BTech Mid-Year Report
Dasarsh Vadugu

Abstract

This project requires the creation of a mobile device application which can serve as an intermediary between users working in a nursery and the information that they gather while doing so. The application will be useful because it will replace the current, paper-based system of collecting information and will therefore consequently save time and effort. Written for the Android platform, the application will merge the current, paper-based system of collecting information with a tablet interface. This will result in a change in the way that information is entered and will also introduce elements which can help to keep the information consistent and structured.

Project Outline

For the sake of quick and efficient recording of data in an outdoor nursery, an android application is to be made. This is to reduce the amount of time taken for the current, paper-based system to be processed and converted into an electronic form. The objective is to create an application which will act as an intermediary between the user who is recording information and the system where the recorded information is to be stored. Such a resultant application will cause a seamless transition from data being recorded to data being stored. Implementing such an application allows for the advantages of a mobile device to be considered. A mobile device's camera allows for the attaching of photos to documents, which can further be edited with freehand notes drawn onto photos as a result of the touchscreen interface. Wireless connectivity can allow for the synchronisation of files between device and server to ensure the consistency and simple transfer of media.

Scion is a New Zealand Crown Research Institute (CRI) that specialises in research, science and technology development for the forestry, wood product and wood-derived materials and other biomaterial sectors (About Scion, 2009). There are large grounds where plants and trees are grown and kept track of for their research purposes. In these grounds and in the nursery, several different things are observed and must be measured and recorded; these can vary from the root structures, rates of growth or colouration of plants and more.

One of the more common processes which require recording information from the nursery is mapping. Mapping is essentially a stocktake of the plants, resulting in a table which shows where plants are located, how many there are, which client has ordered them, and so on. The process of mapping can further be broken down into two categories; static mapping and dynamic mapping.

Static mapping refers to the mapping of plants which are directly planted in soil in the ground. These plants are not moved until the end of their research term or until they are ready to be sent to the clients which have ordered them.
Static mapping requires the following information to be recorded:

Field	Type
Compartment	Text
Bed	Numeric
Start of the bed	Numeric
End of the bed	Numeric
Bed length	Numeric
Number of plants	Numeric
Plant density	Numeric
Client	Text
Species	Text
Details 1 (if any)	Text
Details 2 (if any)	Text
Photo details (if any)	Text

Table 1, the fields and respective types of the data required for Static mapping form

Dynamic mapping refers to plants which are in small pots. These plants are moved from location to location within the nursery depending on their health and stages of growth. An example of this would be a sapling which has been growing inside a shed and receiving care during its early growth being moved outside once it is ready to receive direct sunlight or a plant which has taken too much harsh sunlight being moved back inside to receive careful watering to counteract its negative growth.
Dynamic mapping requires the following information to be recorded:

Field	Type
Environment	Text
Location	Text
Number of plants	Numeric
Plant density	Numeric
Client	Text
Species	Text
Details 1 (if any)	Text
Details 2 (if any)	Text
Photo details (if any)	Text

Table 2, the fields and respective types of the data required for Dynamic mapping form

As can be seen, both types of mapping require similar information to be recorded, with the exception of physical location which is fixed for static mapping and variable for dynamic mapping and are referred to as different (compartments as opposed to environments). Also the omission of the data relating to the bed in the dynamic form reflects the fact that the plants in question are not planted in the ground.

The information regarding the client and species featuring in both types of mapping and that of the location which is only featured in dynamic mapping are not to be entered freely by the user. Users will be required to choose from a list of clients, species and locations. This list will need to be accessible by the application, from which it will read and display the options to the user. The ability to edit the information regarding client, species and location should also be one of the functionalities of the application. This is to accommodate any additions of new clients, species or locations.

Related Work

Google Sheets is a mobile application which enables the creation, editing and collaboration on spreadsheets. Utilising Google Sheets, a user may create any form they wish and then populate it with data. There is also functionality to generate graphs and charts as a form of analysis on the data stored in the spreadsheet. Cell formatting and formulae are also supported.

The missing functionality is the form. There is no way to enter information based on a predefined form, which will then add the entered information into the appropriate spreadsheet.

The specification for the Scion application requires a form which will enable users to easily type data into fields, which will then be added to the appropriate spreadsheet.

Design

There are five sheets in total which will take the form of the underlying information behind the functionality of the application; two sheets for the static and dynamic mapping, and three sheets for the information regarding the clients, species and locations. Each of these sheets will need to be added to and this will be possible by the means of an individual form for each sheet. The form will contain all the different fields which are required to complete a row which can then be added to the corresponding sheet.

The forms for the client, species and location sheets will be simple in that they will not be doing more than appending rows onto the end of their corresponding sheets. The design for these three supplementary sheets will be near identical.

Field	Type
Company name	Text
Contact name	Text
Work phone	Phone number
Mobile phone	Phone number
Address	Text

Table 3, the fields and respective types of the data required for the Client form

Field	Type
Name	Text
Code	Text
Genus	Text
Variety	Text
Common name	Text

Table 4, the fields and respective types of the data required for the Species form

Field	Type
Location	Text
Description	Text

Table 5, the fields and respective types of the data required for the Location form

Added camera-based functionality will be added to the mapping forms. While completing any of the two mapping forms, users will be able to take a photo which can then be attached to the sheet. This attachment is reflected by the value for ‘photo details’ being populated with the filename of the photo that was taken. Additionally, if the user wishes to, they may draw on top of a photo which they have just taken in case they feel the need to draw attention to some part of the photo. This edited photo is attached to the sheet in the same way.

The sheets for clients, species and locations will be read by the application. This allows for the addition of dropdown boxes in the mapping forms so that users may choose one of the existing clients, species, and locations to populate the corresponding fields. This imposes a multiple choice scenario on the user where they are to choose the correct entry from a range of existing choices. As the application will be reading the aforementioned three sheets, any new rows added to the sheets using the application will be reflected in the dropdown boxes instantly.

Forgiveness is an aspect which needs to be implemented to cater for situations when the user has added an erroneous row to a sheet. In such a scenario, a user should be able to select the erroneous row from the sheet preview, which will load that row's values into the form, allowing for the user to edit any field which is incorrect. Once this has been done, the user pressing the button to confirm the row will replace the erroneous row with the corrected one, as opposed to adding onto the end. This aspect of forgiveness is also something which needs to be present in the photo editing process. Any drawn lines should be able to be undone as would be possible in any standard image editing application.

There must be clarity in the design of the forms such that users will be able to easily tell what information is to be filled in which field (Principles of User Interface Design). This serves as a form of assistance to the user in that they can be reminded what to observe and record by the application if they have forgotten or are new to the process.

Validation can be enforced by the application on the values entered into fields. This prevents erroneous entries in the context of format and ensures a kind of structure in the resulting sheet. Ensuring that a user only enters what they are meant to is an element which can easily be achieved on a tablet device. As the only means of entering data is by the soft keyboard, the forms can be configured in such a way that a standard QWERTY keyboard is only shown for field which require text entries and a numerical keyboard being shown for fields which require numerical entries.

Consistency between the designs of the forms is important for the user as this confirms for them that similar things act in the same way (Android Design Principles). For this reason, the layouts of the five forms have been kept the same in that fields which are the same or similar are in the same positions across all forms. This results in interfaces with elements that looks similar which can inherently be assumed of having similar behaviour, and makes it easier for users to understand what is required of them (Principles of User Interface Design).

