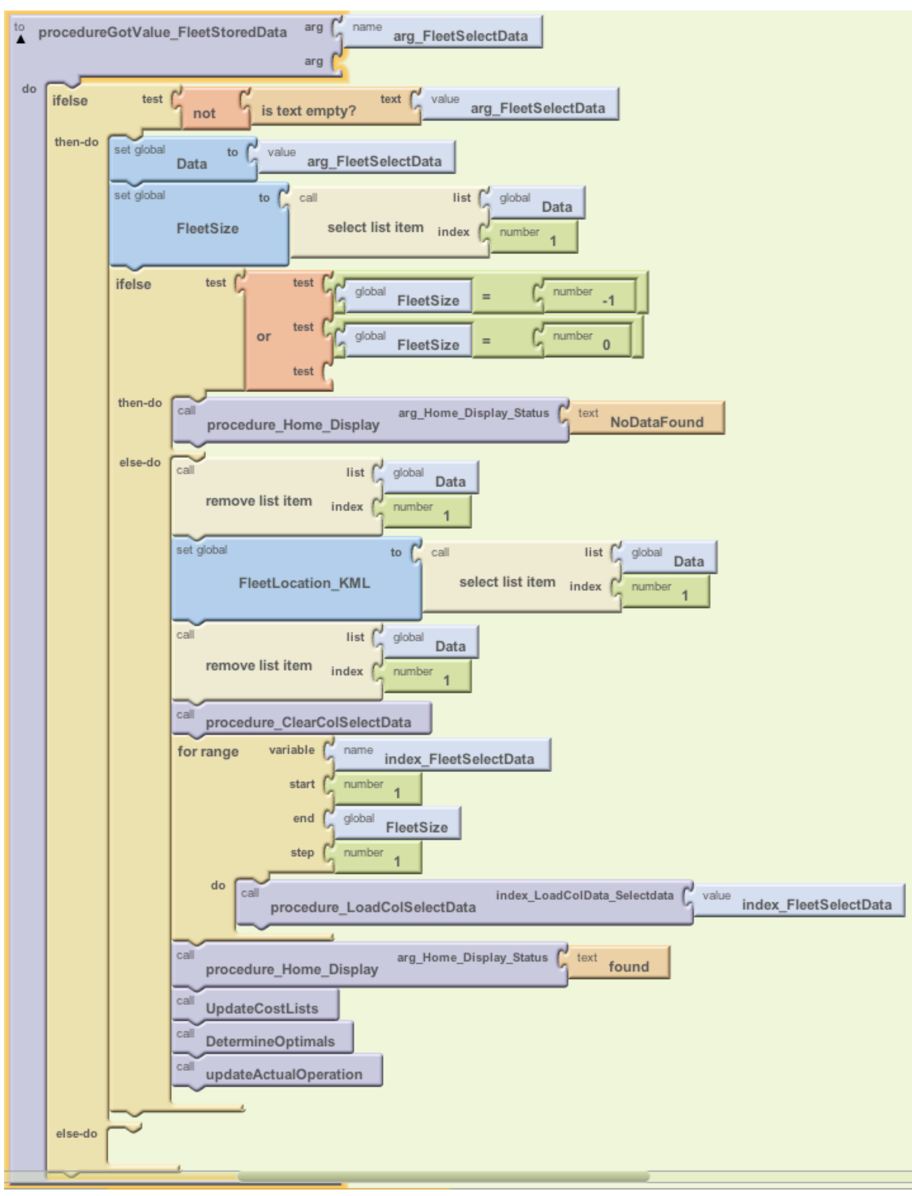



**Figure 4.18. Procedure block handling the Hesston server's response containing machine data for a fleet.**

The procedure_ClearColSelectData, shown in figure 4.18, clears the 27 lists that hold the data for the app. It is important to clear all of the data to prevent old data from being presented with new data.

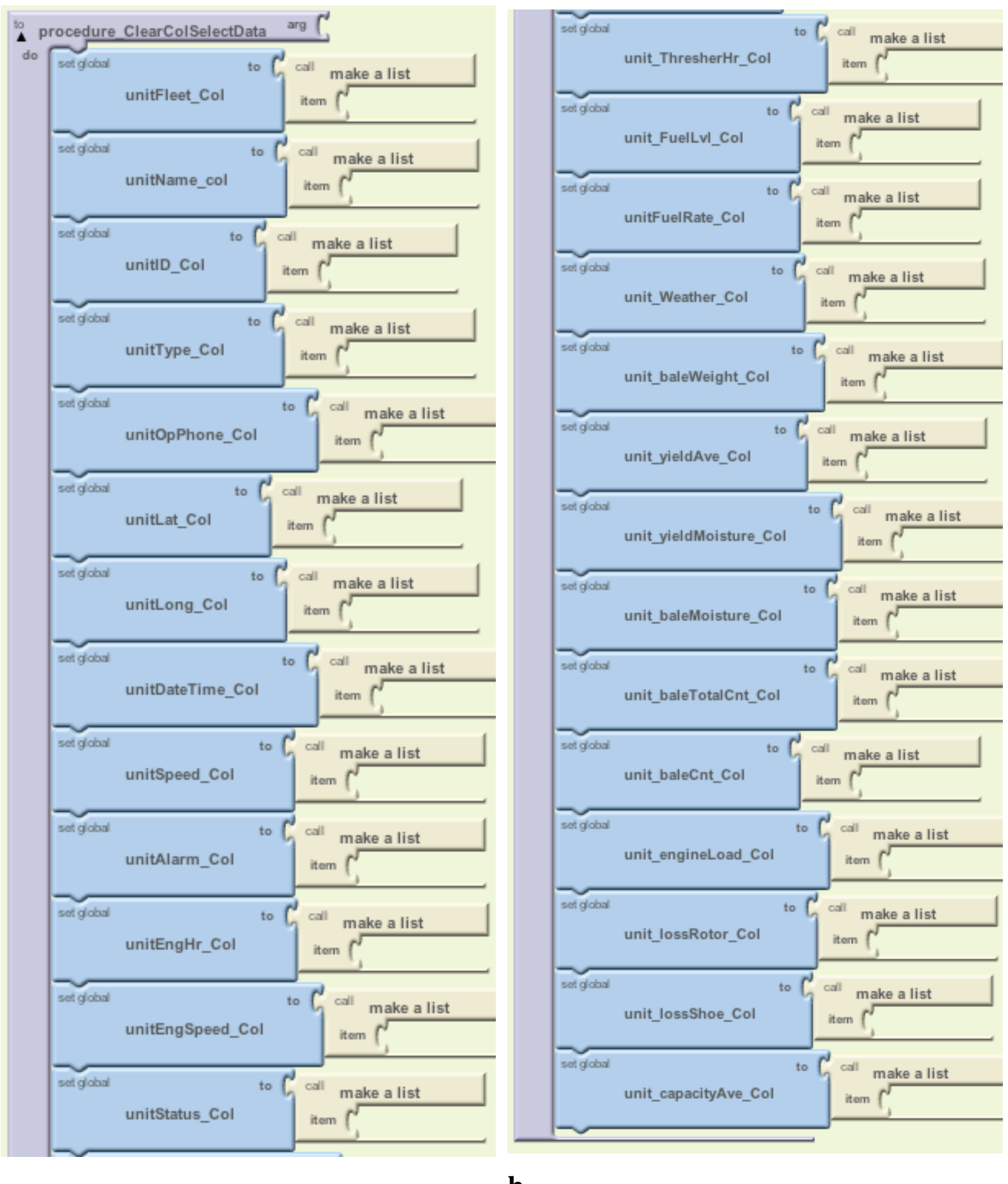



**Figure 4.19. Procedure block that clears the fleet lists for machine data.**

The procedure_LoadColSelectData, shown in figures 4.19 to 4.22, take the data from a single list and transform it to 27 lists. If new data types were to be added to into a machine data response, a new list could be easily added with its unique index for type of data. The start index for type of data is zero. This procedure is the final one for transferring the data from the Hesston server to the app.

