BTech Final Report

Dasarsh Vadugu

Abstract

This project requires the creation and development of a mobile application which can be run on a tablet interface. Its use is centred on acting as an intermediary between users working in a nursery and the information which they gather while doing so. The application will merge the current, paper-based system of collecting information with a tablet interface. This will change the way in which information is gathered while maintaining the same feel and user experience which is analogous to filling out a questionnaire on paper. The existing paper-based system is

prone to several kinds of errors and resultant unstructured data, both of which can be detrimental to users of the data. Questionnaires will be dynamically created based on specifications which the application will be able to read, parse and understand. Integration of cloud services allows for this application to have safe and secure access to the data from anywhere. As a result of this application, time and effort will be saved with regards to collecting information from the field.

Project Outline

Figure 1, Scion.

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.

At Scion, when information is to be collected from the nursery, it is done so on paper. Whatever information is being recorded is done so onto paper, and then this is passed on through many people until it is entered into a digital format represented by an Excel spreadsheet, which is then

passed along to someone else who will then finally enter the information into their systems. This paper-based system is a bottleneck in their operations and deserves to be streamlined.

The main issues with the existing paper-based system are as follows:

1. Illegibility of writing – What's written on a paper form cannot be guaranteed to be interpreted correctly by whoever has the task of entering the data into a spreadsheet. This results in data which can have a high frequency of errors and makes it so that the data is no longer reliable.
2. Validation of input – Another major contributor to unreliable data is the lack of validation. Even if the data recorded is perfectly legible, there are no systems in place to ensure that what has been recorded is appropriate. An example of this is an answer to a question being valid only if it is a number between certain ranges. This is a constraint which cannot be tightly enforced on paper. A consequence of such an issue is incorrect data due to avoidable absentmindedness.
3. Time to get to system – The data collected is passed through the hands of several employees before being entered into a digital format, which is then passed through the hands of more employees before reaching its destination in the system. This is a waste of time and resources, both of which are important to any organisation's operations. Such a delay in the storage of information may have consequences to the organisation especially when dealing with clients who may require some information quickly.
4. Speed of recording information – The speed at which data is collected is something which is also detrimental to the organisation. The pen is slower than the keyboard and although this is can be a minor difference in speed for an individual record, this is something that can add up to become a significant amount of time which is wasted.
5. Unstructured data – Data which is collected correctly, but not in a correct format is a hindrance to operations. This data cannot be placed into a database easily, if it is possible at all. This can prove to be a major issue with organisations as the data is rendered essentially useless.

The substitution of the current paper-based system will eliminate the issues listed above. The addition of a smart device allows for the problems to be mitigated, by means of this application which is to be developed.

1. Illegibility is no longer an issue because all the data entered into the application will be done so via a soft keyboard on the touch screen. Due to this, there is no issue with misinterpretations of what is read.
2. Validation of input becomes possible by polling the fields in which data is to be entered. These fields can be configured to reject input which does not fall within an accepted subset of answers.
3. Time to get to system – The connectivity-enabled device will be able to send and receive data over networks. As a direct result of this, the application will be able to send the data which it collects to the system as soon as it is able. This transfer of data will only be possible in the presence of network connectivity, so if there is no network at that time and place, then the application will wait until there is a network signal which is strong enough for it to perform the action of transferring data to the system where it is to be stored.
4. Speed of recording information – The device's touch screen will display a soft keyboard which will greatly enhance the speed at which information can be input into the application. Keyboards are a faster alternative to the pen and paper.
5. Unstructured data – The application will be able to configure fields in which data is to be entered. This allows for the important pieces of data to be extracted from the user of the device, which ultimately results in structured data which can be entered into a database, allowing for the data to become useful information.

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.

There was a choice to be made regarding the operating system on which this application was to run. The two choices were Apple's iOS and Google's Android.

The decision was one which was easy to make. iOS applications are written in Xcode which is an integrated development environment (IDE) developed by Apple for developing applications for both their mobile and desktop platforms. This requires the use of Objective-

C and their respective compilers for development. Android uses Eclipse as its IDE and the programming language it incorporates to develop is Java.

I have significantly more experience in working with Java, and also have experimented with building Android applications using the Eclipse IDE. Therefore, the decision to work with Android programming in Java was natural.

The Android operating system has several properties integrated into their design which offer some clear advantages for the design of this project. The aim is to reduce and eliminate the need for paper-based systems. For this reason, these Android operating system properties can be exploited to achieve this goal.

1. **The Activity and the Service** - An activity is represented by what's shown on the screen. Each activity is delegated the screen once it has been started and therefore has a user interface. An application may have several activities incorporated in its functionalities. An example of this is a chatting application which may have one activity for viewing all the chats, an activity for beginning a new chat with a person, an activity for adding new people to an existing chat and another activity for the actual chat thread. Activities have the ability to start other activities. Activities work together to create the full functionality of an application.
   In contrast to activities, services run in the background and have no user interface. An example of a service is one which uploads and downloads data behind the scenes without interrupting the user from performing any actions.
2. **The Activity Lifecycle** - The activity lifecycle is a way of describing the series of changes that an activity goes through during after its being run. It can be used to manage the behaviour of the activity via its lifecycle callback methods. For developers, this means that they have control over what happens whenever a user enters an activity or leaves an activity and provides methods to cater to events which should have the ability to pause or stop an activity.

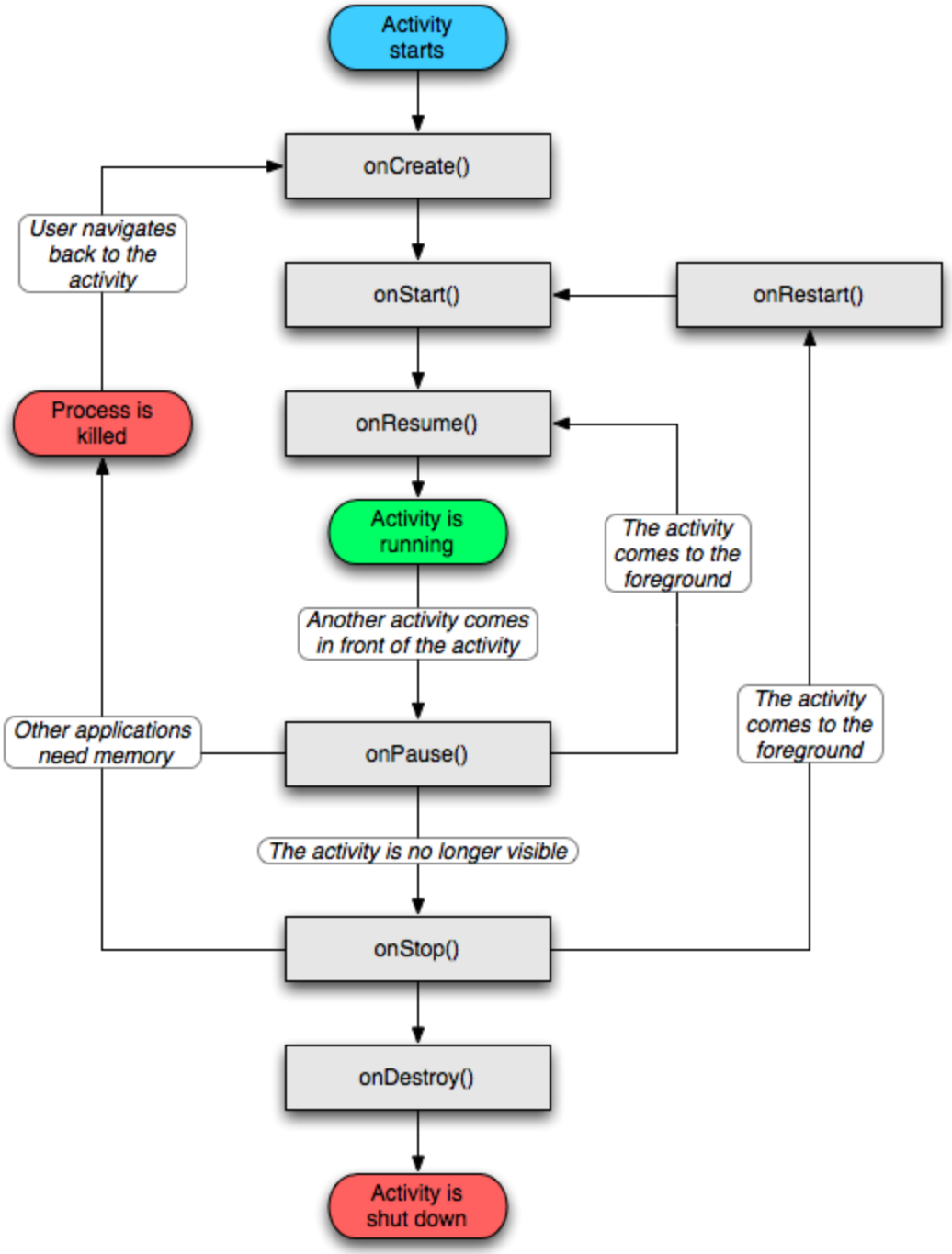


Figure 2, The activity lifecycle, with all the different stages of an activity's life from its starting to its shutting down.