The fields in the form must follow a logical order. This is for the ease of use for the user. A form with a logical flow will mean that users will not need to be jumping from field to field all across the screen. The piece of information which naturally comes first will be featured first and will be followed by that which naturally follows afterwards (Android Design Principles). The application will begin to require information from a large-picture perspective, and then begin to focus on the details of the information being collected.

Android Developer Tools and Software Development Kit

To develop this application, the Android Developer Tools (ADT) and the Android Software Development Kit (SDK) will be used. This allows for the application to be built in Java using Eclipse Integrated Development Environment (IDE). The IDE can emulate Android devices of varying screen size and specifications, allowing for testing to be done on these virtual devices while still retaining functioning features such as the camera, sensors, multitouch and telephony (Developer Tools).

The ADT also allows for a running application to be installed onto a physical Android device, which was what was done for development. The application was run on a tablet which directly reflects its specification to be used on a tablet device. Debugging the application on a real, physical tablet was also advantageous due to the low speed issues and lack of responsiveness of the emulator.

The Android SDK version that was being developed for was 8 so every device running an Android version from 2.2 onwards can run the application.

Data Model

The Entity-Relationship Diagram (ERD) represents the data and the flow of information within the application. As mentioned before, the two mapping forms require information from two (or three in the case of the dynamic mapping) sheets to complete a row which can then be added to the sheet. This is evident in the ERD where the relationship between the two main forms and the three remaining forms is represented as a one-to-many relationship. This means that for any given static mapping entry, there can only be one client, or one species assigned. The same can be said for the dynamic mapping entries with the addition of the one-to-many relationship with locations.

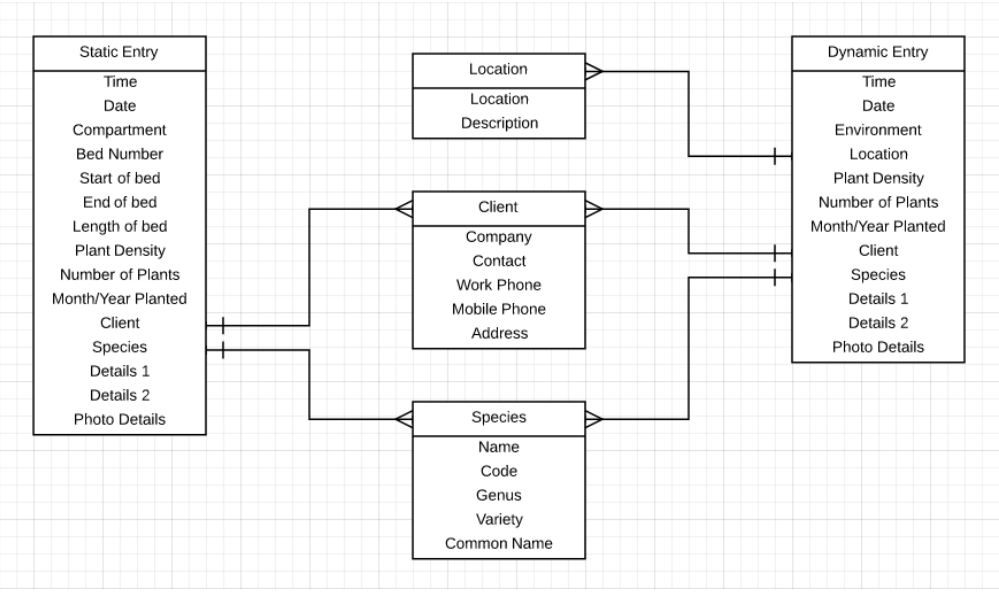

Figure 1, ERD showing the relationship between the data in the different forms

Implementation

Android Manifest

The Android Manifest xml file is mandatory for every application. It is an accumulation of all the information regarding the application which is to be sent to the Android system before the actual running of the application's code (App Manifest). The information that it handles includes the package name, the minimum SDK requirements, all the activities which will be used within the application and the permissions which outline which protected parts of the Android API are to be used, such as activating the camera or writing to the memory card.

For the application at this point, the manifest contains the following:

A package name of com.dasarsh.scion.
A minimum SDK of 8 which correlates to an Android version of 2.2.
A target SDK of 18 which correlates to an Android version of 4.3.
The permissions "android.permission.READ_EXTERNAL_STORAGE" and "android.permission.WRITE_EXTERNAL_STORAGE" for the sake of reading and writing from the memory card.
An activity for each of the following:
- EntryPoint
- StaticForm
- DynamicForm
- ClientForm
- SpeciesForm
- LocationForm
- EditPhotoSurface

Static and Dynamic Mapping Forms

Both types of mapping require one activity each and both activities require two files for each of them; a java file and an xml file. The xml file contains all the information regarding the layout of the activity such as the layouts and their elements on the screen. The java file contains all the code which enables what is seen on the screen to have their respective functionalities and also handles any errors.

The XML Layouts

The layouts of the forms are near identical. They all have half of the screen dedicated to the sheet preview while the other half is dedicated to the actual form. At the bottom of the half which contains the form, there is a panel of buttons. The whole view is contained in a LinearLayout, which was divided in two for the HorizontalScrollView to contain the preview and another LinearLayout to contain the form and the buttons.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```
    android:orientation="vertical"
    android:weightSum="100" >

    <HorizontalScrollView
        android:id="@+id/STATIChsvPreview"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="50" >
    </HorizontalScrollView>

    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:weightSum="50" >
    </LinearLayout>

</LinearLayout>
```

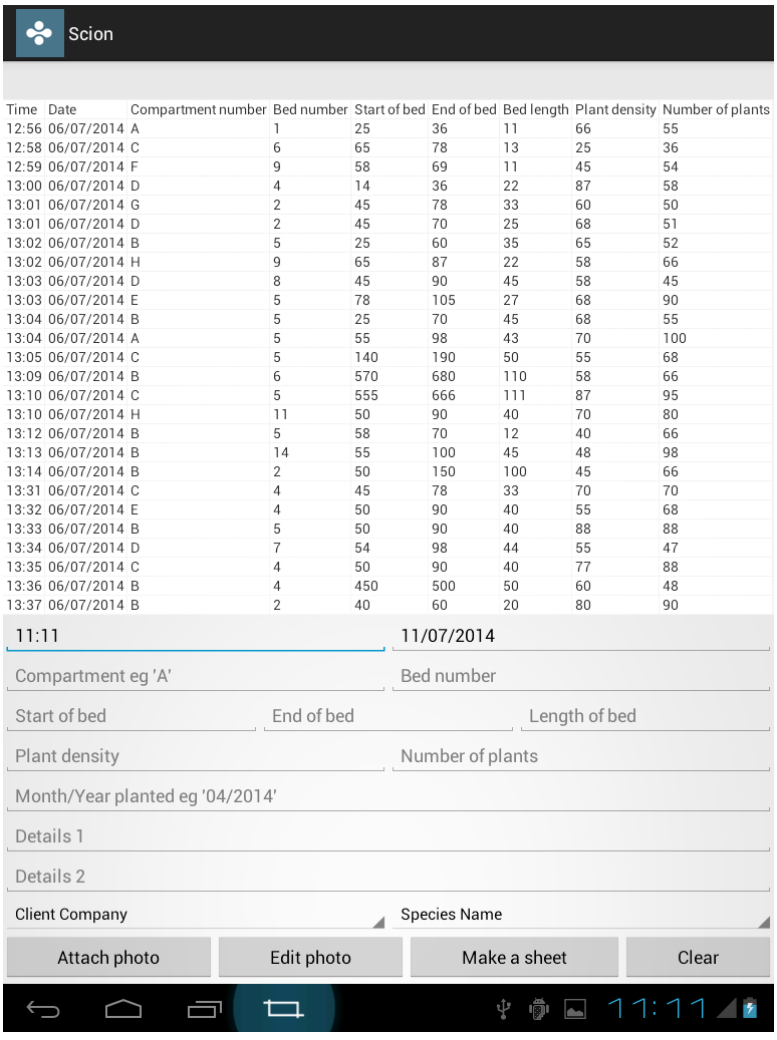

Figure 2, the division of the screen in two for the preview and the form

The sheet preview's design dictates that it should look and act like a spreadsheet editor similar to Microsoft Excel. This means that should the sheet become large in terms of both rows and columns, there should be no issues with both horizontal and vertical scrolling in order to be able to view any cell at any position. To accomplish this requirement, nesting layouts within layouts was necessary.

The HorizontalScrollView further contains a ScrollView which contains a TableLayout. The TableLayout within a ScrollView gives the preview the functionality of vertical scrolling as the TableLayout grows vertically. The ScrollView within a HorizontalScrollView gives the functionality of horizontal scrolling as the TableLayout (and the ScrollView encasing it) grows horizontally.