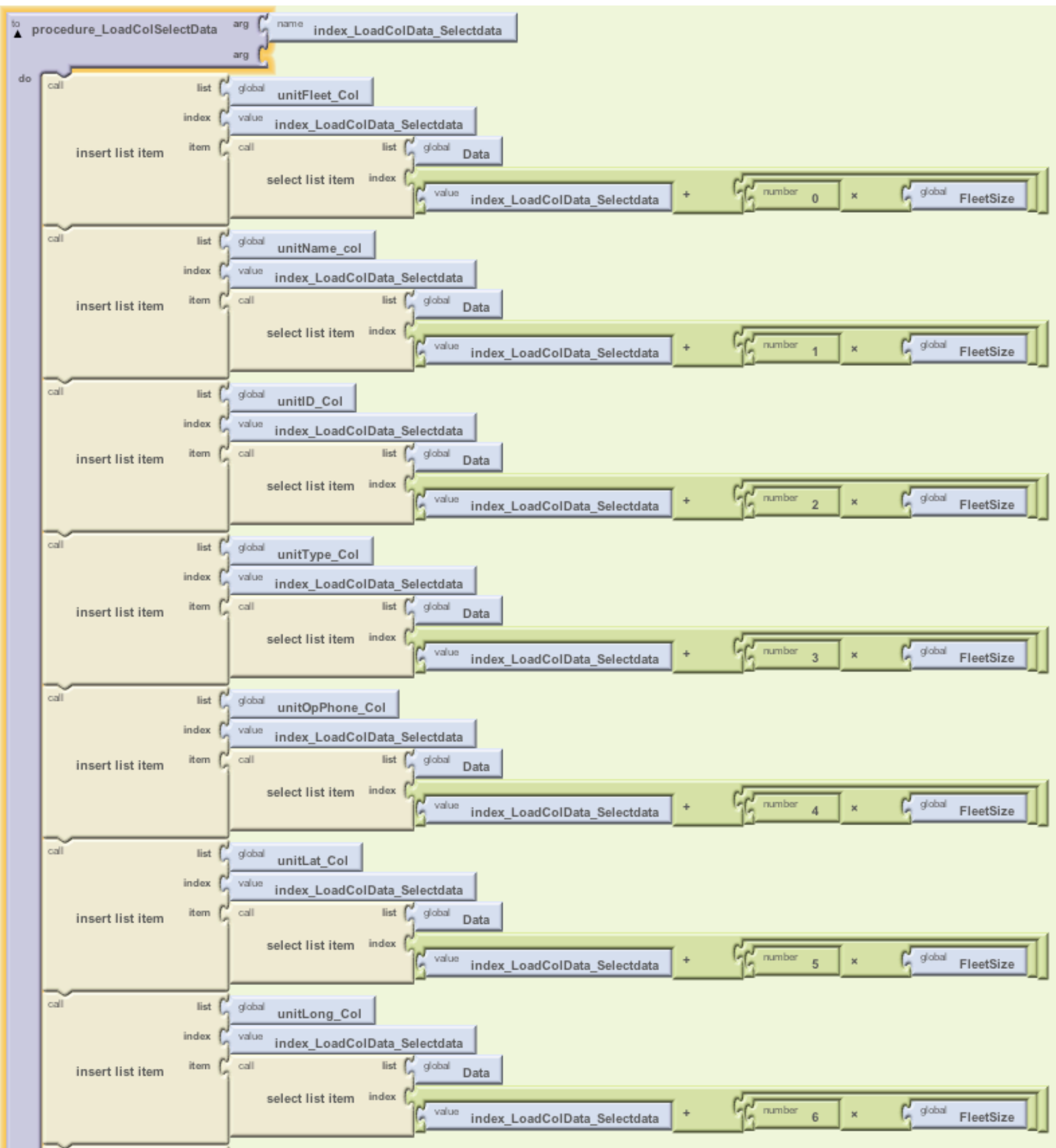



**Figure 4.20. Procedure block that loads machine data into lists. The data for a machine has an index and is organized accordingly in the lists.**