3. **Adaptable Layouts** - The Android operating system is able to manage the layouts that an application has incorporated within it. These layouts are able to scale and

grow appropriately to suit the screen size of the device that the application is being run on. This property is one which is important because of the widespread presence of the Android operating system. It can be run on all sorts of devices and therefore, there are no set screen sizes to which developers may adjust or fine-tune their application to work with as there is with Apple's iOS. An example of this would be an xml layout displaying the same elements, but either larger or smaller as a direct result of the size of the screen. This leads to layouts being universally pleasing to the eye regardless of the device with which the layout is being viewed with.

4. **Media Capabilities** - Applications have access to the media capabilities which are available on the device. Therefore this allows applications to incorporate features requiring either the camera for images or videos, or the audio recorder for audio-related functionality. The touch display can also be considered to be a form of media input and be taken full advantage of when allowing users to draw on top of selected photos, giving them the ability to attract attention or annotate them if necessary.
5. **Inbuilt Sensors** - The sensors built into the devices enable applications to use the operating system to use the information that the sensors gather. Applications can now be aware of their surroundings, locations, temperatures and movements. This collection of readily available information can be taken advantage of by using them to deduce any information which the user would usually be required to do. This reduces the amount of time and effort the user puts into performing their actions.
6. **Connectivity** - Nowadays, devices are almost constantly connected to a network. This is an advantage because web services can be made use of without having to interrupt the user. This is essential for actions such as syncing, uploading or downloading files. Here, the situation where the user is forced to wait is avoided. Android also provides several HTTP clients to allow applications to have the ability to send or receive data.
7. **Validation** - Active policing of data is made possible with a smart device. In contrast to paper questionnaires, a smart device is able to alert the user whenever something erroneous has been entered into a field or if the user is about to perform an action which is illegal or can cause issues in the handling of that information. Such data validation results in data which is structured and cannot be misconstrued. This data can reliably and consistently be transformed to meaningful information.

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

Kupzyk and Cohen (KA Kupzyk, 2014) acknowledge that data which is collected must be done so correctly. Direct consequences of data entry errors are delayed analyses and incorrect conclusions being reached. Two contributors to data entry errors are users entering the incorrect values and users unintentionally skipping questions.
Kupzyk and Cohen proposed that these two factors can be resolved using simple strategies which both require giving suggestions to the user. Incorrect data cannot be entered if users are

given a choice of valid answers to choose from, such as in a dropdown box or a multiple choice scenario. The skipping of items can be prevented by implementing a counter which ensures that all the questions have been answered, and will not allow the user to progress if there are any answers which are missing.
Their system, implemented in Microsoft Excel, resulted in improved reporting in a nursing home facility for reporting incidents of falls.

Kumar et al. (A. M. V. Kumar, 2013) address the fact that most published articles which rely on data collection are not guaranteed to have data which is of a quality that is ensured by the definitive gold standard of double entry and validation. The data is incorrect due to data entry errors which are not accounted for.
The proposed solution to this by Kumar et al. was a system which would allow for those collecting data to enter this data into fields which would be policed by inbuilt checks, thereby reducing the frequency of data entry errors. Another feature which was added to this system was that which used Dropbox to act as a file sharing service with both online and offline capability. Its functionality extended beyond a file sharing service to a service which provided near real-time file synchronisation.
The implementation of their system was a success in that it reduced the frequency of data entry errors, and also positively impacted the time and effort required to work with the system.

Beretta et al. (L. Beretta, 2007) were faced with auditing a database and correcting identified errors. In doing so, they acknowledge the importance of accurate data recording to prevent such errors from being stored. They state that such errors can lead to serious information bias which can be avoided by having consistent data recording.
Upon identifying and correcting erroneous data in their database, Beretta et al. found that data collection being improved by a computer-assisted system would contribute to a better quality of data in databases. They acknowledge that a computer-assisted system would be able to limit errors in data entry by detecting and disallowing data which would not fall within an accepted range.

Coons et al. (Stephen Joel Coons, 2009) investigated the possibility of electronic forms performing differently in terms of the quality of the information which they gather when compared to their paper-based alternatives. Their aim was to deduce and state any clear advantages and/or disadvantages that electronic forms may have over paper-based forms.

Paper-based patient-reported outcome (PRO) measure forms were adapted to an electronic format (ePRO) and their comparability or measurement equivalence was used to juxtapose the quality of the information gathered. So as to not be biased towards any one platform of electronic media, Coons et al. administered the PROs on several screen-based devices. Their results show that there was no bias, indicating that any screen-based device could be used. The data collected from an ePRO was determined to be of equal or superior quality when compared to data collected by its paper-based predecessor. Coons et al. state that administering the ePRO subsequent to the paper measure bears the same results that one would get if the paper measure was administered twice.

Allenby et al. (A. ALLENBY, 2002) address the feasibility of touchscreen devices being used in clinics for the purpose of patients recording information on their own. Their aim was to determine how effective and useful the touchscreen devices will be as an alternative to a paper-based alternative.
The comparison between paper-based and touchscreen-based questionnaires can be made by converting the paper-based and then allowing the questionnaires to be put to use. The results will indicate the feasibility of touchscreen devices being used in the clinic. Allenby et al. state that data becomes more structured when collected in a more regulated fashion, as opposed to systems similar to paper-based ones. It is claimed that making use of electronic media to collect information also increases the speed of the data collection. This could allow for more information to be collected and/or less time being spent on the questionnaires and/or both.
The results showed that the accuracy and completeness of the data which was collected by means of the touchscreen device was excellent. This meant that the method, which eliminated the need for transcription and secondary data entry, was deemed feasible to be used in the clinic.

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.

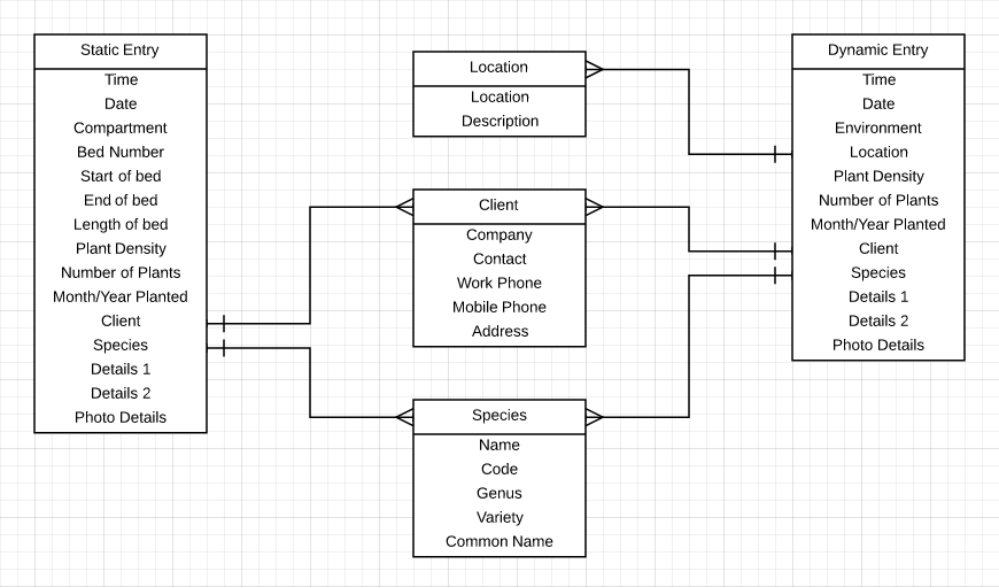


Figure 3, ERD showing the relationship between the data in the different forms

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>
```

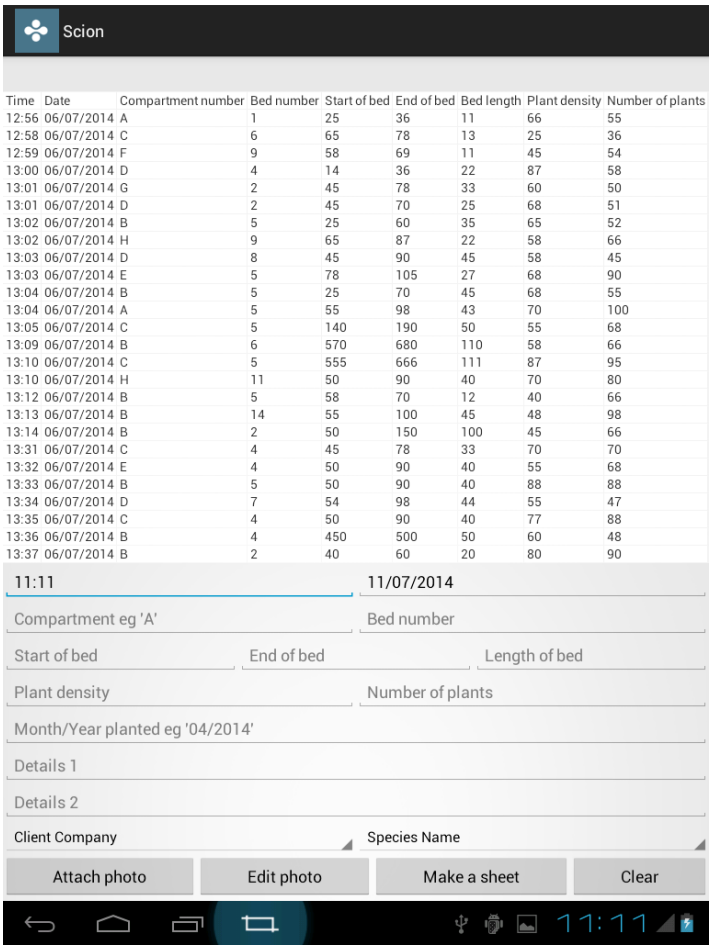


Figure 4, the division of the screen in two for the preview and the form

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>
```