```
<HorizontalScrollView
        android:id="@+id/STATIChsvPreview"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="50" >

        <ScrollView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="horizontal" >

            <TableLayout
                android:id="@+id/STATICtableLayoutPreview"
                android:layout_width="fill_parent"
                android:layout_height="fill_parent" >


                    …

            </TableLayout>
        </ScrollView>
</HorizontalScrollView>
```

The lower half of the screen is dedicated to the LinearLayout of the form with which users are to record information to add rows to the sheet. This was implemented in a straightforward way such that every field which requires the whole width of the screen is simply added to the LinearLayout, whereas fields which are related and are to be grouped together on the same line are added into a nested LinearLayout. An example of this is the two fields regarding the beginning and ending of a bed, and the field regarding the length of the bed. All three fields are added to a LinearLayout which is nested within the LinearLayout dedicated for the bottom half of the screen.

```
        <LinearLayout
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:orientation="vertical"
          android:weightSum="50" >
```

```
...

        <LinearLayout
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:orientation="horizontal"
              android:weightSum="99" >

              <EditText
                  android:id="@+id/STATICetBedStart"
                  android:layout_width="fill_parent"
                  android:layout_height="wrap_content"
                  android:layout_weight="33"
                  android:ems="10"
                  android:hint="@string/StartOfBed"
                  android:inputType="number" />

              <EditText
                  android:id="@+id/STATICetBedEnd"
                  android:layout_width="fill_parent"
                  android:layout_height="wrap_content"
                  android:layout_weight="33"
                  android:ems="10"
                  android:hint="@string/EndOfBed"
                  android:inputType="number" />

              <EditText
                  android:id="@+id/STATICetBedLength"
                  android:layout_width="fill_parent"
                  android:layout_height="wrap_content"
                  android:layout_weight="33"
                  android:ems="10"
                  android:hint="@string/LengthOfBed"
                  android:inputType="number" />
                </LinearLayout>

        ...

</LinearLayout>
```

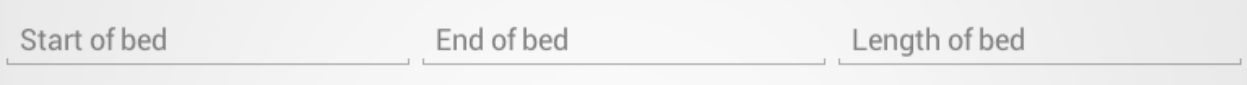

Figure 3, the grouping of the bed-related fields on one line

The form layout features fields in which users are to enter the data which they record. These fields are EditText elements and text can be entered into them. EditText elements can give users hints to serve as reminders on what to enter. These hints are useful to provide clarity and also functions to remind the users of what is to be entered. An EditText element can be configured to only accept a type of input, which serves as validation in the following case where

only a number will be accepted, which prompts the user with a numberpad as opposed to a QWERTY keyboard to type with.

```
<EditText
      android:id="@+id/STATICetBed"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="50"
      android:ems="10"
      android:hint="@string/BedNumber"
      android:inputType="number" />
```

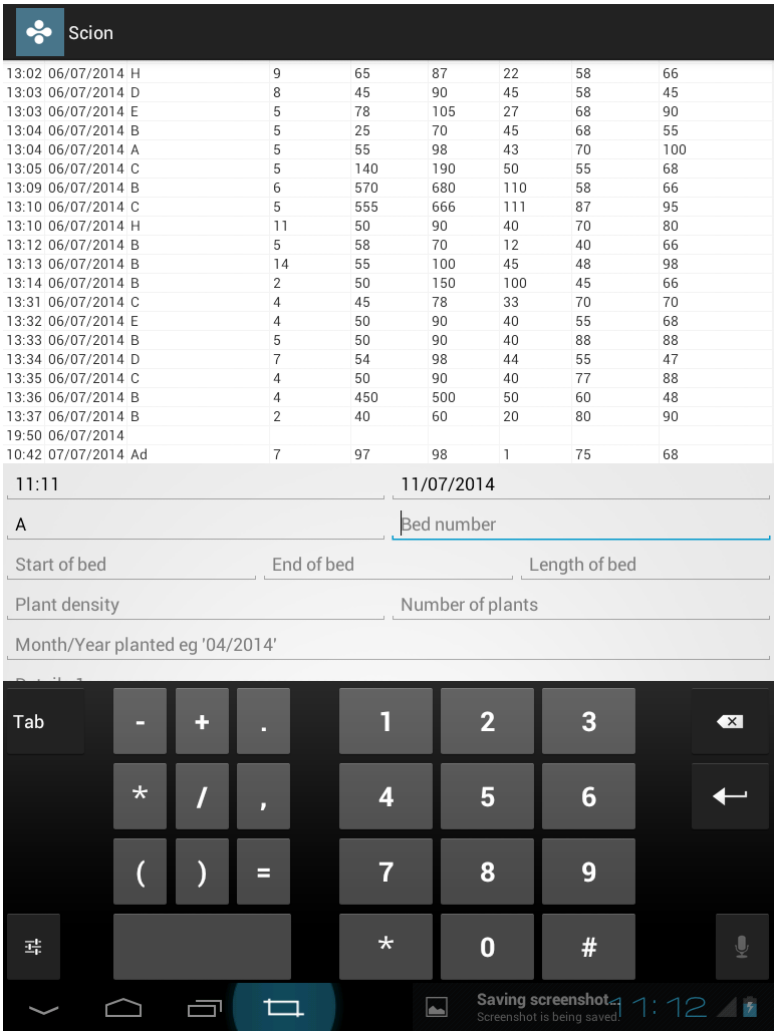

Figure 4, a numberpad is shown for input which should strictly be numeric

Spinner elements feature in the form layout. These elements are for giving the user a multiple choice scenario where they are to choose a single option which is most appropriate. For this application, the Spinner elements are used to provide a dropdown box where users are to select the appropriate Client, Species and Location.

```
<Spinner
      android:id="@+id/STATICspCompany"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_weight="50" />
```

Button elements conclude the form layout. These elements are used to trigger events when they are pressed and are listened to within the form's accompanying java class. The two main mapping forms contain four buttons which are nested within a LinearLayout which groups them together.