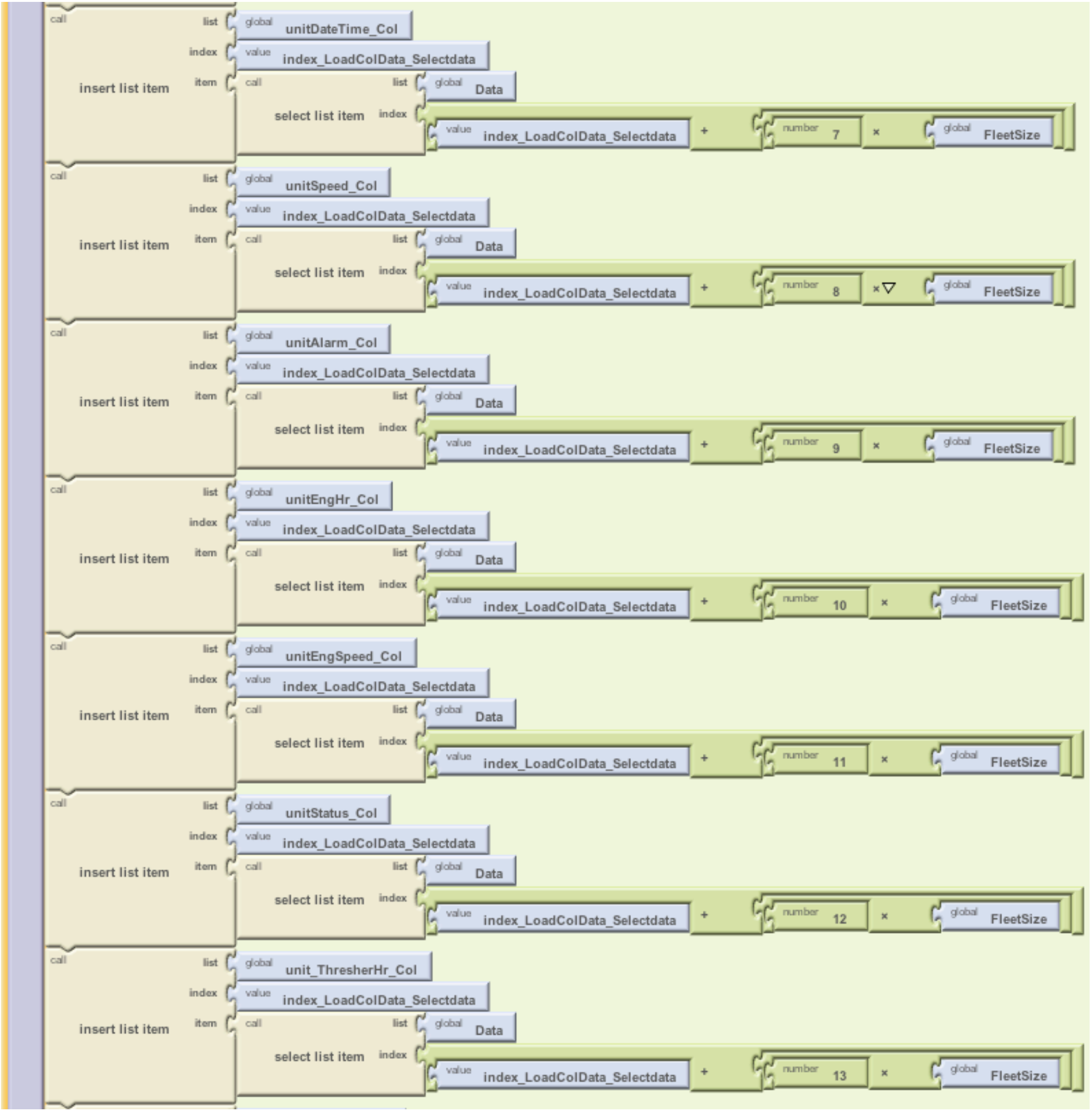



**Figure 4.21. Continuation of block loading machine data into lists.**

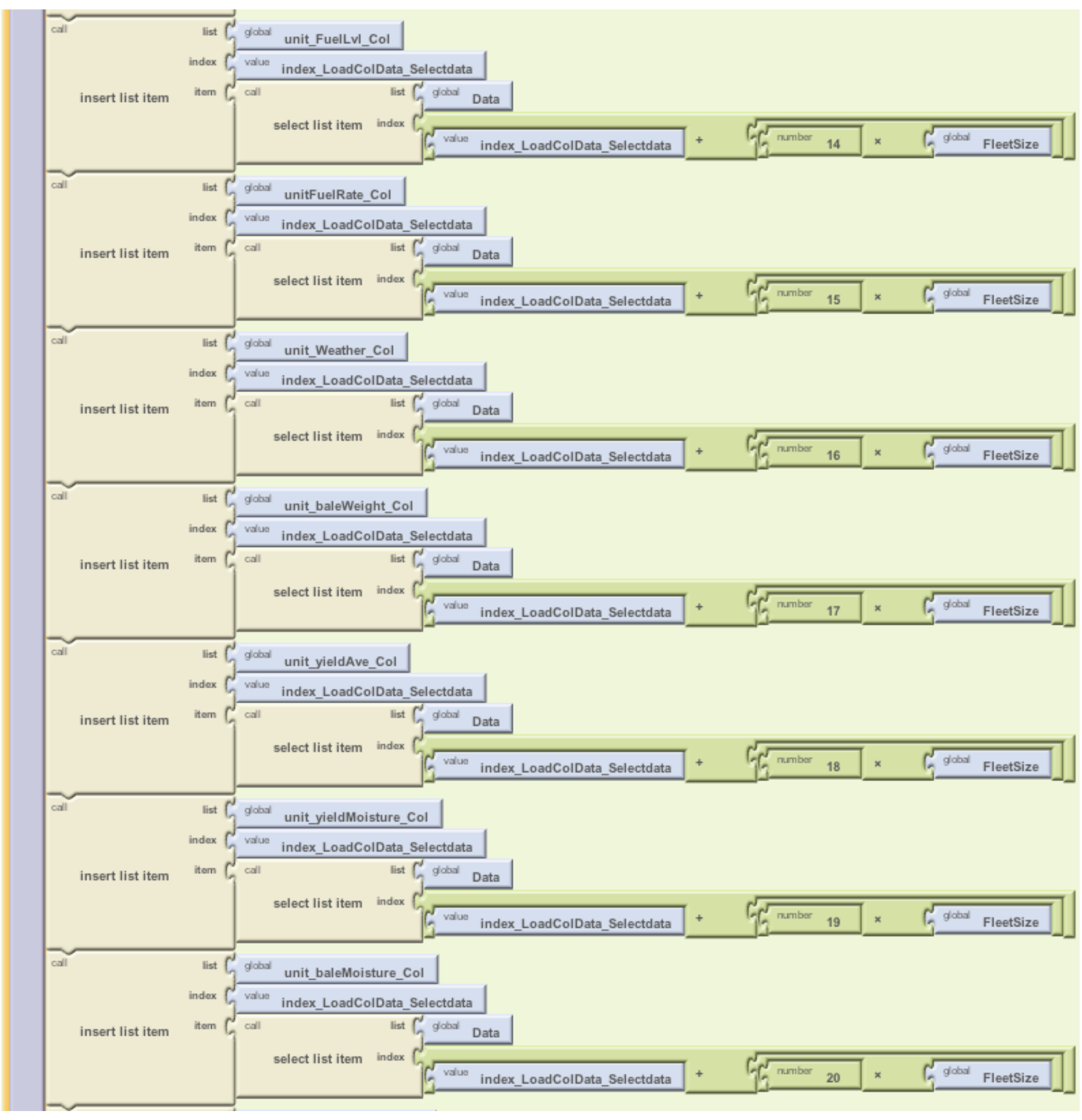



**Figure 4.22. Continuation of block loading machine data into lists.**

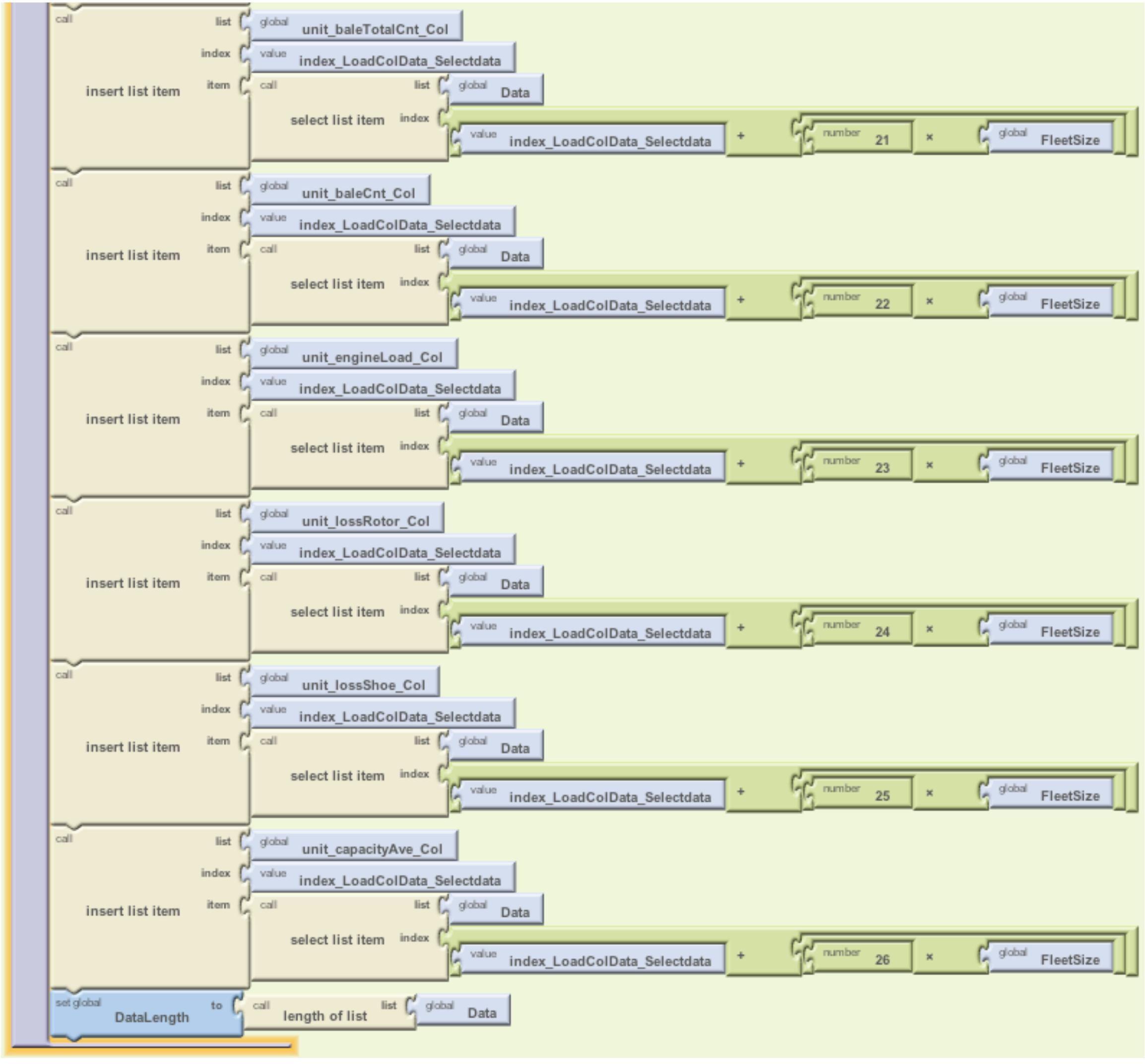
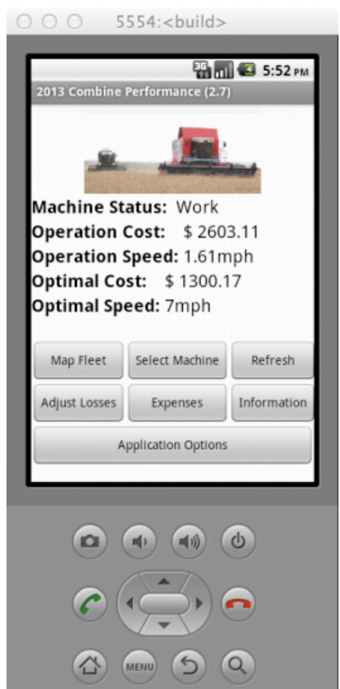



**Figure 4.23. Continuation of block loading machine data into lists.**

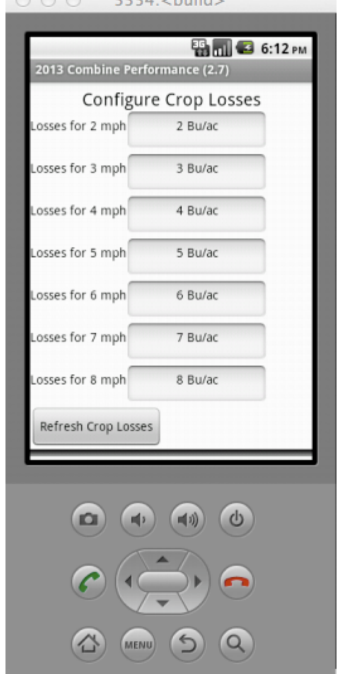
### *Changing Fuel and Crop Losses*

The farmer's knowledge is important for executing a correct model. The farmer can change the values for crop losses and fuel consumption at speeds from 2 to 8 mph. This range was selected as a base for most operating conditions and is changeable in the blocks.

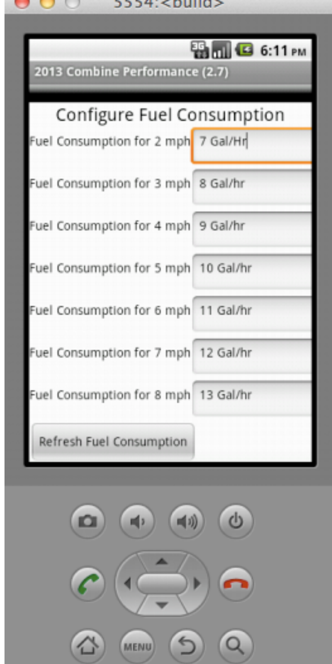
From the homepage, the user can change these values by clicking the button with "Adjust Losses" shown in figure 4.23a. This triggers a change in the window viewed that allows the user to scroll to the crop losses view or the fuel consumption view shown in figures 4.23b and 4.23c respectively.




a. Home View




b. Machine View – Crop Losses




c. Machine View – Fuel Consumption

**Figure 4.24. Clicking the "Adjust Losses" button in the Home View allows the user to change values for the Crop Losses and Fuel Consumption.**

The block code that controls this behavior is an event block called btn_Hm_Configure.Click, shown in figure 4.24. This block calls the WindowToDisplay with the argument "configLosses." This changes the visibility properties of components, resulting in the display in figures 4.23b and 4.23c.

**Figure 4.25. Event block occurring when the button labeled "Adjust Losses" is clicked in the Home View. This allows the user to change values for the Crop Losses and Fuel Consumption.**

The user changes the values by selecting a textbox, inputting a new value, and then pressing the refresh button. These three steps are described below.

Selecting a textbox clears its contents, after which a user is restricted to inputting number values. The figures 4.25a and 4.25b, show event blocks for selecting a textbox for the Fuel Consumption and for the Field Loss. Both the Fuel Consumption and the Field Loss require seven of these event blocks for selecting a textbox.