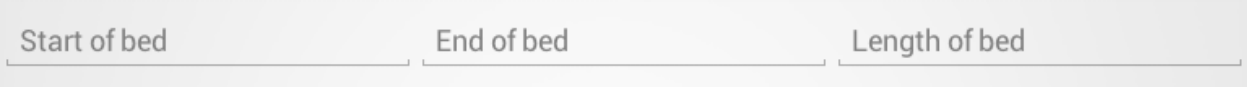


Figure 5, the grouping of the bed-related fields on one line

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" />
```

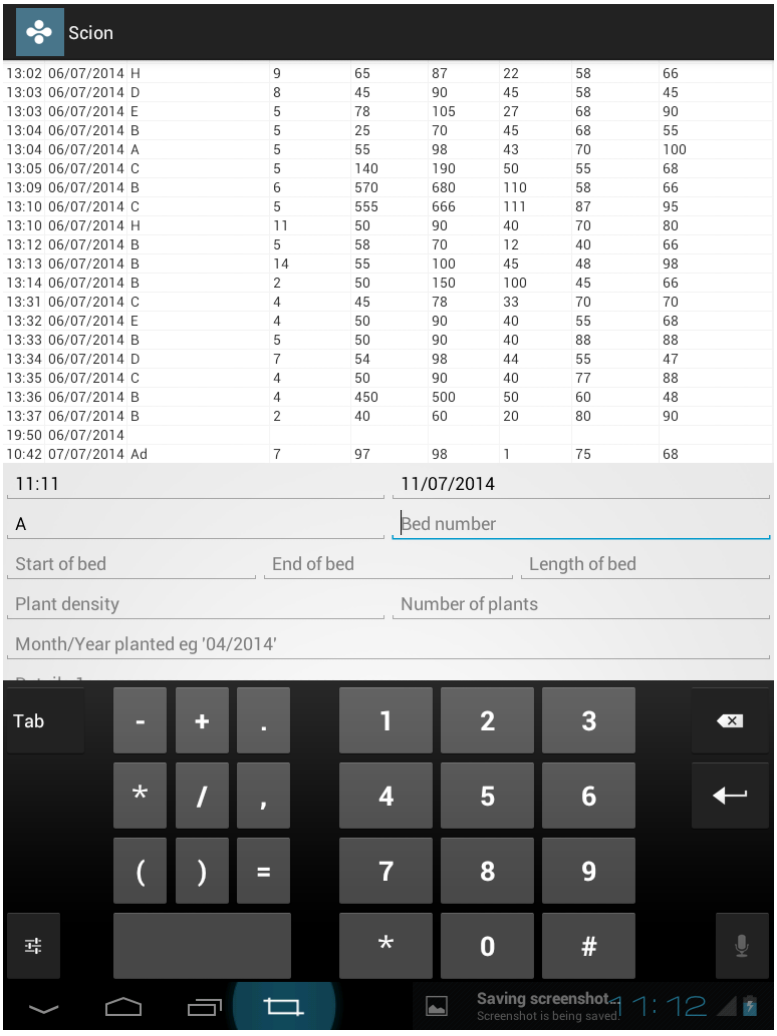


Figure 6, a numberpad is shown for input which should strictly be numeric

`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" />
```

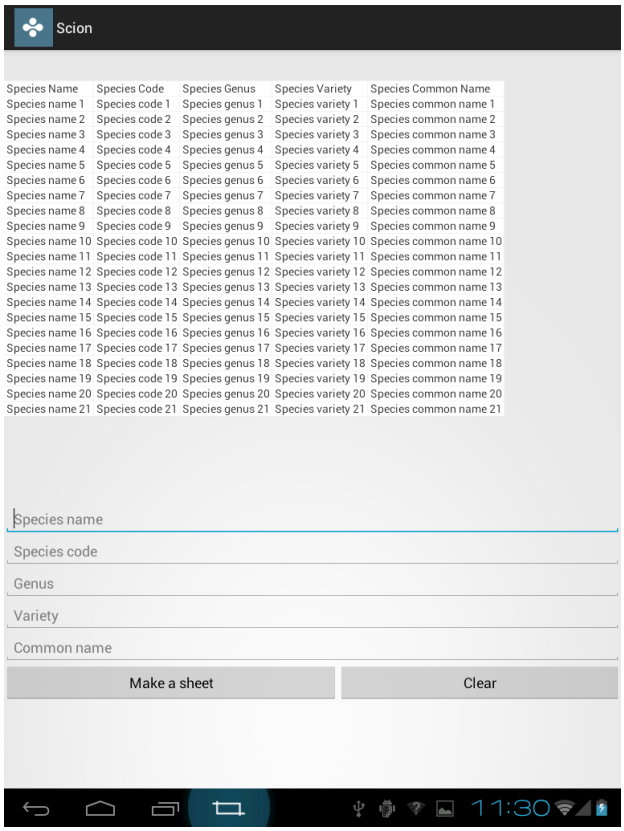
```
        <Button
            android:id="@+id/STATICbClear"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="25"
            android:text="@string/Clear" />
    </LinearLayout>
```

Figure 7, the layout of buttons featured in both Static and Dynamic mapping forms

Figure 8, 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.


Scion
Client Company Client Contact Client Work phone Client Mobile phone Client Address
The Dasarsh Company Dasarsh Vadugu (09)1234567 +6411234567 31 Client Rd
We Want Plants John Planter (09)1234568 +6411234567 32 Client Rd
The Green Party Kim Dotcom (09)1234569 +6411234567 33 Client Rd
Colyplay Chris Martin (09)1234570 +6411234567 34 Client Rd
GeoTech Louis Hamilton (09)1234571 +6411234567 35 Client Rd
Major Plants Olga Gurlukovich (09)1234572 +6411234567 36 Client Rd
Android Prince George (09)1234573 +6411234567 37 Client Rd
Nintendo Homer Simpson (09)1234574 +6411234567 38 Client Rd
Vodafone Avril Lavigne (09)1234575 +6411234567 39 Client Rd
Windows Bill Gates (09)1234576 +6411234567 40 Client Rd
Nascar Lightning McQueen (09)1234577 +6411234567 41 Client Rd
Disney David Beckham (09)1234578 +6411234567 42 Client Rd
Monsters Inc. Mike Mazowski (09)1234579 +6411234567 43 Client Rd
HP Tom Jerry (09)1234580 +6411234567 44 Client Rd
Les Mills Brad Pitt (09)1234581 +6411234567 45 Client Rd
University of Auckland Angelina Jolie (09)1234582 +6411234567 46 Client Rd
WWF Dwayne Johnson (09)1234583 +6411234567 47 Client Rd
National Basketball Association Kobe Bryant (09)1234584 +6411234567 48 Client Rd
Unicef Walter White (09)1234585 +6411234567 49 Client Rd
Skinny Mobile Guysen Lang (09)1234586 +6411234567 50 Client Rd
Orcon Gandalf White (09)1234587 +6411234567 51 Client Rd
Company
Contact name
Work phone
Mobile phone
Postal address
Make a sheet
Clear
11:30


Scion
Species Name Species Code Species Genus Species Variety Species Common Name
Species name 1 Species code 1 Species genus 1 Species variety 1 Species common name 1
Species name 2 Species code 2 Species genus 2 Species variety 2 Species common name 2
Species name 3 Species code 3 Species genus 3 Species variety 3 Species common name 3
Species name 4 Species code 4 Species genus 4 Species variety 4 Species common name 4
Species name 5 Species code 5 Species genus 5 Species variety 5 Species common name 5
Species name 6 Species code 6 Species genus 6 Species variety 6 Species common name 6
Species name 7 Species code 7 Species genus 7 Species variety 7 Species common name 7
Species name 8 Species code 8 Species genus 8 Species variety 8 Species common name 8
Species name 9 Species code 9 Species genus 9 Species variety 9 Species common name 9
Species name 10 Species code 10 Species genus 10 Species variety 10 Species common name 10
Species name 11 Species code 11 Species genus 11 Species variety 11 Species common name 11
Species name 12 Species code 12 Species genus 12 Species variety 12 Species common name 12
Species name 13 Species code 13 Species genus 13 Species variety 13 Species common name 13
Species name 14 Species code 14 Species genus 14 Species variety 14 Species common name 14
Species name 15 Species code 15 Species genus 15 Species variety 15 Species common name 15
Species name 16 Species code 16 Species genus 16 Species variety 16 Species common name 16
Species name 17 Species code 17 Species genus 17 Species variety 17 Species common name 17
Species name 18 Species code 18 Species genus 18 Species variety 18 Species common name 18
Species name 19 Species code 19 Species genus 19 Species variety 19 Species common name 19
Species name 20 Species code 20 Species genus 20 Species variety 20 Species common name 20
Species name 21 Species code 21 Species genus 21 Species variety 21 Species common name 21
Species name
Species code
Genus
Variety
Common name
Make a sheet
Clear
11:30