```
<LinearLayout
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:orientation="horizontal"
      android:weightSum="100" >

      <Button
          android:id="@+id/STATICbAttachPhoto"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:layout_weight="25"
          android:text="@string/AttachPhoto" />

      <Button
          android:id="@+id/STATICbEditPhoto"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:layout_weight="25"
          android:text="@string/EditPhoto" />

      <Button
          android:id="@+id/STATICbSubmit"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:layout_weight="25"
          android:text="@string/MakeASheet" />

      <Button
          android:id="@+id/STATICbClear"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:layout_weight="25"
          android:text="@string/Clear" />
 </LinearLayout>
```

Figure 5, the layout of buttons featured in both Static and Dynamic mapping forms

Figure 6, the layout of buttons featured in the Client, Species and Location forms

The three forms for Client, Species and Location which do not require camera and photo editing functionalities have only the buttons required to add rows and clear fields. They are near identical.

Figure 7, the three forms for Client, Species and Location are near identical aesthetically and functionally

Imported APIs

The Java Excel API of version 2.6.12 was imported for this application. Its purpose is to read, write and modify Excel spreadsheets (Khan, Java Excel API - A Java API to read, write and modify Excel spreadsheets). This API allows for Java code to access the means to create spreadsheets dynamically as well as read spreadsheets and access the cell data.

The Java Classes

Java classes which accompany their respective xml classes instantiate the xml file's elements and handle events which are either triggered by those elements or require updating the values of those elements.

All the java classes created extends the Activity class which allows the java class to be launched as an activity from within other java classes. Having the Activity class as a super class allows for the java class to have access to many of the methods within said Activity class, such as the onCreate method which allows for the java class to be linked to its respective xml layout.

All of the java classes implement the OnClickListener to be able to better handle events where buttons have been pressed. The two mapping forms implement the OnItemSelectedListener for the case of when an item in a Spinner element has been selected. Due to the fact that these two listeners are interfaces, the methods within them must be implemented within the java class, even if empty.

What must be done first in the java class is the linking of the java class with its respective xml layout class. This is done by setting the content of the screen to be the xml view.

```
setContentView(R.layout.static_form);
```

Only after this can the instantiation of the xml elements be done. Certain EditText elements have been programmatically configured in the java class such that they are not able to be edited. These are the time and date EditText elements which are to be automatically generated by the Android API. This was done by passing null to an EditText element's keyListener method.

```
EditText date, time;
time = (EditText) findViewById(R.id.STATICetTime);
time.setKeyListener(null);
```

The purpose of using this method to nullify any changes made to the EditText as opposed to using a simple label which cannot be changed by default is to retain the look and feel of a form which is going to be used to populate a row which will be added to a sheet.

The time and date values which are displayed in the time and date EditText elements are derived from the Date class. The java class instantiates a Date which matches the system time of the device and deduces the hours and minutes to use for the time and the day, month and

year to use for the date. String operations are done to prepend ‘0’ to the minutes and seconds values to prevent the time being shown as 12:9 instead of 12:09. The date and time are also lastly combined together to create a String which will be used as the filenames of photos taken using the application.

The two EditText elements for the starting and ending lengths of the beds, which are then used to calculate the length of the bed itself, are set to update the value of the EditText element dedicated to the length of the bed once the focus has changed from either the start or end EditText. This was done using the EditText element’s setOnFocusChangeListener method and passing a custom OnFocusChangeListener into it.

```
bedStart.setOnFocusChangeListener(new OnFocusChangeListener() {
		public void onFocusChange(View v, boolean hasFocus) {
			if (!hasFocus) {
				updateBedLength();
			}
		}
	});
```

Spinner elements are instantiated by means of population with the data which is being read from the corresponding xls sheet. The Client Spinner element will be reading the column of clients from the client.xls sheet and then displaying those clients as Spinner items.

```
ArrayList<String> list = new ArrayList<String>();
…
// add the names of the companies stored in the client.xsl file into
// the ArrayList for the spinner
for (int i = 0; i < numRows; i++) {
	list.add(sheet.getCell(0, i).getContents());
}
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
			android.R.layout.simple_spinner_item, list);

adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
;
spinner.setAdapter(adapter);
spinner.setOnItemSelectedListener(this);
```

The Spinner is populated with items which correlate to columns of the sheet which is being read, inclusive of the column heading such as “Client Company” or “Species Name”. The headings are included due to the fact that there is no direct way to programmatically configure a hint for the user in the same way that can be done with an EditText element.

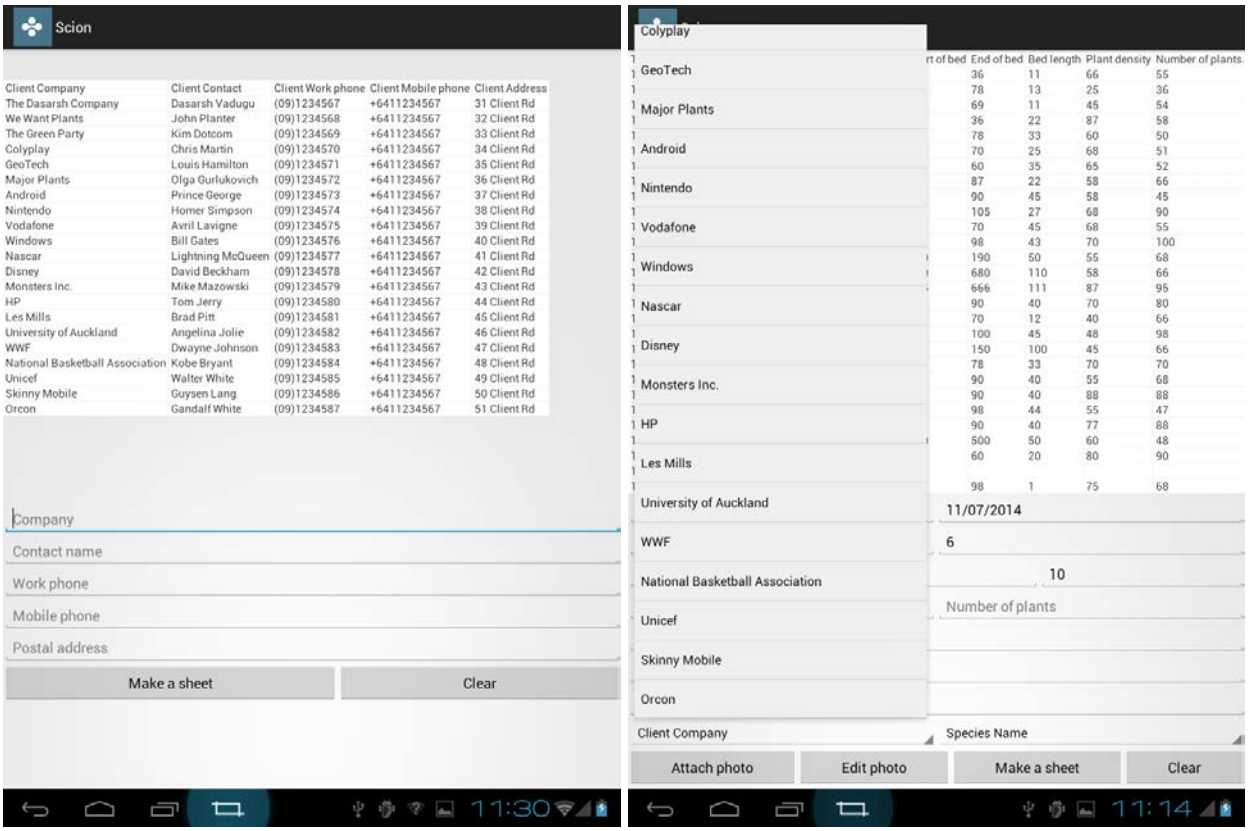

Figure 8, the entries from the Client Company column from the Client sheet are shown as items in the Spinner element in the Static form

The onItemSelected method of the OnItemSelectedListener interface which was mandatorily implemented includes a switch case which deduces which Spinner element has had an item selected from, and then proceeds to check the position of the item which was selected. If the item is in position 0, then that means the first element has been selected, which is the column heading. The column heading is not valid data which can be entered into the sheet, and so if it has been selected, the data added in its stead is simply an empty String.