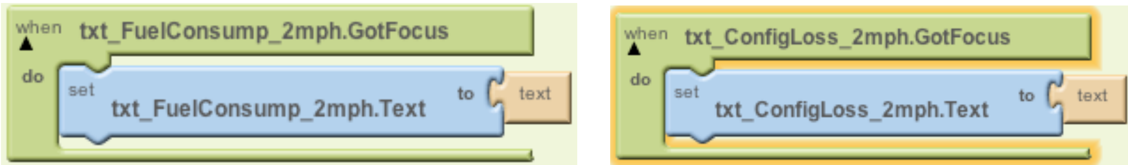
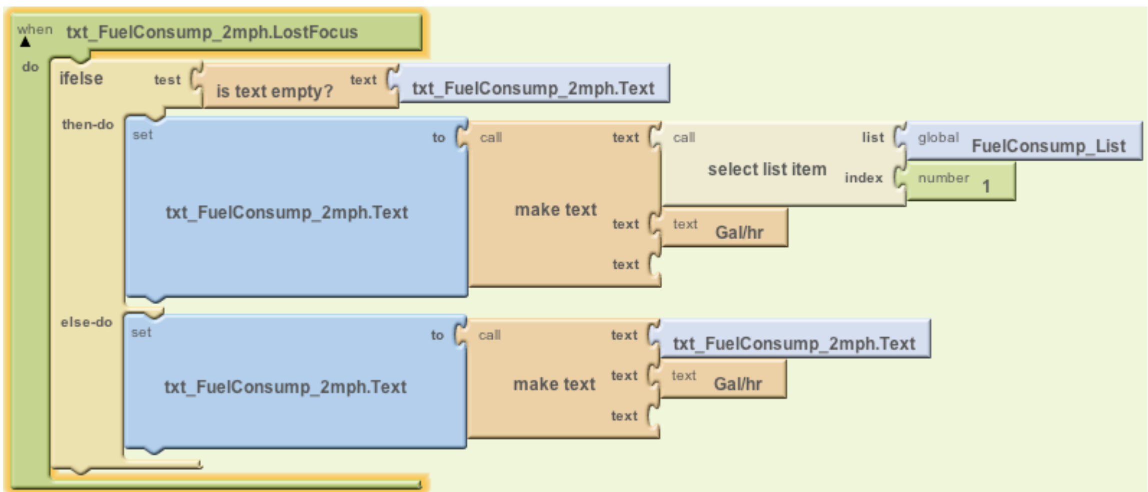



**Figure 4.26. Event blocks occurring when the textbox has the focus or is selected. This event clears the textbox contents.**

The blocks in figures 4.26a and 4.26b are event blocks that happen after their specific textbox has lost focus. These blocks dictate that once their textbox has lost focus, the textbox's contents will change to show the new value with the unit or the old value with the unit if a new value has not been input.

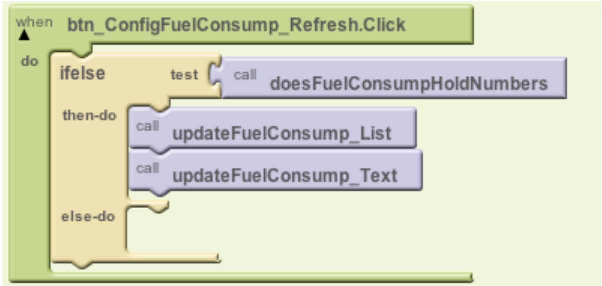



**a**

**b**

**Figure 4.27. Event blocks occurring when the textbox has lost focus.**

Pressing the refresh button saves the value to a list and displays the value with its unit, such as Bu/ac, in the textbox. The refresh button block codes controlling these behaviors are shown in figures 4.27a and 4.27b. However, pressing the refresh button does not make a textbox lose focus, providing a need for a number check.



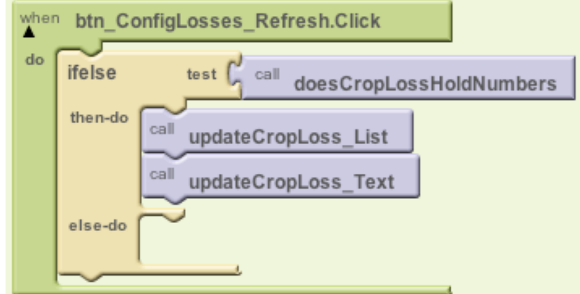
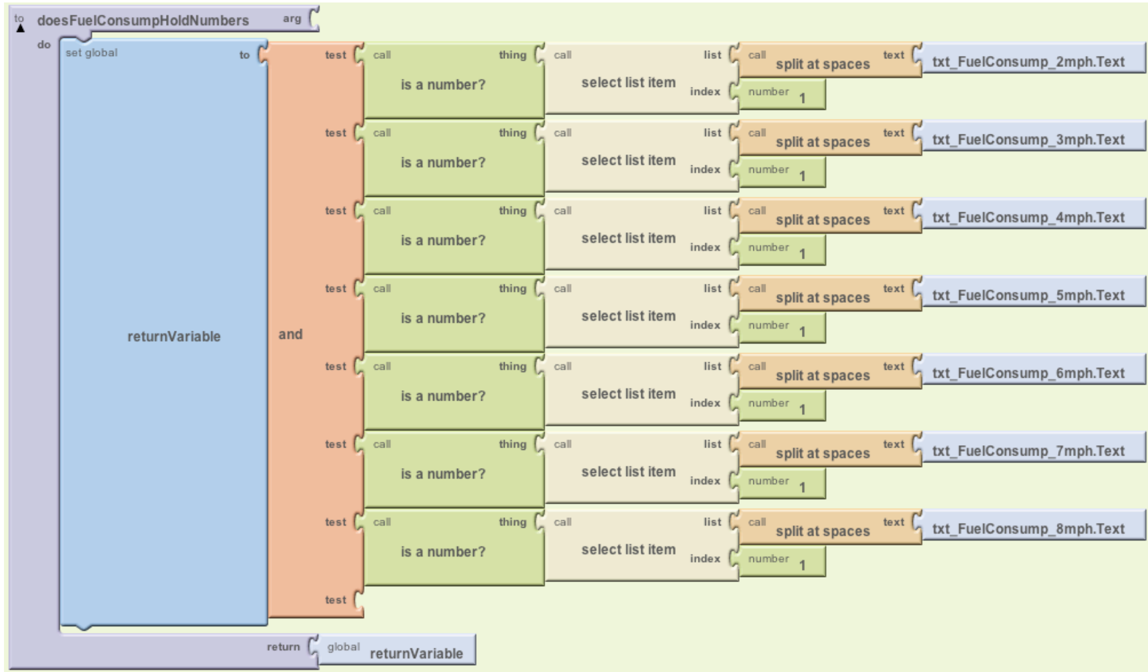
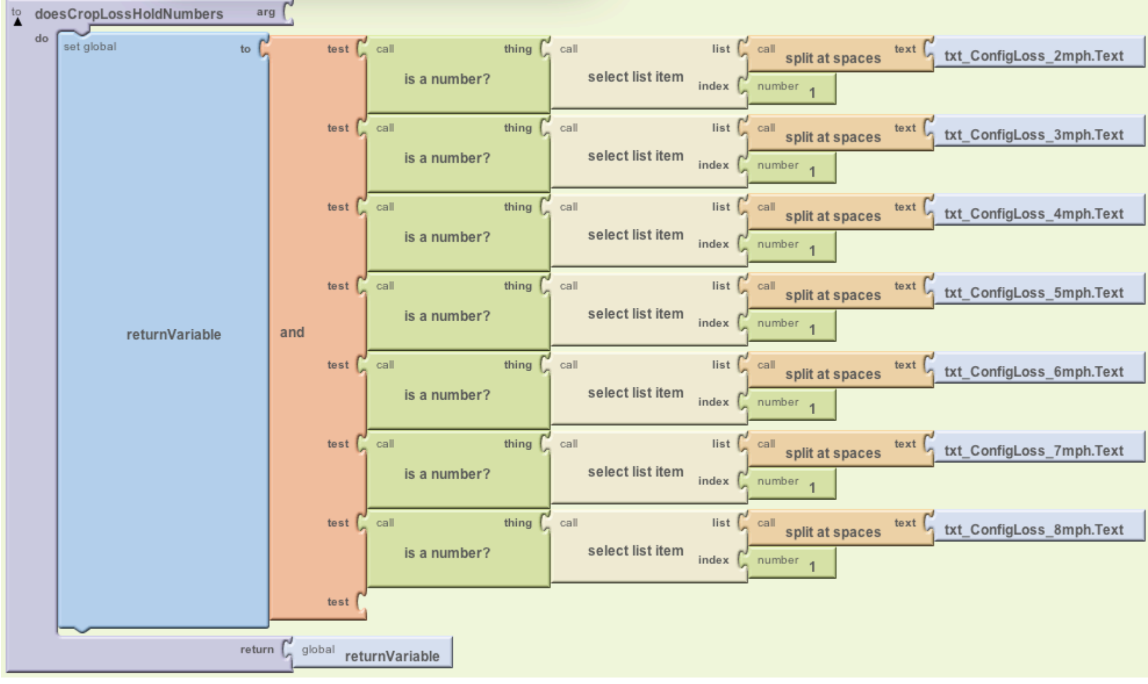




**Figure 4.28. Event blocks occurring when the "Refresh" button is clicked for either the Crop Loss list or the Fuel Consumption list.**

The test within the refresh block codes returns true if a number is found and the app then performs the update blocks. The block codes for these tests are shown in the figures 4.28a and 4.28b. These tests are necessary because the model must use numbers. Without the test blocks, a user could input nothing into a textbox and then press the refresh button. This would result in an empty value for a list item and the textbox would then be changed to hold the unit such as "Bu/ac." If repeated the list item would hold a value of the unit and the textbox would change to "Bu/ac Bu/ac."