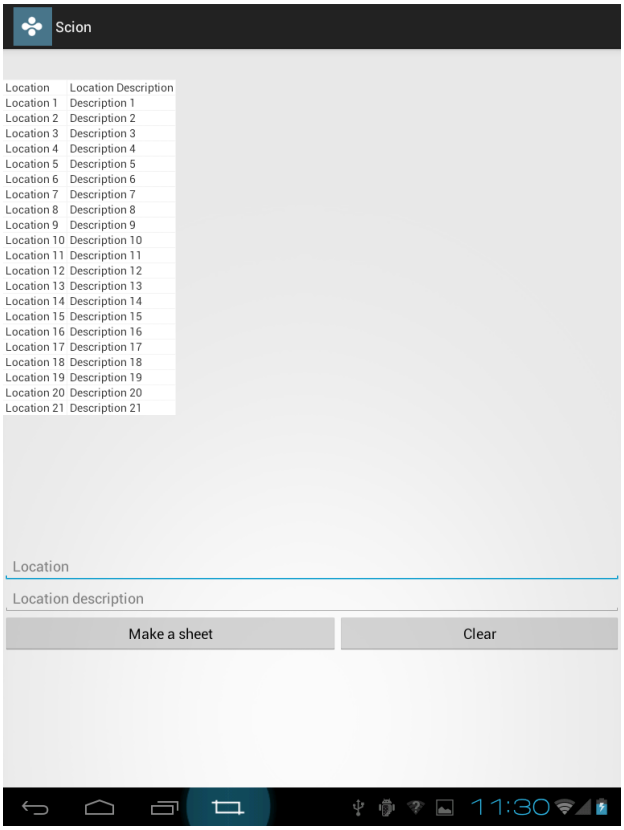

Scion
Location Location Description
Location 1 Description 1
Location 2 Description 2
Location 3 Description 3
Location 4 Description 4
Location 5 Description 5
Location 6 Description 6
Location 7 Description 7
Location 8 Description 8
Location 9 Description 9
Location 10 Description 10
Location 11 Description 11
Location 12 Description 12
Location 13 Description 13
Location 14 Description 14
Location 15 Description 15
Location 16 Description 16
Location 17 Description 17
Location 18 Description 18
Location 19 Description 19
Location 20 Description 20
Location 21 Description 21
Location
Location description
Make a sheet
Clear
11:30

Figure 9, 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.

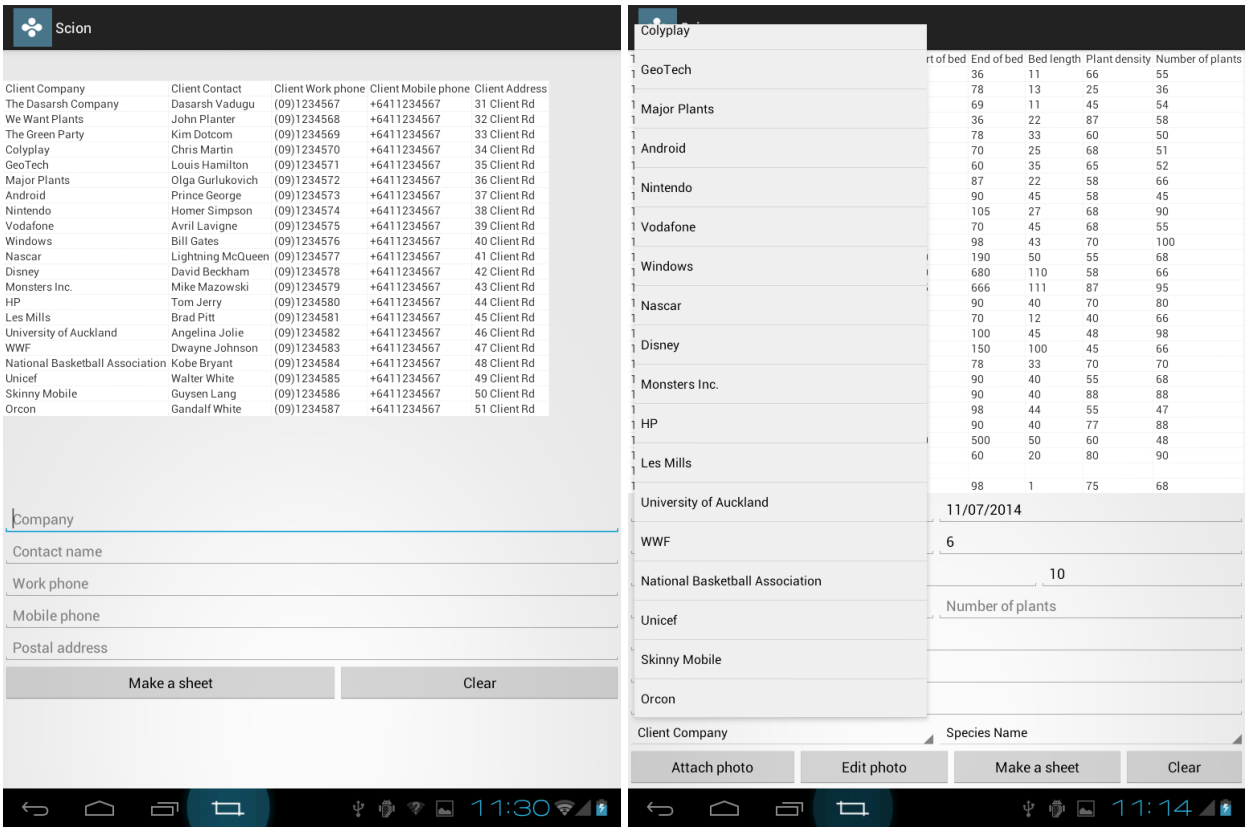


Figure 10, the entries from the Client Company column from the Client sheet are shown as items in the `Spinner` element in the Static form

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();
                    }
                }
            }
```

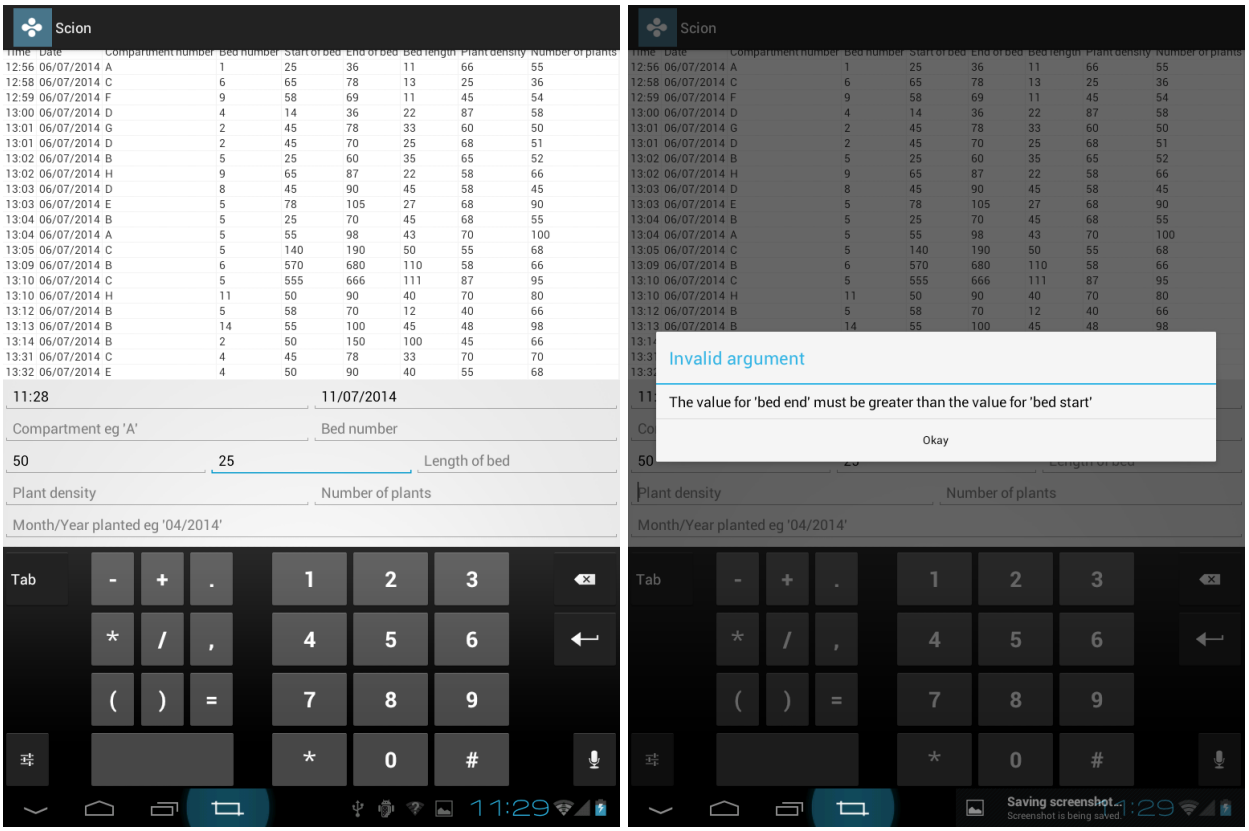


Figure 11, the value for the end of the bed (25) is less than that of the start of the bed (50), and so the user is prompted with an alert

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();
}
```