```
@Override
public void onItemSelected(AdapterView<?> parent, View view, int position,
            long id) {

      // for handling when a Spinner item is selected

      switch (parent.getId()) {
      case R.id.STATICspCompany:
            // if the column header is not the one which is selected
            if (position != 0) {
                  company.setSelection(position);
                  companyData = (String) company.getSelectedItem();
            } else {
                  companyData = "";
            }
```

```
                break;
        case R.id.STATICspSpecies:
                // if the column header is not the one which is selected
                if (position != 0) {
                        species.setSelection(position);
                        speciesData = (String) species.getSelectedItem();
                } else {
                        speciesData = "";
                }

                break;

        // cases are set such that if there is not change to the spinner
        // values and they display the column headers, they do not appear on
        // the sheet once added

        }

}
```

In the xml layout, the sheet preview consists of a HorizontalScrollView encasing a ScrollView encasing a TableLayout. The TableLayout is the real sheet preview being displayed, and the HorizontalScrollView and ScrollView which it is nested in serves as a means of navigation within the sheet. A TableLayout is to be populated with TableRow elements, which can contain any element within it. The sheet that is to be previewed is being read from one row at a time, and each cell in that row is being used to create a TextView element which is then added to a TableRow element and finally added to the TableLayout element. The TextView is being made with some padding on both left and right sides of the cell information and a simple resource defining a border is being drawn do differentiate the cells from one another and separate the data.

```
HorizontalScrollView hsc;
TableLayout tl;
…
hsc = (HorizontalScrollView) findViewById(R.id.STATIChsvPreview);
tl = (TableLayout) findViewById(R.id.STATICtableLayoutPreview);
…
// starting at rowsDisplayed to add only new rows
for (int i = rowsDisplayed; i < numRows; i++) {
        // create a new TableRow
        TableRow tr = new TableRow(this);
        // width FILL_PARENT height WRAP_CONTENT
        tr.setLayoutParams(new TableRow.LayoutParams(
                        TableRow.LayoutParams.FILL_PARENT,
                        TableRow.LayoutParams.WRAP_CONTENT));
        for (int j = 0; j < numCols; j++) {
                // create a new TextView
                TextView b = new TextView(this);
                // because jxl works like (columns, rows)
                b.setText(" " + sheet.getCell(j, i).getContents() + " ");
                // width FILL_PARENT height WRAP_CONTENT
```

```
                b.setLayoutParams(new TableRow.LayoutParams(
                            TableRow.LayoutParams.FILL_PARENT,
                            TableRow.LayoutParams.WRAP_CONTENT));
                // draw the border around each TextView for grid look
                b.setBackgroundResource(R.drawable.border);
                // add the TextView to the TableRow
                tr.addView(b);

            }
            // Add the TableRow to the TableLayout which is width
            // FILL_PARENT and height WRAP_CONTENT
            tl.addView(tr, new TableLayout.LayoutParams(
                        TableLayout.LayoutParams.FILL_PARENT,
                        TableLayout.LayoutParams.WRAP_CONTENT));
        }
```

For the static form, which features the starting and ending bed EditText elements and the bed length EditText element, there is a method which calculates the length from the starting and ending values. This method is called whenever there is a shift in focus from either of the two EditText elements. The first check being made is one which ensures that the lengths of the data in the two elements are greater than 0, which means that there is a non-null value in the EditText element. The values are then parsed from their default type of String to integer. This is a safe operation and there is no case where data which is not a number is attempted to be parsed into an integer format. This is due to the validation occurring on the xml layout side where the input type is assigned to being numeric, which only allows users to enter numbers. On successfully parsing the values, there is a check to see whether the end value is greater than the start value. If this is true, the length is calculated by subtraction. If that is not the case, the user will be alerted to it by means of an AlertDialog which then prompts the user back to the EditText element for the end of the bed by requesting the focus to that element.

```
        @SuppressWarnings("deprecation")
        public void updateBedLength() {
            // if both bedStart and bedEnd have values
            if (bedStart.getText().length() > 0 && bedEnd.getText().length() > 0) {
                // get these values
                int start = Integer.parseInt(bedStart.getText().toString());
                int end = Integer.parseInt(bedEnd.getText().toString());
                // compute the length
                int length = end - start;
                // if end is greater than start (as it should always be)
                if (end > start) {
                    // set the text to bedLength
                    bedLength.setText(length + "");
                } else {
                    // otherwise alert the user
                    AlertDialog alertDialog = new AlertDialog.Builder(context)
                                .create();
                    alertDialog.setTitle("Invalid argument");
                    alertDialog.setMessage("The value for 'bed end' must be
greater than the value for 'bed start'");
                    alertDialog.setButton("Okay",
```