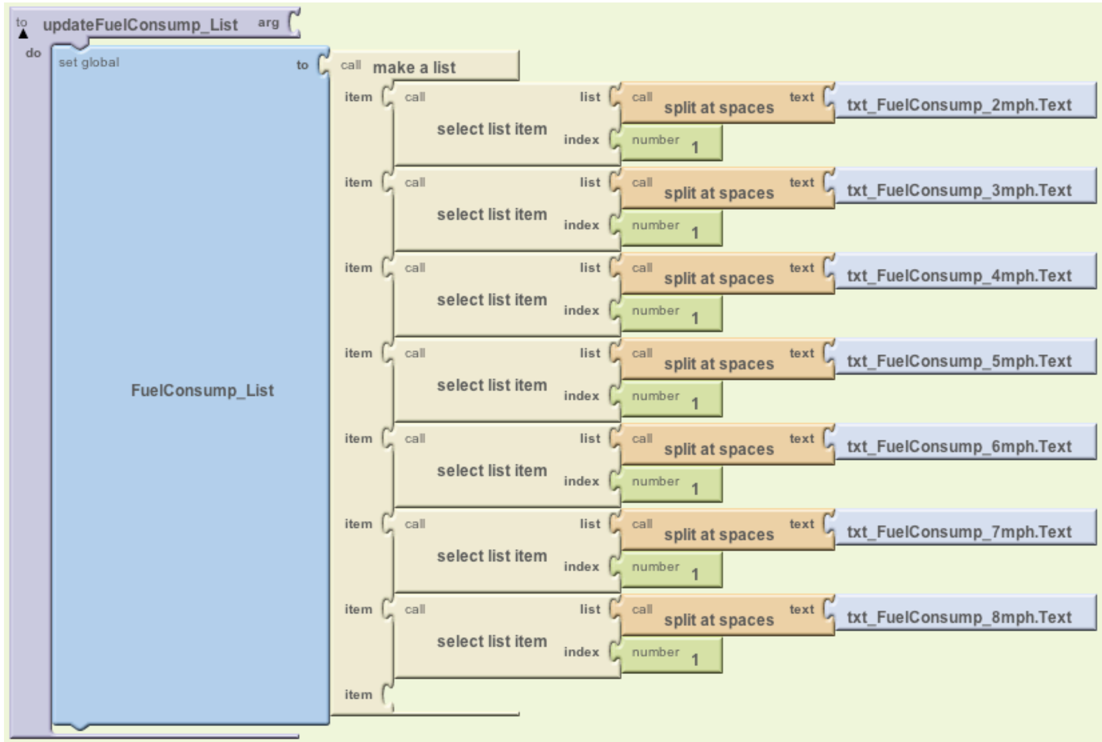

**a.**



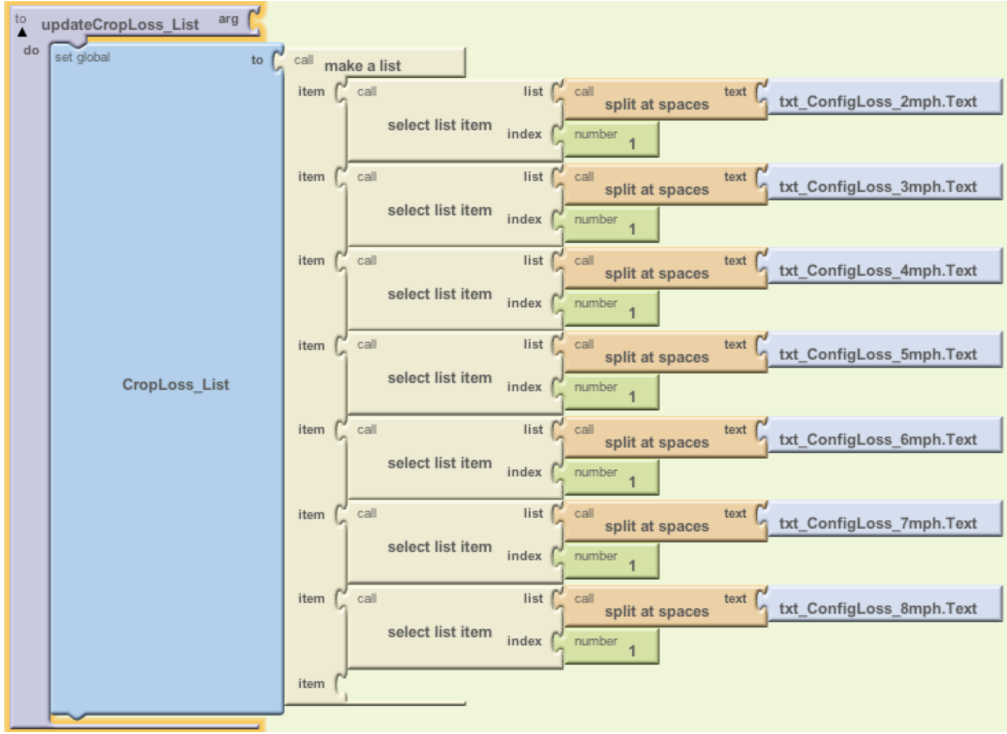

**b.**

**Figure 4.29. Procedure blocks that check for a valid entries in the Crop Loss and the Fuel Consumption lists.**

The block code updating the lists is shown in figures 4.29a and 4.29b. These blocks treat the textbox text as a list and select the first item that is the number value.