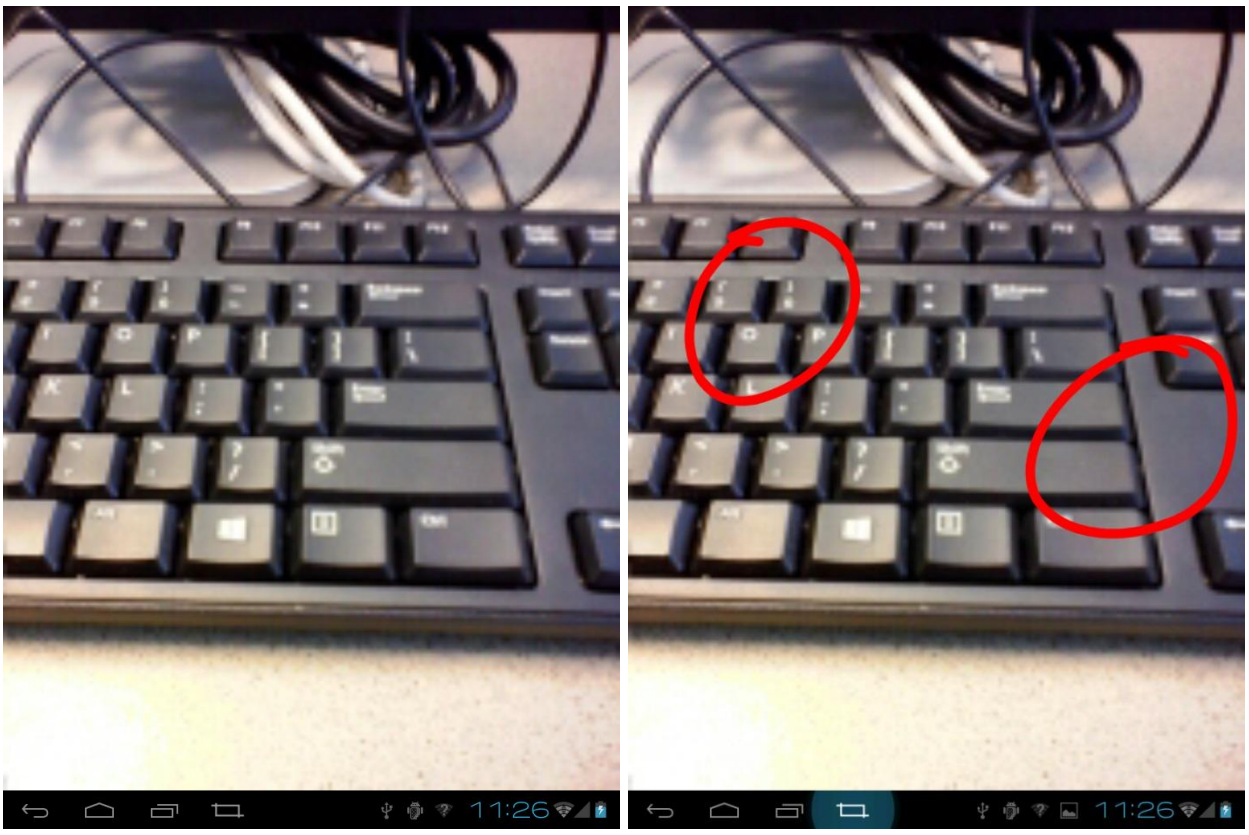


Figure 12, an attached photo can be drawn on to bring attention to any particular area