```
                                        new DialogInterface.OnClickListener() {
                                            public void onClick(DialogInterface
dialog, int which) {
                                                bedEnd.requestFocus();
                                            }
                                        });
                                alertDialog.show();
                            }
                        }
                }
```

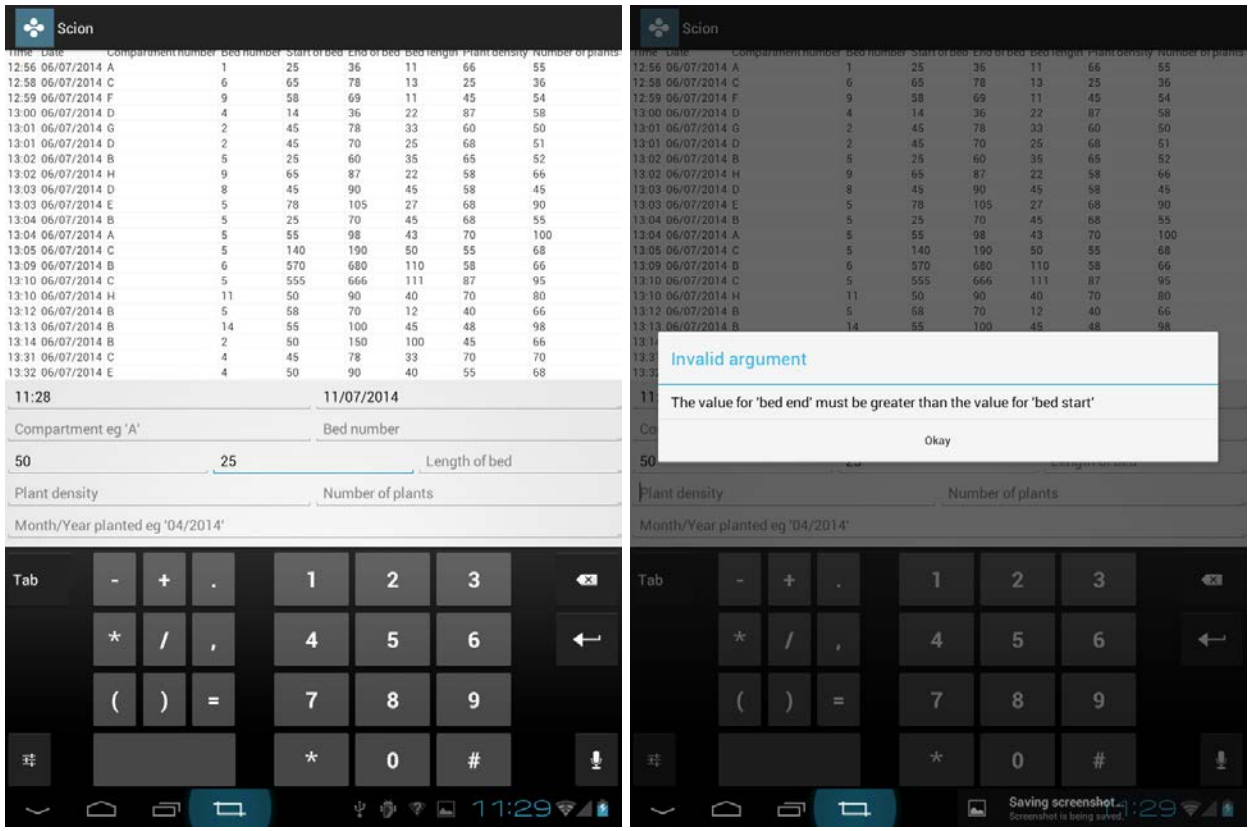

Figure 9, the value for the end of the bed (25) is less than that of the start of the bed (50), and so the user is prompted with an alert

The onClick method which must be implemented due to the class implementing the OnClickListener interface contains the code which handles what is to happen if any of the buttons are pressed. The method is passed a view, from which a switch case is required to determine which of the buttons has been pressed.

```
        @SuppressWarnings("deprecation")
        @Override
        public void onClick(View v) {

                // what to do when buttons are pressed

                switch (v.getId()) {
```

```
        // submit button is pressed
        case R.id.STATICbSubmit:
              …
              break;
        // attach photo button is pressed
        case R.id.STATICbAttachPhoto:
              …
              break;
        // edit photo button is pressed
        case R.id.STATICbEditPhoto:
              …
              break;
        // clear button is pressed
        case R.id.STATICbClear:
              …
              break;
        }

}
```

In the case of the “Submit” button being pressed, all the information from the respective fields will be collected, and then there will be a check to determine whether the sheet for the mapping exists.

```
collect();
File path = new File("sdcard/scion/static_mapping.xls");
```

If the path exists, the collected data will be appended to the existing file as a new row. If the path does not exist, that means there is no previously existing spreadsheet to append onto and a new sheet is required to be made. A Toast message is displayed afterwards to confirm to the user that the required operation has completed.

```
if (path.exists()) {
        // if so, add to the sheet
        excelAdder();
        Toast toast = Toast.makeText(this, "Row added to sheet",
                      Toast.LENGTH_SHORT);
        toast.show();
} else {
        // if not, make a new sheet and add to it
        excelTesterStatic();
        Toast toast = Toast.makeText(this, "Sheet created",
                      Toast.LENGTH_SHORT);
        toast.show();
}
```

Once the new row has been added, the preview is required to show the changes made to either the existing sheet, or display the new sheet. This is done by essentially reloading the whole activity again and finishing the running activity so as not to waste resources.

```
// restart the activity so that the form can be used again to add
// more rows
startActivity(starterIntent);
// finish this activity so as not to waste resources more
finish();
```

The variable starterIntent is required to restart the same activity. It is an Intent type and is a global variable which was instantiated in the onCreate method as such:

```
starterIntent = getIntent();
```

In the case of the “Attach photo” button being pressed, an intent for starting the camera application will be started. The user will be prompted to take a photo using the device’s default camera application and once they have done so, the camera activity will complete and the code from the onActivityResult method will be run. This is due to the fact that the startActivityForResult method is used, which means that there is an opportunity to do something once an activity finishes, as opposed to letting them finish and moving on which is what would happen if the startActivity method was used instead.

```
// create an intent for starting the camera
i = new Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTURE);
// activity for result returns to the onActivityResult() method
startActivityForResult(i, cameraData);
```

On completion of the camera activity, there is a switch case which determines that the camera activity has finished and has results which are to be handled. These results are the data regarding the photo taken. With this data, the application saves the photo in the jpg format, using the date and time string mentioned earlier as the filename. Lastly, a boolean used to store whether a photo has been taken or not is set to true.

```
switch (requestCode) {
case cameraData:
	Bundle extras = data.getExtras();
	Bitmap bmp = (Bitmap) extras.get("data");
	OutputStream stream;
	try {

		File scionDir = new File("/sdcard/scion/photos/");
		// If /sdcard/scion/photos/ is not a directory, then make it
		if (!scionDir.isDirectory()) {
			scionDir.mkdirs();
		}
		// save the photo using the timestamp (dateTimeString) as
		// its filename
		stream = new FileOutputStream("/sdcard/scion/photos/"
				+ dateTimeString + ".jpg");
		bmp.compress(CompressFormat.JPEG, 100, stream);
		photoData = dateTimeString + ".jpg";
	} catch (FileNotFoundException e) {
		e.printStackTrace();
```

```
            }

            Toast toast = Toast.makeText(this, "Photo saved and attached",
                        Toast.LENGTH_SHORT);
            toast.show();

            // set isPhotoAttached boolean to true so editing can take place
            isPhotoAttached = true;
            break;
      case editPhotoData:
            ...
            break;
      }
```

In the case of the “Edit photo” button being pressed, the boolean for deducing whether a photo has been taken will be checked. If the boolean is true and a photo has been taken, then an activity in which the taken photo can be edited is started. A Bundle object is used to send information from between activities. In this case, the Bundle object is carrying the path of the previously taken photo as this will be used as the path for the edited photo once editing has been completed. If a photo has not been taken, then the user is prompted to do so by means of an AlertDialog before trying to edit a photo.

```
      // if a photo has been attached
      if (isPhotoAttached == true) {
            // create a new bundle for sending to the next activity
            Bundle basket = new Bundle();
            // attach the path to the attached photo
            basket.putString("photoPath", "/sdcard/scion/photos/"
                        + dateTimeString + ".jpg");
            // create an intent for the editing activity
            Intent a = new Intent(this, EditPhotoSurface.class);
            // send the bundle
            a.putExtras(basket);
            // start the editing activity
            startActivityForResult(a, editPhotoData);
      } else {
            // if a photo has not been attached, alert and prompt the user
            // to do so with a dialog
            AlertDialog alertDialog = new AlertDialog.Builder(context)
                        .create();
            alertDialog.setTitle("No photo to edit");
            alertDialog
                        .setMessage("Take a photo first in order to edit it.");
            alertDialog.setButton("Okay",
                        new DialogInterface.OnClickListener() {
                              public void onClick(DialogInterface dialog,
                                          int which) {
                                    // nothing happens
                              }
                        });
            alertDialog.show();
      }
```

In the case that the “Clear” button is pressed, the initialise and clearEditTexts methods are called. Calling the initialise method ensures that the Spinner elements are reset to their default values and calling the clearEditTexts method programmatically sets the values for all the EditText elements to be that of an empty String.