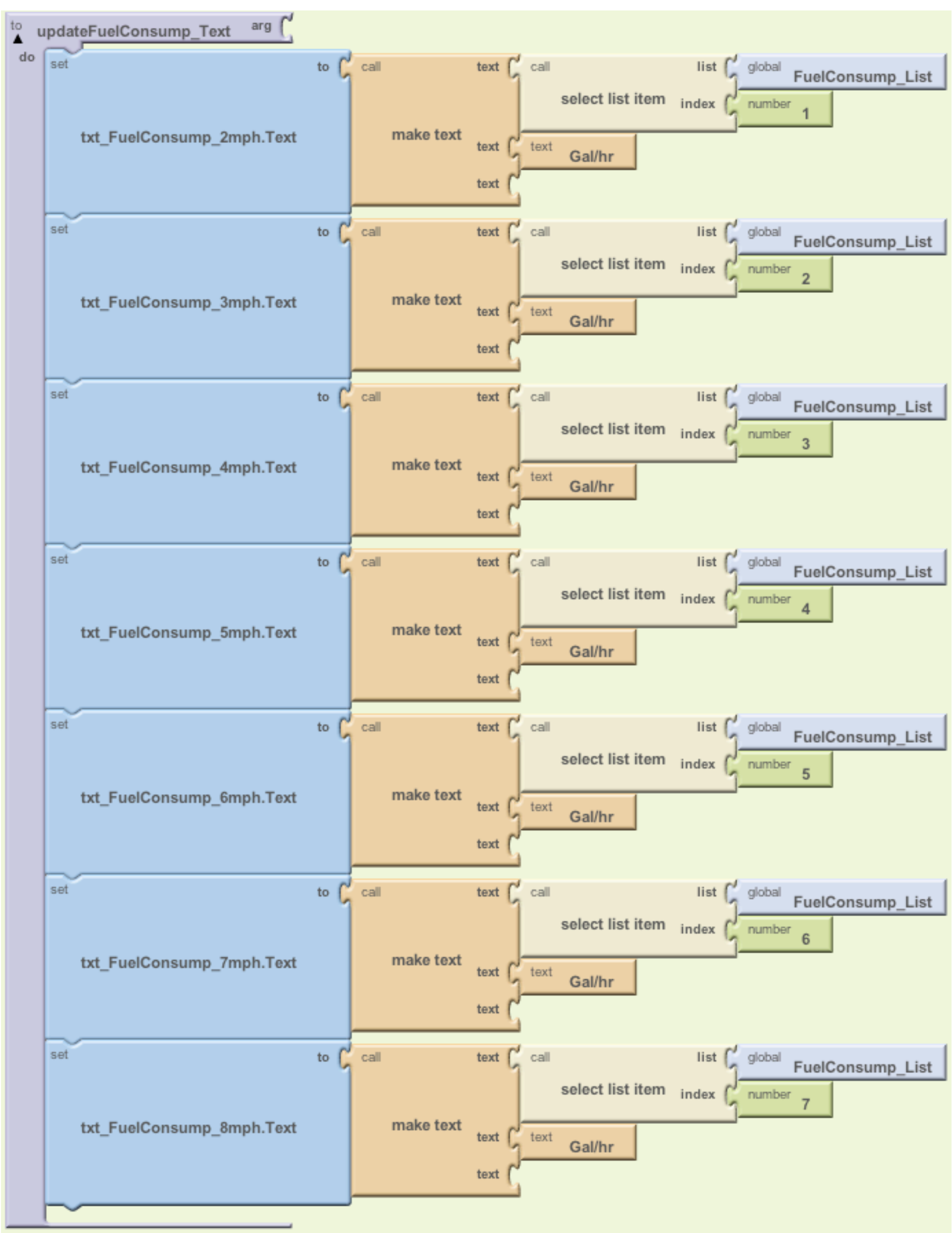

**a.**



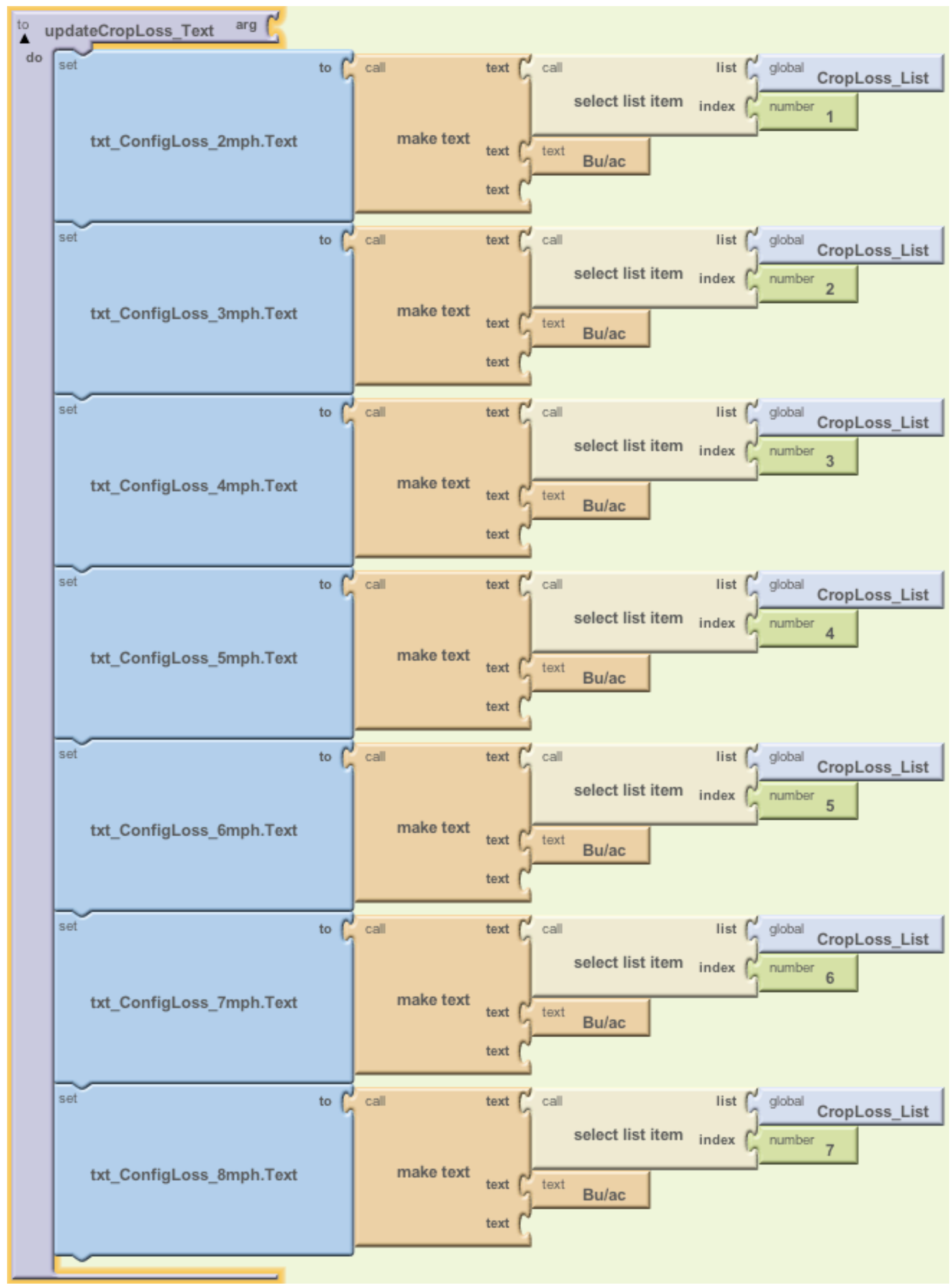

**b.**

**Figure 4.30. Procedure blocks that update the Crop Loss and Fuel Consumption lists.**

The blocks updating the textboxes text are shown in figures 4.30a and 4.30b. These blocks pull the value from the respective lists and then append a unit.




**a.**




**b.**

**Figure 4.31. Procedure blocks that update the textbox text for the Crop Loss and Fuel Consumption entries.**

### *Mashing Machine Data with Model*

The block code that manipulates the combine data to perform models is called after the data is loaded to lists in procedureGotValue_FleetStoredData . They are also called when a different combine is selected and when the user changes inputs. Figure 4.31 is the event block code that is activated after a machine has been picked and figure 4.32 is the event block that is activated when a button is clicked after inputs are changed.



**Figure 4.32. Event block occurring after a machine has been picked.**

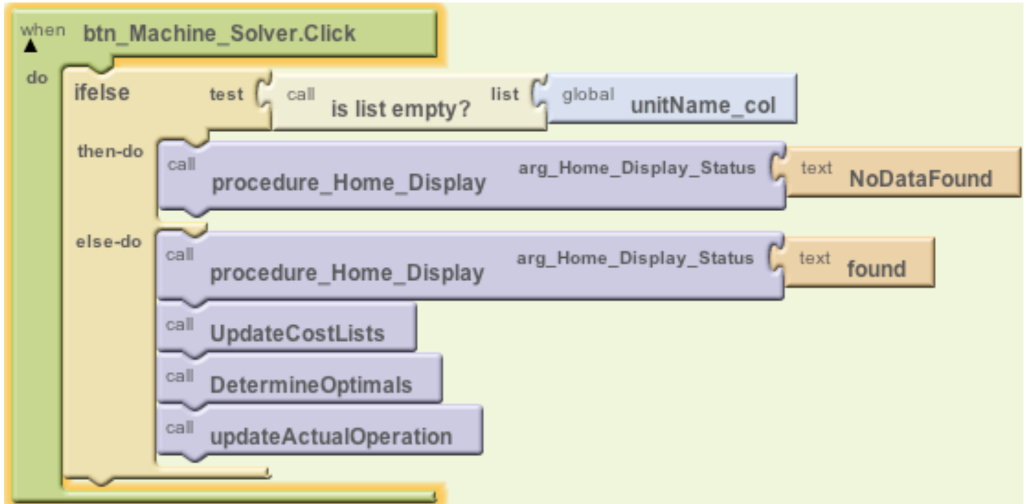



**Figure 4.33. Event block occurring at the click of the Solver button in the Machine View.**

The first procedure to be called from both of the event handler is the procedure to change the view to the home page. After this procedure, the event blocks above follow different paths to update the model. After selecting a machine from the listpicker, the app does not need to change any inputs other than the actual operation. Changing inputs such as fuel consumption, field losses, machine depreciation, and header width requires altering the model for calculating the optimal operating cost and speed. To alter the model two procedures are called. The first is

UpdateCostLists, shown in figure 4.33, and the second is DetermineOptimals, shown in figure 4.34.

The procedure UpdateCostLists renews the list items for five lists. These lists are Performance_List, MachineCost_List, CropLossCost_List, FuelCost_List, and TotalCostList. The “foreach” block performs its code on every “speed” variable in the “Speed_List.” This list has seven variables, speeds two to eight mph. The index variable sets the list position of the new items. It starts at one and ends at seven for the seven list items.

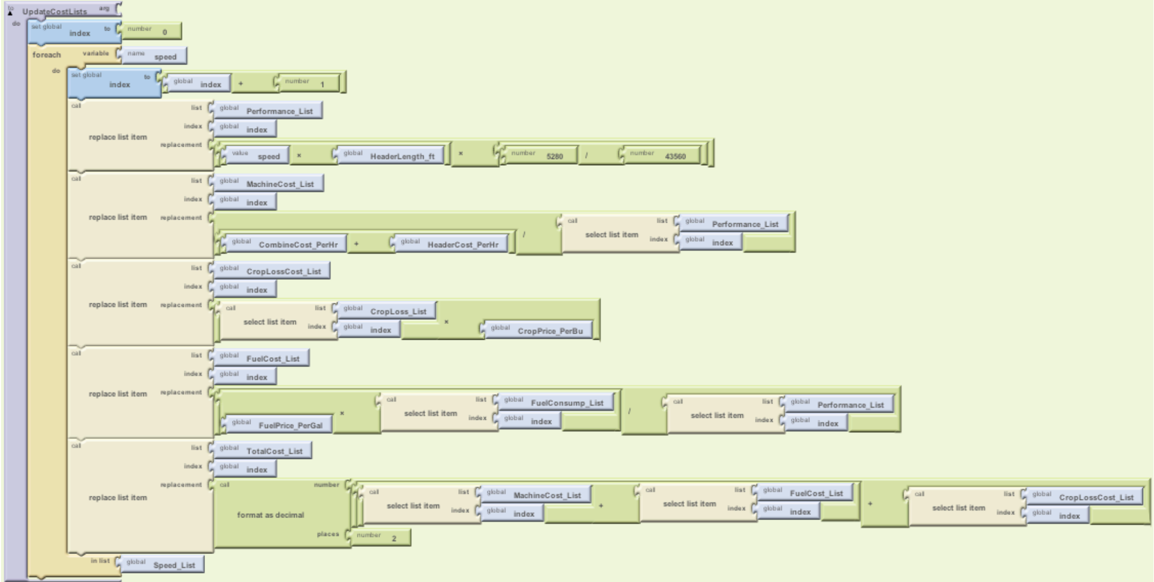



**Figure 4.34. Procedure block that updates lists for the model.**

The procedure DetermineOptimals uses the “min” block given by AI to find the lowest operating cost. The “position in list” block then identifies the index value for the lowest operating cost in the TotalCost_List and that index value is set to OptimalIndex. The OptimalSpeed and OptimalCost are subsequently chosen with the OptimalIndex from their respective lists, Speed_List and TotalCost_List.