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);
    }
```

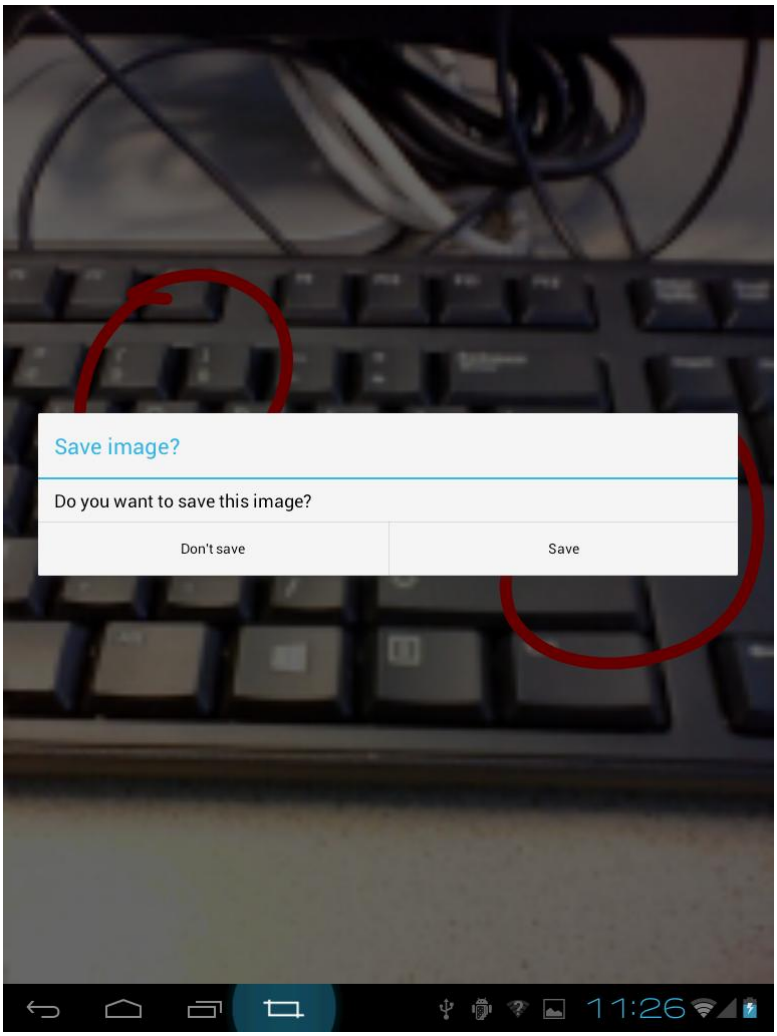


Figure 13, the user is prompted to save or discard the edits that have been made

Abstraction

This application, although achieving most of the goals, was restricted to the very specific forms that were required by Scion. They were hardcoded and therefore limited to only the static and dynamic mapping processes. The level of abstraction could be lifted such that the application could grow into one which could construct any digital questionnaire based on specifications.

Updated Design

Updated Layout Design

With regards to design, the application needed to change in such a way that the preview would no longer consume half of the available screen real estate. This caused a problem with the display in that when the soft keyboard appears so that users may enter input, the screen becomes congested and the preview causes the actual form itself to appear small, making it difficult to navigate through.

The new design required a change in layout, while still keeping the layouts and functionalities of the form and the preview intact. This would give the user a cleaner, unobstructed view of both the questionnaire and the preview.

Questionnaire Specifications

The questionnaire is to be based on a specification and this is required to cater to all the different types of questions. All of these types of questions must be accounted for in the specification which is to be read from and understood by the application.
These are as follows:

1. **Text questions** – These questions require answers which comprise of text only. This can be a question suitable for answers which are names, addresses, descriptions or explanations.
2. **Numeric questions** – These questions require answers which are strictly numeric in format. An example of such a question is one which may ask for a particular number of times that an event has occurred, or one which may ask for the difference of two measurements.

3. **Multiple choice questions** – What is presented is a question and multiple answers from which the user may select one, or more depending on the context. The two different contexts are as follows:
    a. **Single response** – Users may only select one of the presented answers. This is analogous to a question where there are many possible answers, but only one definite answer.
    b. **Multiple response** – Users may select many of the presented answers. This is analogous to a question where there may be more than one definite answer.
4. **Time/Date questions** – The responses to these questions are the time and/or the date. This type of question can be automatically filled in by the application itself.
5. **Location questions** – The field is to be filled with the address of the user. This field is one which can be populated automatically by use of the device's sensors.
6. **Media questions** – A question is asked, to which the answer is an image, a video or an audio recording.

Cloud Service Integration

The new iteration of the application should have access to a cloud service. This allows for files to be loaded from and saved to the cloud. This provides resultant safe and secure access to those files.

There should be the ability to download a specification file from the cloud, which can then be used to display the questionnaire that it details. The questionnaire displayed will correspond to an Excel sheet containing all the data collected from the fields in the questionnaire. Using the application to fill out the fields in the form, the user should be able to update the current sheet, and then upload the sheet to the cloud.

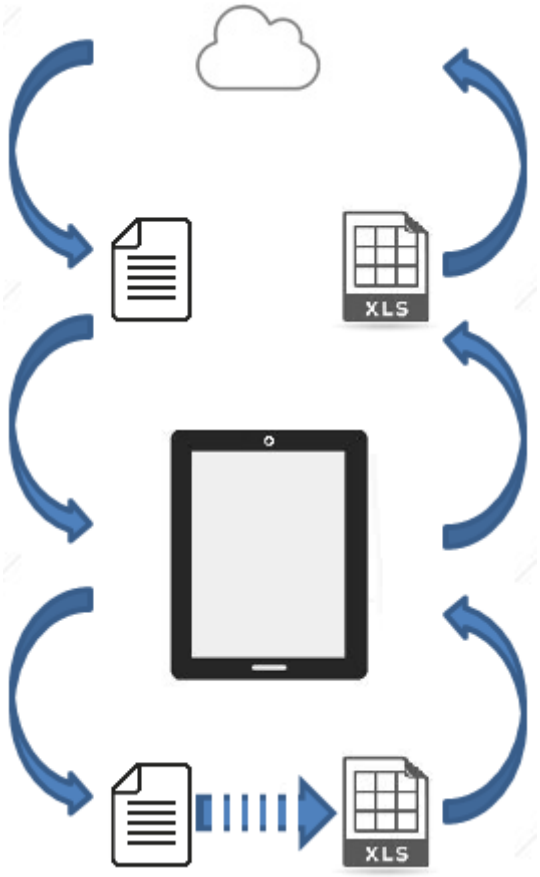


Figure 14, The transfer of data between cloud storage and the device.

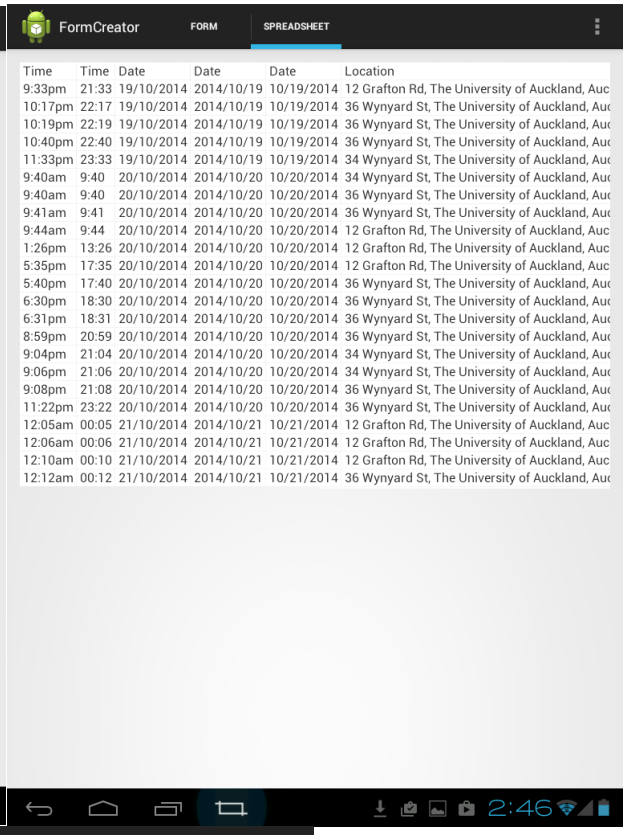
In the case of there being no network connectivity for the sake of uploading sheets to the cloud, there should be an offline synchronisation feature incorporated. This will ensure that even if there is no network connectivity when the sheet should be uploaded to the cloud, the application will make it such that the sheet is uploaded whenever network connectivity is regained.

Updated Implementation

Updated Layout Implementation

In order to present a cleaner and less obstructive view of both the questionnaire and the preview, the whole project was changed to accommodate swipe views with tabs. Swipe views are an effective way of navigating, allowing users to switch between sibling screens using a simple horizontal finger gesture (Google, Android Horizontal Paging). The tab views are called Fragments which are representative of a portion of an Activity's user interface (Google, Android Fragments). There are two Fragments, one for the dynamic questionnaire and the second for the spreadsheet preview.


FormCreator
FORM
SPREADSHEET
Time
2:46pm
Time
14:46
Date
27/10/2014
Date
2014/10/27
Date
10/27/2014
Location
Inside the catch
Pick one
One
Two
Three
Four
Pick some
One
Two
Three
Four
Please select an appropriate photo.
CAMERA
GALLERY
EDIT PHOTO
2:46


FormCreator
FORM
SPREADSHEET
Time Time Date Date Date Location
9:33pm 21:33 19/10/2014 2014/10/19 10/19/2014 12 Grafton Rd, The University of Auckland, Auc
10:17pm 22:17 19/10/2014 2014/10/19 10/19/2014 36 Wynyard St, The University of Auckland, Auc
10:19pm 22:19 19/10/2014 2014/10/19 10/19/2014 36 Wynyard St, The University of Auckland, Auc
10:40pm 22:40 19/10/2014 2014/10/19 10/19/2014 36 Wynyard St, The University of Auckland, Auc
11:33pm 23:33 19/10/2014 2014/10/19 10/19/2014 34 Wynyard St, The University of Auckland, Auc
9:40am 9:40 20/10/2014 2014/10/20 10/20/2014 34 Wynyard St, The University of Auckland, Auc
9:40am 9:40 20/10/2014 2014/10/20 10/20/2014 36 Wynyard St, The University of Auckland, Auc
9:41am 9:41 20/10/2014 2014/10/20 10/20/2014 36 Wynyard St, The University of Auckland, Auc
9:44am 9:44 20/10/2014 2014/10/20 10/20/2014 12 Grafton Rd, The University of Auckland, Auc
1:26pm 13:26 20/10/2014 2014/10/20 10/20/2014 12 Grafton Rd, The University of Auckland, Auc
5:35pm 17:35 20/10/2014 2014/10/20 10/20/2014 12 Grafton Rd, The University of Auckland, Auc
5:40pm 17:40 20/10/2014 2014/10/20 10/20/2014 36 Wynyard St, The University of Auckland, Auc
6:30pm 18:30 20/10/2014 2014/10/20 10/20/2014 36 Wynyard St, The University of Auckland, Auc
6:31pm 18:31 20/10/2014 2014/10/20 10/20/2014 36 Wynyard St, The University of Auckland, Auc
8:59pm 20:59 20/10/2014 2014/10/20 10/20/2014 36 Wynyard St, The University of Auckland, Auc
9:04pm 21:04 20/10/2014 2014/10/20 10/20/2014 34 Wynyard St, The University of Auckland, Auc
9:06pm 21:06 20/10/2014 2014/10/20 10/20/2014 34 Wynyard St, The University of Auckland, Auc
9:08pm 21:08 20/10/2014 2014/10/20 10/20/2014 36 Wynyard St, The University of Auckland, Auc
11:22pm 23:22 20/10/2014 2014/10/20 10/20/2014 36 Wynyard St, The University of Auckland, Auc
12:05am 00:05 21/10/2014 2014/10/21 10/21/2014 12 Grafton Rd, The University of Auckland, Auc
12:06am 00:06 21/10/2014 2014/10/21 10/21/2014 12 Grafton Rd, The University of Auckland, Auc
12:10am 00:10 21/10/2014 2014/10/21 10/21/2014 12 Grafton Rd, The University of Auckland, Auc
12:12am 00:12 21/10/2014 2014/10/21 10/21/2014 36 Wynyard St, The University of Auckland, Auc
2:46

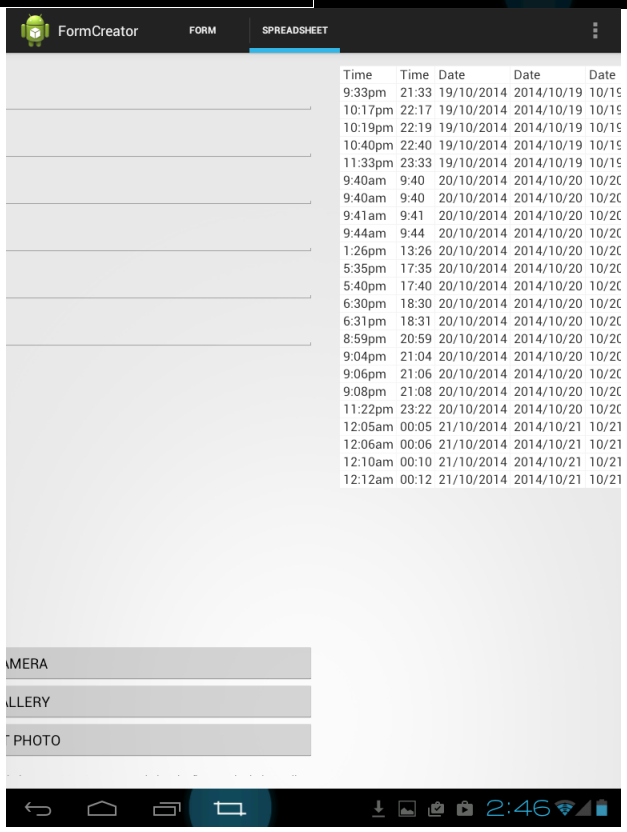

FormCreator
FORM
SPREADSHEET
Time Time Date Date Date
9:33pm 21:33 19/10/2014 2014/10/19 10/19
10:17pm 22:17 19/10/2014 2014/10/19 10/19
10:19pm 22:19 19/10/2014 2014/10/19 10/19
10:40pm 22:40 19/10/2014 2014/10/19 10/19
11:33pm 23:33 19/10/2014 2014/10/19 10/19
9:40am 9:40 20/10/2014 2014/10/20 10/20
9:40am 9:40 20/10/2014 2014/10/20 10/20
9:41am 9:41 20/10/2014 2014/10/20 10/20
9:44am 9:44 20/10/2014 2014/10/20 10/20
1:26pm 13:26 20/10/2014 2014/10/20 10/20
5:35pm 17:35 20/10/2014 2014/10/20 10/20
5:40pm 17:40 20/10/2014 2014/10/20 10/20
6:30pm 18:30 20/10/2014 2014/10/20 10/20
6:31pm 18:31 20/10/2014 2014/10/20 10/20
8:59pm 20:59 20/10/2014 2014/10/20 10/20
9:04pm 21:04 20/10/2014 2014/10/20 10/20
9:06pm 21:06 20/10/2014 2014/10/20 10/20
9:08pm 21:08 20/10/2014 2014/10/20 10/20
11:22pm 23:22 20/10/2014 2014/10/20 10/20
12:05am 00:05 21/10/2014 2014/10/21 10/21
12:06am 00:06 21/10/2014 2014/10/21 10/21
12:10am 00:10 21/10/2014 2014/10/21 10/21
12:12am 00:12 21/10/2014 2014/10/21 10/21
AMERA
LLERY
PHOTO
2:46

Figure 15, The two slide tabs contain the questionnaire and preview sections respectively. Sliding between the two gives a smooth interface where a user can easily switch their view.

This meant that the minimum SDK version had changed from 8 to 11 to accommodate this. Therefore, a device is required to be running an Android version of 3.0 or higher in order to run this application.

As a result of this, users can easily switch between the two tabs. This preserves the functionality of both the form and the preview components. The preview has also been modified so that on pressing one of the cells, the cell's respective row is highlighted. This is a minor adaptation made for the ease of use for the user.

Newly Imported APIs

The Dropbox Sync API requires libraries which contain the necessary Java files regarding the different data structures and exceptions related to the Sync API. This library was imported.

For the Chooser to work, the Chooser SDK project was also imported. This project was added to the form creator application as a dependency, which allowed the application to use the Chooser to navigate through a user's Dropbox account.

Updated Android Manifest

As a result of the new project layout design, the manifest itself was subject to change. The changes are as follows:

A package name of com.dasarsh.formcreator.

A minimum SDK of 11 which correlates to an Android version of 3.0.

A target SDK of 18 which correlates to an Android version of 4.3.

The permissions:

1. "android.permission.READ_EXTERNAL_STORAGE" and "android.permission.WRITE_EXTERNAL_STORAGE" for the sake of reading and writing from the memory card.

2. “android.permission.ACCESS_FINE_LOCATION” and “android.permission.ACCESS_COURSE_LOCATION” in order for the application to have access to the location data which can be deduced by the device.
3. “android.permission.INTERNET” AND “android.permission.NETWORK_STATE” so that the application is able to test the state of the network and have access to the Internet if it is available.
4. “android.permission.WAKE_LOCK” used to allow the service to run even if the screen is off.

An activity for each of the following:

1. XMLReader
2. XMLElement
3. DbxAuthActivity
4. AuthActivity
5. EditPhotoSurface

A service for:

1. DbxSyncService

Updated XML Layouts

The XML layouts are no longer static due to the newly dynamic nature of the application. As a result of this, there are only two XML layouts, both of which are nearly empty.

The two XML layouts, `activity_xmlreader.xml` and `fragment_xmlreader_dummy.xml`, are linked to the main `XMLReader` activity and for the fragments which it uses, respectively.

The activity_xmlreader.xml layout file contains the default `ViewPager` item which is often used to encase Fragments (Google, Android ViewPager). This is empty as it will be populated with the Fragments and there is no need to make changes to this.