```
// initialise() resets the spinners
initialise();
// clearEditTexts() resets the EditTexts
clearEditTexts();

private void clearEditTexts() {

	// clears the values for the EditTexts

	…

	plantDensity.setText("");
	whenPlanted.setText("");
	numPlants.setText("");

	…

}
```

There are two methods in which spreadsheets are written to; the method used to create a spreadsheet if there is not one already existing, and the method used to add a row to an existing spreadsheet. Both methods make use of the Java Excel API to do so. Writing to a spreadsheet requires creating a WritableWorkbook object using the Workbook’s factory method (Khan, Java Excel API Tutorial). Then the WritableWorkbook object is used to create a WritableSheet object which Label objects are added to. Label objects dictate the cell position of the information that is to be entered into the sheet. Once all the changes have been made, the WritableWorkbook object must be written to and then closed in that order, otherwise an empty file will be created. The resultant file is an xls file which can be read by Excel.

In both methods, the data collected from the fields are added to an ArrayList. This ArrayList is cycled through and each item is added to the sheet. The difference between the two methods is that the excelAdder method creates a copy of the existing sheet, counts the number of rows, appends the new row, and then overwrites the existing sheet. The excelCreate method adds column headers first, then proceeds to add the data and then saves the sheet.

The java classes for the three supplementary forms (Client, Species and Location) are near identical to one another. They do not contain any of the functionality with regards to the camera, or editing photos. They are simply for adding rows to their respective sheets and contain many of the same methods, with the exceptions being those which are necessary to have as a consequence of implementing the OnItemSelectedListener interface.

EditPhotoSurface.java

The EditPhotoSurface.java class is used to enable photo editing functionality. Its purpose is to receive photos which have been taken by the user, and then give them the capability of drawing on top of that photo to bring attention to any area which requires it.

The EditPhotoSurface class extends the Activity class, and contains a nested class; the DrawingView which extends the View class. The DrawingView class is what allows the user to draw on top of the photo.

An instance of a DrawingView object is created, and then the Bundle object which is passed to the EditPhotoSurface activity is opened. This Bundle object contains the path to the image that was taken. With this, the image can be loaded as a Bitmap object from local memory and drawn as the background of the DrawingView. Once this has been done, the DrawingView is passed as an argument to the setContentView method.

```
DrawingView dv;
String photoPath;
…
dv = new DrawingView(this);

Bundle gotBasket = getIntent().getExtras();
photoPath = gotBasket.getString("photoPath");
…
Bitmap source = BitmapFactory.decodeFile(photoPath, options);
Drawable bg = new BitmapDrawable(source);
dv.setBackgroundDrawable(bg);
setContentView(dv);
```

The nested DrawingView class handles all the drawing events. By means of a switch case on a MotionEvent object, the onTouchEvent method determines whether the user has touched the screen, is moving their finger while touching the screen or lifted their finger from the screen.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
	float x = event.getX();
	float y = event.getY();

	switch (event.getAction()) {
	case MotionEvent.ACTION_DOWN:
		touch_start(x, y);
		invalidate();
		break;
	case MotionEvent.ACTION_MOVE:
		touch_move(x, y);
		invalidate();
		break;
	case MotionEvent.ACTION_UP:
		touch_up();
		invalidate();
```

```
                break;
        }
        return true;
}
```

Once a user has touched the screen, the application will give initial values to an already defined Path object. As the user moves their finger across the screen, the Path object is given more coordinates to map to, giving the effect of a line being drawn. Finally once the user lifts their finger from the screen, the line drawn is set and the Path object is reset and ready to be used again for any further lines.

```
private Path mPath;
mPath = new Path();

private float mX, mY;
private static final float TOUCH_TOLERANCE = 4;

private void touch_start(float x, float y) {
        mPath.reset();
        mPath.moveTo(x, y);
        mX = x;
        mY = y;
}

private void touch_move(float x, float y) {
        float dx = Math.abs(x - mX);
        float dy = Math.abs(y - mY);
        if (dx >= TOUCH_TOLERANCE || dy >= TOUCH_TOLERANCE) {
                mPath.quadTo(mX, mY, (x + mX) / 2, (y + mY) / 2);
                mX = x;
                mY = y;
        }
}

private void touch_up() {
        mPath.lineTo(mX, mY);
        mCanvas.drawPath(mPath, mPaint);
        mPath.reset();
}
```

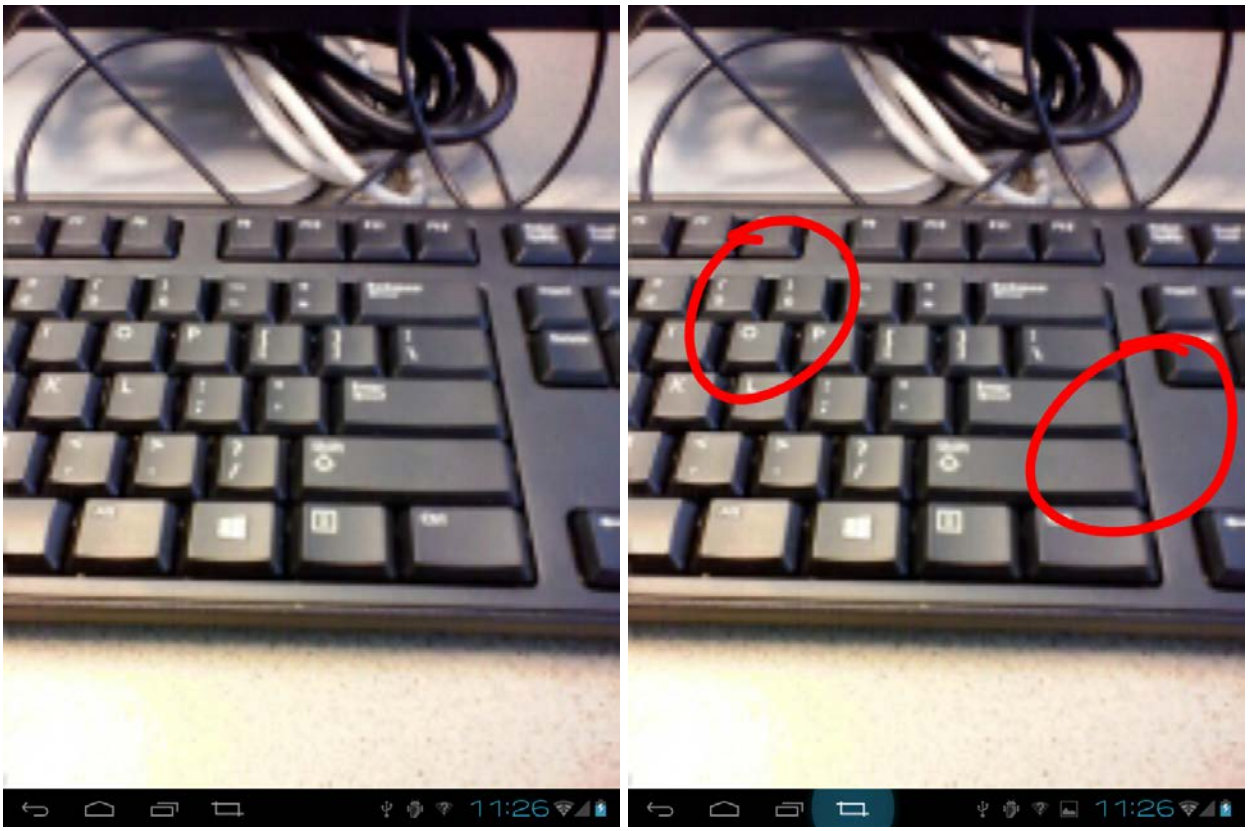

Figure 10, an attached photo can be drawn on to bring attention to any particular area