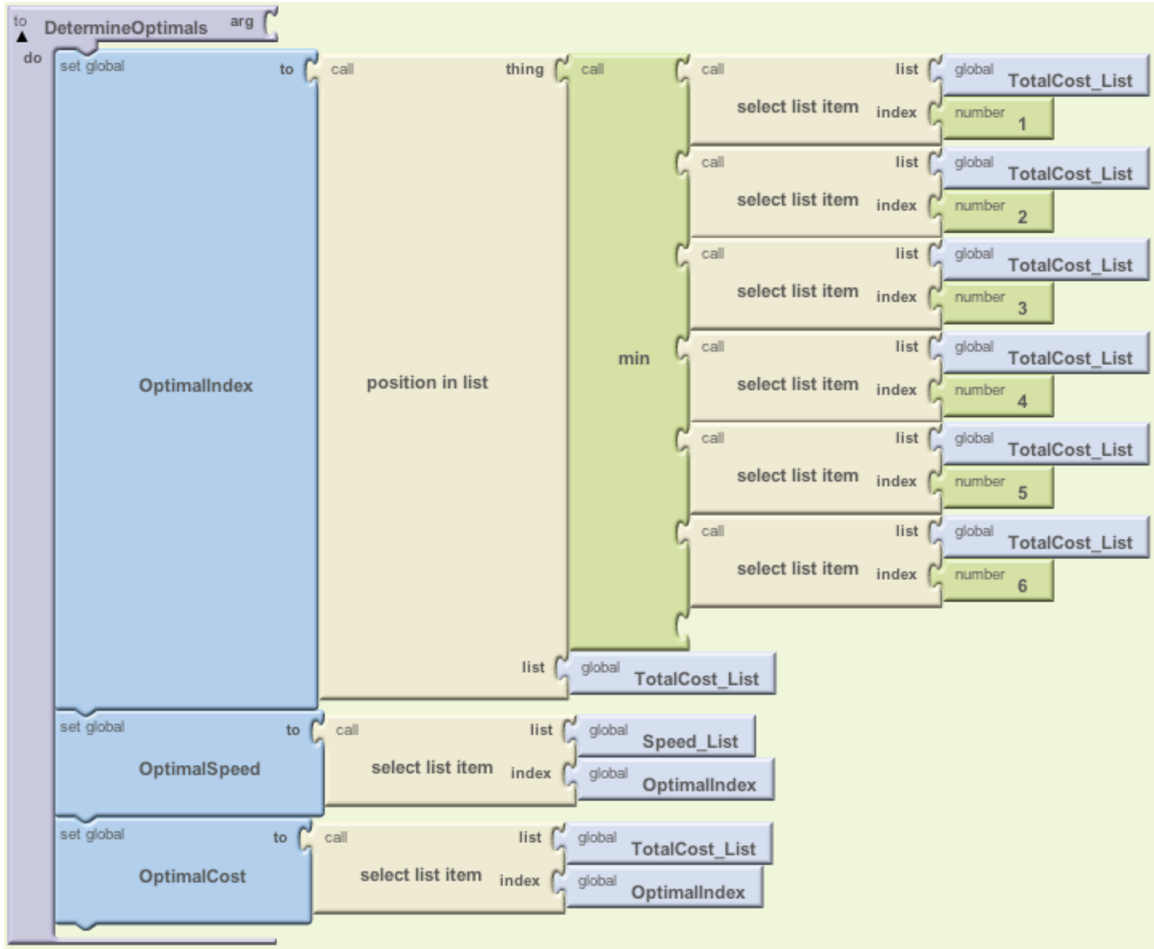



**Figure 4.35. Procedure block that determines the optimal speed and cost.**

The procedure updateActualOperation contains the block code that is responsible for changing the actual operation parameters to the latest data from the Hesston server and for deciding the text shown to the user. The largest determinant of what to show the user is the machine status. The status can be OFF/Parked, Idle, Work, Headland, Transport, and No Contact. The model created with this research is intended for only the Work status, but options remain to add new code to the other statuses. Regardless, the machine status is displayed on the app.

When a machine status is Work, this app rounds the machine's speed to the nearest integer and then checks that value to see if it exists on the Speed_List. If it does not exist on the list, the app sets a label to report to the user that it does not recognize the speed as a valid working speed. If it does exist on the list, the app then sets the actual operating cost to the amount corresponding to the item in the TotalCost_List with the index of the rounded ActualSpeed in the Speed_List. Labels are then set to show the user the actual operation and optimal cost and speed.

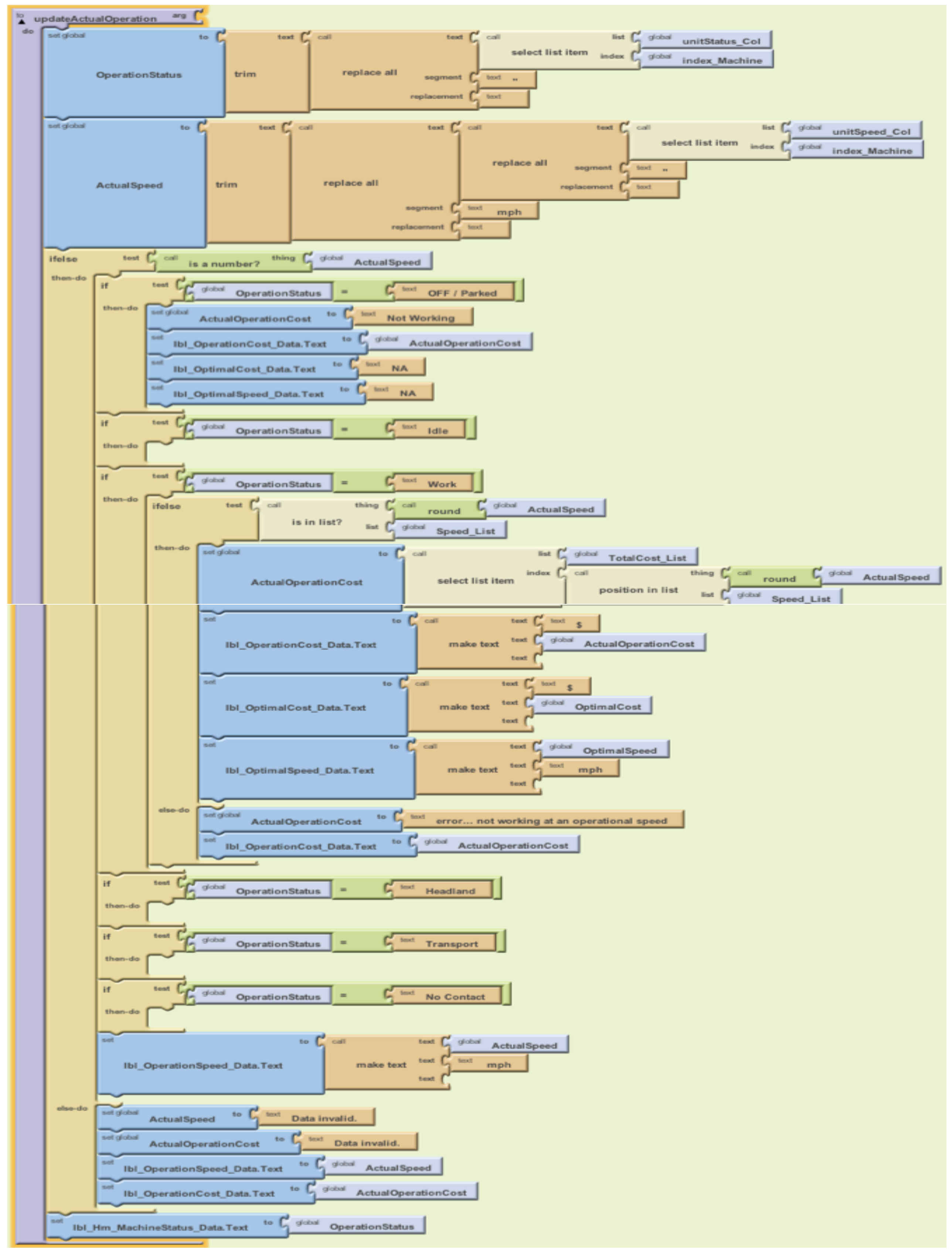



**Figure 4.36. Procedure block that sets the Home View labels based on the machine status.**

## Results

The blocks used for requesting and handling responses for the fleet data can be reused in other fleet management apps.  This app and others like it don't need all of the data that is given. However, the potential remains to expand this app's functionality and use all of the data.  This is countered by the noted difficulties that AI has with handling large files.

# Chapter 5 - Conclusions

Two apps were developed using MIT Application Inventor and Google App Engine with the server support from AGCO Corporation.

The first app's intent is to measure Growing Degree Days and has yet to be successfully implemented. Possibly two solutions exist. The first would change the app and not allow the user to set the planting date in the past. The second may not be possible but involves changing the calls accepted by the servers and allows the retrieval of historical weather data.

The second app determines the efficiency of combines that are harvesting based on their speed and input from an app user, preferably a farmer. This app retrieves a combine fleet's data from AGCO's servers, displays information for a selected machine, and determines the best speed and operating costs based on an app user's inputs.

These apps demonstrate the capability of sharing machine-generated data with independent app developers. MIT Application Inventor is a user-friendly development environment allowing a larger population of developers to implement their app ideas with less hassle. With AGCO's server support, developers can create apps specific to farmers and expand on the value of their machine data.

# Chapter 6 - Future Work Possibilities

The open source nature of App Inventor leaves opportunities for developing new apps using AGCO's API. These opportunities exist for developers and for AGCO. From this research, future developers have available a template app to help with their own apps development. AGCO has the possibility of expanding their API to accommodate other practices.

The Combine Efficiency App was stripped of its model components, leaving it with the components to retrieve data from the AGCO API and serving as a template for future apps using other developer's models. This template is available on AGCO's App Engine site. A QR code and URL for downloading the template app and its code are displayed in Appendix B.

The opportunity also exists for AGCO to expand its API for purposes other than combining. For example, data could be used for apps developed for planting, spraying, spreading, and other applications. This creates more opportunities and a stronger environment for developers to make apps.

References

AgCommand. 6/14/2013. Massey Ferguson. http://www.masseyferguson.com/EMEA/int-en/products/2847.aspx retrieved on June 17 2013.

API. TechTerms.com. http://www.techterms.com/definition/api retrieved on June 13, 2013.

Apple's App Store Marks Historic 50 Billionth Download. May 16, 2013. Apple Press Info. http://www.apple.com/pr/library/2013/05/16Apples-App-Store-Marks-Historic-50-Billionth-Download.html retrieved on July 2, 2013.

Bank of America Mobile Banking App. 2013. Bank of America. https://www.bankofamerica.com/online-banking/mobile-banking-applications.go retrieved on July 29, 2013.

Dvorak, Joseph S., Tanya C. Franke-Dvorak, Randy R. Price. December 2012. "Apps"-An Innovative Way to Share Extension Knowledge. Journal of Extension.

Google Play Developer Console. 2013. Google. https://play.google.com/apps/publish/signup/ retrieved on July 3, 2013.

Growers Edge. 2009. Growers Edge. https://www.growers-edge.com/howitworks-mobile-app.html#&panel1-10 retrieved on July 30, 2013.

HTTP Requests: Get vs Post. http://www.w3schools.com/tags/ref_httpmethods.asp retrieved on June 16, 2013.

Ingraham, Nathan. June 10, 2013. Apple announces 600 million iOS devices sold, 93 percent of devices running iOS 6. The Verge. http://www.theverge.com/2013/6/10/4415258/apple-announces-600-million-ios-devices-sold retrieved on June 2, 2013.

iOS Developer Program. 2013. Apple. https://developer.apple.com/programs/ios/ retrieved on July 3, 2013.

iOS Developer Program 3. Distribute. 2013. Apple. https://developer.apple.com/programs/ios/distribute.html retrieved on July 3, 2013.

Jones, Chuck. July 2, 2013. Apple and Android Trading Smartphone Market Shares in the Largest Markets. Forbes. http://www.forbes.com/sites/chuckjones/2013/07/02/apple-and-android-trading-market-shares-in-the-largest-markets/ retrieved on July 2, 2013.

Kansas Farmer. 2011. iNet Solutions Group. http://farmprogress.com/customPage.aspx?p=260 retrieved on August 1, 2013.

Kirk, Matthew, Julie Steele, Christèle Delbé, Laura Crow. Connected Agriculture: The role of mobile in driving efficiency and sustainability in the food and agriculture value chain. Vodafone, Accenture, Oxfam.

Kovach, Steve. June 4, 2013. 8 Great Apps For Managing Your Finances. Business Insider. http://www.businessinsider.com/best-financial-apps-2013-5?op=1 retrieved on July 29, 2013.

Lely International N.V. July 30, 2013. Lely Vector Control. Google Play. https://play.google.com/store/apps/details?id=com.lelycontroller retrieved on July 30, 2013.

Panzarino, Matthew. 15 May 2013. Google announces 900 million Android activations, 48 billion apps downloaded. The Next Web. http://thenextweb.com/google/2013/05/15/google-announces-900-million-activations-of-android-in-total-to-date/ retrieved on June 11, 2013.

Panzarino, Matthew. 16 May 2013. Billions: How exactly do Apple and Google count app downloads? The Next Web. http://thenextweb.com/apple/2013/05/16/billions-how-exactly-do-apple-and-google-count-app-downloads/ retrieved on June 11, 2013.

Samsung WatchON. July 31, 2013. Google Play. https://play.google.com/store/apps/details?id=tv.peel.samsung.app retrieved on August 1, 2013.

Siegler, MG. July 11 2010. Is Google App Inventor A Gateway Drug Or A Doomsday Device For Android? http://techcrunch.com/2010/07/11/google-app-inventor/ retrieved June 11 2013.

Smith, Aaron. July 11, 2011. Overview of smartphone adoption. Pew Internet. http://pewinternet.org/Reports/2011/Smartphones/Section-1.aspx retrieved on July 2, 2013.

Transaction Fees. Dec 13, 2012. Google Play. https://support.google.com/googleplay/android-developer/answer/112622?hl=en&ref_topic=15867 retrieved on July 3, 2013.

Walter, John. 11/02/2011. Smartphones a big trend. Agriculture.com. http://www.agriculture.com/farm-management/technology/cell-phone-and-smart-phones/smartphones-a-big-trend_325-ar20351 retrieved June 14, 2013.

Wen, Howard. June 3, 2011. The ascendance of App Inventor. O'reilly Programming. http://programming.oreilly.com/2011/06/google-app-inventor-programmers-mobile-apps.html retrieved on June 12, 2013.

Wolber, David, Hal Abelson, Ellen Spertus, and Liz Looney. April 2011. App Inventor: Create Your Own Android Apps. Sebastopol, California. O'Reilly Media.

Woodill, G., and C. Udell. 2012. mAgriculture: The Application of Mobile Computing to the Business of Farming. *Float Mobile Learning*.

Woodill, G., and C. Udell. 2012. Future Uses of Mobile Technologies in Farming, Fishing and Forestry. *Float Mobile Learning*.

# Appendix A - Combine Efficiency Spreadsheet

| Dados Base Milho | |
|---|---|
| Custo Hora Máquina | R$ 252.00 |
| Custo Hora Plataforma Milho | R$ 116.00 |
| Custo TOTAL M+P | R$ 368.00 |
| Preço Diesel | R$ 1.89 |
| Preço Saca do milho (60 kg) | R$ 60.00 |
| Espaçamento (m) | 0.45 |
| Linhas | 15 |

| Velocidade (km/h) | Combustível (Litros/hora) | Perdas (kg/há) | Ha / Hora | Custo Máquina (R$/ha) | Perdas (R$/ha) | Diesel (R$/ha) | Resultado (R$/ha) |
|---|---|---|---|---|---|---|---|
| 3 | 38 | 20 | 2.025 | R$ 181.73 | R$ 20.00 | 35.466667 | R$ 237.20 |
| 4 | 40 | 30 | 2.700 | R$ 136.30 | R$ 30.00 | 28 | R$ 194.30 |
| 5 | 44 | 40 | 3.375 | R$ 109.04 | R$ 40.00 | 24.64 | R$ 173.68 |
| 6 | 47 | 50 | 4.050 | R$ 90.86 | R$ 50.00 | 21.933333 | R$ 162.80 |
| 7 | 50 | 60 | 4.725 | R$ 77.88 | R$ 60.00 | 20 | R$ 157.88 |
| 8 | 53 | 70 | 5.400 | R$ 68.15 | R$ 70.00 | 18.55 | R$ 156.70 |

| Dados Base Soja | |
|---|---|
| Custo Hora Máquina | R$ 252.00 |
| Custo Hora Plataforma Corte | R$ 90.00 |
| Custo TOTAL M+P | R$ 342.00 |
| Preço Diesel | R$ 1.89 |
| Preço Saca do Soja (60 kg) | R$ 80.00 |
| Largura de Corte(m) | 9.1 |

| Velocidade (km/h) | Combustível (Litros/hora) | Perdas (kg/há) | Ha / Hora | Custo Máquina (R$/ha) | Perdas (R$/ha) | Diesel (R$/ha) | Resultado (R$/ha) |
|---|---|---|---|---|---|---|---|
| 3 | 38 | 20 | 2.730 | R$ 125.27 | R$ 26.67 | 26.307692 | R$ 178.25 |
| 4 | 40 | 30 | 3.640 | R$ 93.96 | R$ 40.00 | 20.769231 | R$ 154.73 |
| 5 | 44 | 40 | 4.550 | R$ 75.16 | R$ 53.33 | 18.276923 | R$ 146.78 |
| 6 | 47 | 50 | 5.460 | R$ 62.64 | R$ 66.67 | 16.269231 | R$ 145.57 |
| 7 | 50 | 60 | 6.370 | R$ 53.69 | R$ 80.00 | 14.835165 | R$ 148.52 |
| 8 | 53 | 70 | 7.280 | R$ 46.98 | R$ 93.33 | 13.759615 | R$ 154.07 |

| Dados Base Feijão | |
|---|---|
| Custo Hora Máquina | R$ 252.00 |
| Custo Hora Plataforma Corte | R$ 90.00 |
| Custo TOTAL M+P | R$ 342.00 |
| Preço Diesel | R$ 1.89 |
| Preço Saca do Feijão (60 kg) | R$ 110.00 |
| Largura de Corte(m) | 9.1 |

| Velocidade (km/h) | Combustível (Litros/hora) | Perdas (kg/há) | Ha / Hora | Custo Máquina (R$/ha) | Perdas (R$/ha) | Diesel (R$/ha) | Resultado (R$/ha) |
|---|---|---|---|---|---|---|---|
| 3 | 38 | 80 | 2.730 | R$ 125.27 | R$ 146.67 | 26.307692 | R$ 298.25 |
| 4 | 40 | 90 | 3.640 | R$ 93.96 | R$ 165.00 | 20.769231 | R$ 279.73 |
| 5 | 44 | 100 | 4.550 | R$ 75.16 | R$ 183.33 | 18.276923 | R$ 276.78 |
| 6 | 47 | 115 | 5.460 | R$ 62.64 | R$ 210.83 | 16.269231 | R$ 289.74 |
| 7 | 50 | 130 | 6.370 | R$ 53.69 | R$ 238.33 | 14.835165 | R$ 306.86 |
| 8 | 53 | 145 | 7.280 | R$ 46.98 | R$ 265.83 | 13.759615 | R$ 326.57 |

# Appendix B - Template App Sources

http://ag-hes-server.appspot.com/aigui?fmt=combinetemplateapp.html