Saving of the image has been handled by making use of the device's default Android back button. If this is pressed, the user will be asked whether they want to save the photo with the changes they have made, or whether to discard them. Both of these choices call the finish method, causing the activity to end and return to the onActivityResult method from the respective mapping activity which called it. Control is returned to the onActivityResult method because the EditPhotoSurface activity was started using the startActivityForResult method.

```
	@SuppressWarnings("deprecation")
	@Override
	public boolean onKeyDown(int keyCode, KeyEvent event) {
		if (keyCode == KeyEvent.KEYCODE_BACK && event.getRepeatCount() == 0) {

				AlertDialog alertDialog = new
AlertDialog.Builder(context).create();
				alertDialog.setTitle("Save image?");
				alertDialog.setMessage("Do you want to save this image?");
				alertDialog.setButton("Save",
					new DialogInterface.OnClickListener() {
					public void onClick(DialogInterface dialog, int which) {
						dv.setDrawingCacheEnabled(true);
						Bitmap b = dv.getDrawingCache();
						try {
							b.compress(CompressFormat.JPEG, 95,
							new FileOutputStream(photoPath));
```

```
                        finish();
                    } catch (FileNotFoundException e) {
                        e.printStackTrace();
                    }
                }
            });
            alertDialog.setButton2("Don't save",
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int which) {
                        finish();
                    }
            });
            alertDialog.show();

            return true;
        }

        return super.onKeyDown(keyCode, event);
}
```

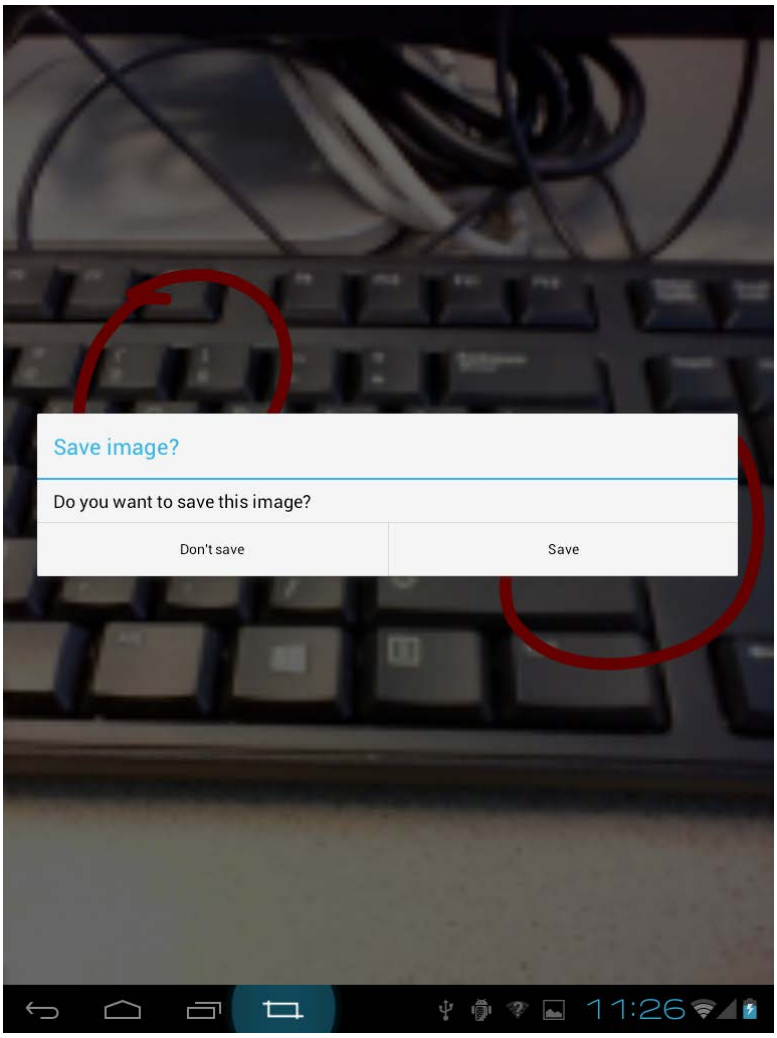

Figure 11, the user is prompted to save or discard the edits that have been made

Conclusion

There are a few minor modifications which need to be made to the application. These are regarding the automatic calculation of the bed length, the date formatting and the image quality of the photo editing functionality. As these changes are minor, they will not require much time to make, leaving more time for new additions or removals if it is seen fit.

Considering that the project was delayed due to issues with travel and organising the best time to do so, there has been significant progress made in a short amount of time. This was partially due to the fact that the break between semesters provided a time where work was solely focussed on the project and nothing else like university papers.

Upcoming Stages

There are still some features which have not been implemented as of yet. One of these is the forgiveness aspect of the design which requires that users be able to effectively undo any mistakes that they make. Implementing this will mean adding listeners to the preview in such a way that if a user was to press and of the cells, the data stored in the row which contains that cell will populate the form, allowing for the user to change any of the information. Another element which requires the addition of this forgiveness aspect is the photo editing functionality, as a user may wish to undo whatever they have drawn and try again.

The photo editing functionality needs modification in that the method used to load the photo from the memory and display it on the screen causes the photo to be displayed in low quality. This is to do with the fact that the photo is being drawn directly onto a Canvas object, and is not being used as an element in a view. Subsequent to some research, there seems to be a way which allows for the photo to be set as the background of an ImageView element which should produce a photo of better quality.

As of now, the application saves and loads spreadsheets from local memory; the memory card. The next feature to implement regards the transferring of the sheets from the client-side device to a server over a wireless network. This may require the use of the Google Drive API. The API is able to handle offline access and syncing files, allowing for reading from and writing to files as you would using a local file system (Introduction to the Google Drive Android API). Using this API, a user will be able to always have access to the latest revision of any of the five sheets which are used in this application.

Bibliography

About Scion. (2009). Retrieved August 8, 2014, from Scion: http://www.scionresearch.com/general/about-us

Android Design Principles. (n.d.). Retrieved August 8, 2014, from Android Developers: http://developer.android.com/design/get-started/principles.html

App Manifest. (n.d.). Retrieved August 9, 2014, from Android Developers: http://developer.android.com/guide/topics/manifest/manifest-intro.html

Developer Tools. (n.d.). Retrieved August 8, 2014, from Android Developers: http://developer.android.com/tools/index.html

Introduction to the Google Drive Android API. (n.d.). Retrieved August 7, 2014, from Android Developers: https://developers.google.com/drive/android/intro

Khan, A. (n.d.). *Java Excel API - A Java API to read, write and modify Excel spreadsheets*. Retrieved August 7, 2014, from Java Excel API: http://www.andykhan.com/jexcelapi/

Khan, A. (n.d.). *Java Excel API Tutorial*. Retrieved August 7, 2014, from Java Excel API Tutorial: http://www.andykhan.com/jexcelapi/tutorial.html#writing

Principles of User Interface Design. (n.d.). Retrieved August 8, 2014, from Bokardo: http://bokardo.com/principles-of-user-interface-design/