```
<android.support.v4.view.ViewPager
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```
        tools:context=".XMLReader" >
    </android.support.v4.view.ViewPager>
```

The `fragment_xmlreader_dummy.xml` layout file contains a `RelativeLayout` which contains a `LinearLayout`. This inner `LinearLayout` is important because it is here that, depending on the Fragment chosen, the layout is populated either with the dynamically generated questionnaire or the spreadsheet.

```
    <RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingBottom="@dimen/activity_vertical_margin"
        android:paddingLeft="@dimen/activity_horizontal_margin"
        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        tools:context=".XMLReader$DummySectionFragment" >

        <LinearLayout
            android:id="@+id/layoutXML"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="vertical" >
        </LinearLayout>

    </RelativeLayout>
```

Improved Photo Editing

The photo editing functionality with which users are able to draw on top of taken photos was previously displaying an image which was of notably low quality. This problem was caused by the image being saved into the local memory in a way which was compressing the image as much as possible. This was addressed by allowing the whole image to be saved and used without any compression. As a result of this, the image which can be drawn on top of is now of full quality, just as the user had taken it.

```
OutputStream stream;
try {
	Bitmap bmp = MediaStore.Images.Media.getBitmap(
		getActivity().getContentResolver(), imageUri);
	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
	if(!dateTimeStringValid){
		setupTimeAndDate(true, new EditText(getActivity()), true,
		0);
	}
	stream = new FileOutputStream("/sdcard/scion/photos/"
			+ dateTimeString + ".png");
	bmp.compress(CompressFormat.PNG, 100, stream);
	stream.flush();
	stream.close();
} catch (FileNotFoundException e) {
	e.printStackTrace();
} catch (IOException e) {
	e.printStackTrace();
}
```

XML Specification File and XSD

The application is required to dynamically generate a questionnaire from a file which specifies the questions and formats in the questionnaire. This specification file was decided to be an Extensible Markup Language (XML) file. The XML file is to be constructed from an XML Schema Definition (XSD) which declares all the rules regarding what is legal and illegal when creating a well-formed XML file which is applicable to this application. In addition to the discussed types of questions being included in the schema, an option to display just text was also implemented.

The XSD includes rules and regulations regarding video and audio questions. Due to time constraints, this feature was not implemented and will be skipped over when the application is performing a parse of the input xml file. Therefore, even if there is a question which requires a video or audio answer in the XML specification file, it will not be displayed in the dynamically generated questionnaire on the screen.

Representation of Question Types

All of the questions are represented by a question portion and an answer portion. The question portion is reflects the actual question, e.g. “What is your name?” or “How many days has it been since your last visit?” The answer portion is generated depending on the type of question.
The answer portion is defined as follows:

1. **Text questions** – Text answers are represented by a single `EditText`. As this type of question is general and open to any kind of input, there are no restrictions placed on what the user enters.
2. **Numeric questions** – An `EditText` is generated below the question and it is here that the answer is to be given. Numeric answers must strictly be of numeric format so the soft keyboard will display the numberpad as opposed to the usual QWERTY keyboard layout.
3. **Multiple answer** – The answer portion is defined depending on the type of multiple choice question:
    a. **Single response** – The answers are represented as radio buttons. Of these buttons, only one may be selected at any one time.
    b. **Multiple response** – The answers are represented as checkboxes. Of these checkboxes, one or more may be selected at any one time.
4. **Time/Date questions** – The time answer is shown as an `EditText` which has been populated by the current time. This time is the system time of the device. This `EditText` is editable, which allows for the user to correct any errors.
5. **Location questions** – The location answer is an `EditText` which is automatically populated by the system using the location which is deduced from the GPS coordinates of the device. The location shown is the address of the device’s position. This `EditText` can be edited, which allows for any incorrect locations to be corrected.
6. **Media questions** – For this implementation, there is only functionality for attaching images. Media questions are represented by three buttons. The first of the three buttons is the button which allows users to take a photo. This opens the device’s native camera application and proceeds to allow the user to take a photo of whichever subject they wish. The second button is one which allows users to select a photo which has already been taken. Pressing this button opens the device’s default photo gallery application from which the user is able to select a photo. The final button allows users to edit the photo which they have either just taken using the camera or has selected from their

gallery. This leads the user to a screen where they are able to draw on top of their selected photo as they see fit.

Parsing the XML Specification File

The XML file is parsed so that it can be understood by the application, only then can the questionnaire be generated dynamically by specifications in the XML file. The parsing is performed by the application and each question from the XML file is made into an `XMLElement`. All of these XMLElements are placed into an array in the order that they appear in the XML file and once that is done, they can all be generated into the constituent questions and answers that are specified in the XML file.

Figure 16, The method in which a question specified in the XML file is parsed to an `XMLElement` object which is processed into a question which can then be displayed.

XMLElement.java

An `XMLElement` is a custom Java class which was created in order to represent the questions which are read from the XML specification file. While reading the XML file, the application uses the information it gathers about each question specified in the XML code to create and populate an array of XMLElements.

An XMLElement has six properties;

1. **Type** – Defines and used to differentiate between the different types of questions.
2. **Attributes** – A collection of the attributes which have been read.
3. **Items** – A collection of the items which can appear as options which users can choose from.
4. **Question** – The question which is being to be answered.

5. **Label** – The string of text which is to be displayed if the element is a label as opposed to a question.
6. **Index** – A number assigned to each number which is used in order to keep track of the questions when placed in an array.

Cloud Service Implementation

Cloud services were implemented using the Dropbox API. The Dropbox API offered some useful utilities which proved to make this application able to perform its goals. The application makes use of the Chooser Drop-in and the Sync API.

Both the Sync API and the Chooser Drop-in require the user to have a Dropbox account which is linked to the application. This is ensured every time the user presses the button which lets them add a new row to the sheet.

The Chooser Drop-in allows for the application to display a chooser. Using this, the user may browse through their full Dropbox account file system to navigate to the specification file which they wish to load into the application.

The Chooser is initialised and launched as such:

```
DbxChooser mChooser;
…
mChooser = new DbxChooser(APP_KEY);
mChooser.forResultType(DbxChooser.ResultType.DIRECT_LINK).launch(this,
DBX_CHOOSER_REQUEST);
```

Here, the *APP_KEY* is the key which is assigned to the application once it has been registered on your Dropbox account. The integer value for *DBX_CHOOSER_REQUEST* can be any number so long as you handle this activity using the same number in the `onActivityResult` method.

On beginning this activity which launches the Chooser, the user is shown a Chooser in the Dropbox style and layout. From here, the user is to select the XML specification file of their choice. This will cause the selected file to be downloaded into the device's local memory.

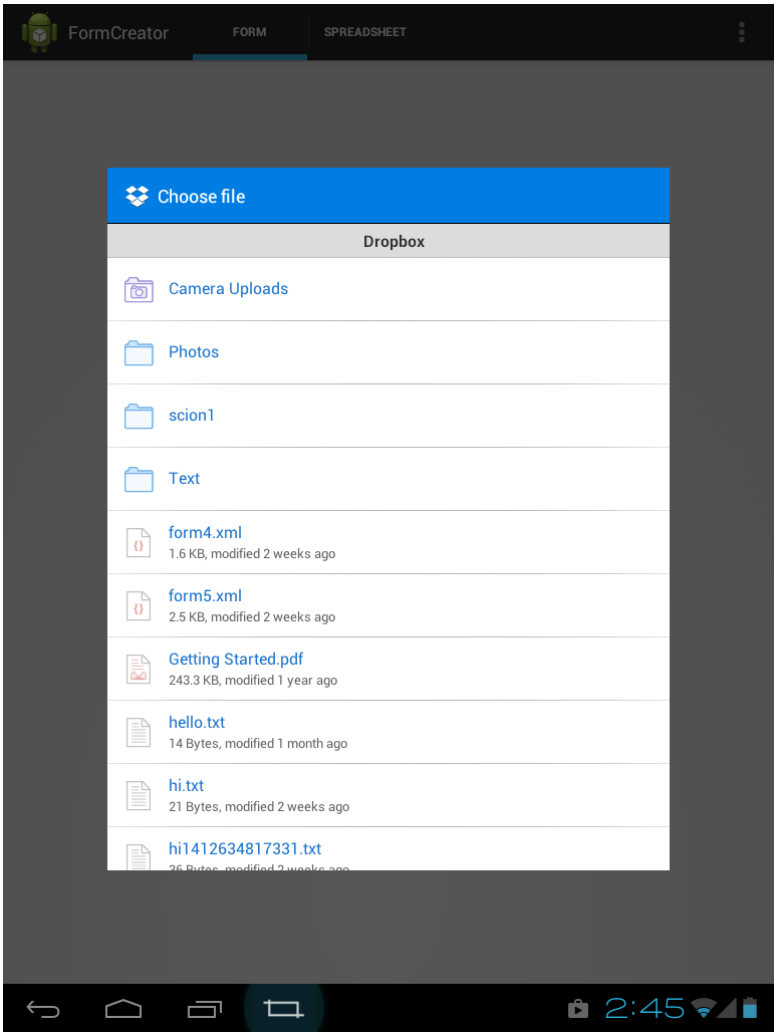


Figure 17, The Dropbox Chooser API used to navigate the user's Dropbox storage.

This is done in the handling of the call to the Chooser in the `onActivityResult` method. Here, the application checks all the finished results from activities and differentiates them according to the identifier which was given when they were launched. In this case, the identifier given was *DBX_CHOOSER_REQUEST* which was simply a number.

```
static final int DBX_CHOOSER_REQUEST = 0; // Can be anything
```

In the handling of this request, the URL specifying the file which was chosen is used in order to download the file from the user's Dropbox account over the network. This file is downloaded and stored into a folder based in the device's local root memory called "scion".

```
	} else if (requestCode == DBX_CHOOSER_REQUEST) {
		if (resultCode == Activity.RESULT_OK) {
			DbxChooser.Result result = new DbxChooser.Result(data);
			// Handle the result

			// DOWNLOAD THE URL FROM GETLINK()FOR DOWNLOAD
			final DownloadTask downloadTask = new
DownloadTask(getActivity());
			downloadTask.setFilename(result.getName());
			xmlFile += result.getName();
			filename = result.getName();
			saveFilename = filename.substring(0, filename.indexOf('.'))
+ ".xls";
		downloadTask.execute(result.getLink().toString());

		}else {
			// Failed or was cancelled by the user.
			Toast toast = Toast.makeText(getActivity(),
					"Failed or was cancelled by the user",
					Toast.LENGTH_LONG);
			toast.show();
		}
	}
```

Once this has been done, the application proceeds to process the file which has been downloaded and parse the XML file into questions.

The Sync API provides a way for the application to access the user's Dropbox account, and also to download and upload files.

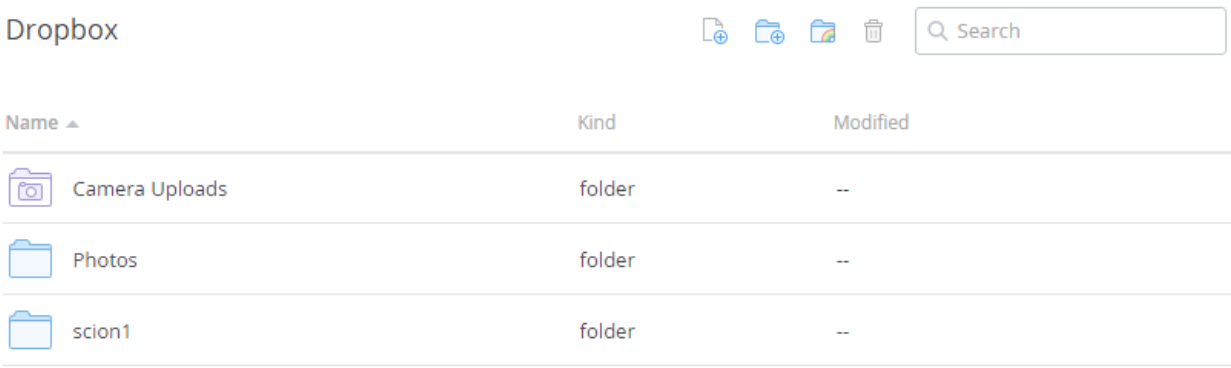
This application uses this Sync API for the purpose of uploading spreadsheets to the user's Dropbox account. This is done whenever the user presses the Submit button, which has a listener set to it which in turn uploads the most recent revision of the spreadsheet which the user has been working on to their Dropbox account.

```
	submit.setOnClickListener(new OnClickListener() {
		@Override
```

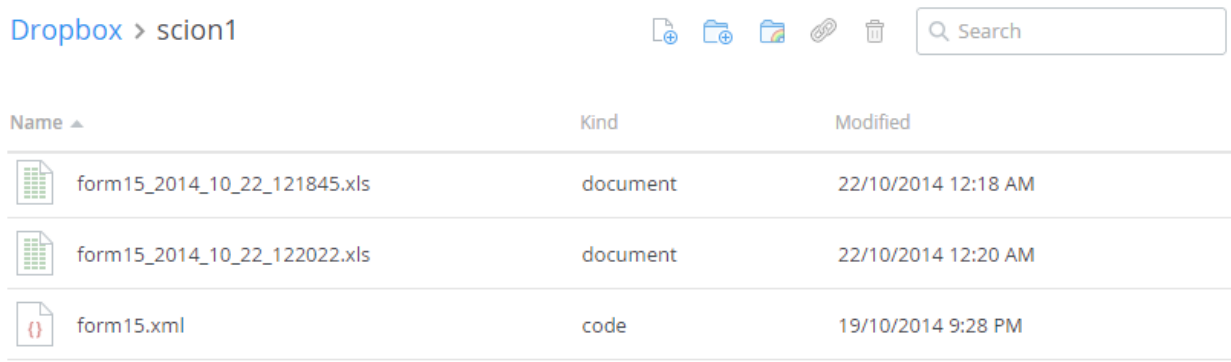
```
            public void onClick(View view) {
                  …
                  accessDropBox();
            }
      });
```

The `accessDropBox` method takes the current spreadsheet and adds a timestamp to the end of the filename and then uploads it to the user's Dropbox storage. This is currently being statically saved to a folder named "scion1" in the root of the Dropbox storage. This is due to the fact that there is no implementation of the Dropbox Saver API for the Android operating system.

```
      private void accessDropBox() {
            try {
                  DbxFileSystem dbxFs = DbxFileSystem.forAccount(mDbxAcctMgr
                              .getLinkedAccount());
                  String name = saveFilename.substring(0,
saveFilename.indexOf('.')) + "_" + dateTimeString + ".xls";
                  DbxFile testFile = dbxFs.create(new DbxPath("/scion1/" +
name));
                  try {
                        File f = new File("sdcard/scion/" + saveFilename);
                        testFile.writeFromExistingFile(f, false);
                  } finally {
                        testFile.close();
                  }
            } catch (Unauthorized e) {
                  e.printStackTrace();
            } catch (InvalidPathException e) {
                  e.printStackTrace();
            } catch (DbxException e) {
                  e.printStackTrace();
            } catch (IOException e1) {
                  e1.printStackTrace();
            }
      }
```



The scion1 folder located in the root is where the .xls spreadsheet files are uploaded to.



Within the scion1 directory, there is an example of an XML specification file, form15.xml, and two revisions of spreadsheets which have been created from that specification. The revisions are appended with a timestamp for the sake of differentiating between them.

Communicator.java

This class is used for the communication between Fragments. The need for this arose when looking for a way for one Fragment to invoke a method which was to be executed in another Fragment. In particular, the intended operation was for the preview in the second Fragment to update every time the Fragment in which it was enclosed was selected. The `Communicator` interface can be used to invoke methods from a separate Fragment, as it cannot be done directly by referencing the separate Fragment.

`Communicator.java` is an interface which contains only the `setupPreview` method.

The XMLReader class implements this `Communicator` interface. It is for the sake of use in the nested `DummySectionFragment` class where it is initialised as follows:

```
Communicator comm;
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                  Bundle savedInstanceState) {
      …
      comm = (Communicator)getActivity();
      …
}
```

On clicking the “Add row” button which will add the row to the sheet, the `Communicator` object will be prompted to update the preview by calling the `setupPreview` method.

```
add.setOnClickListener(new OnClickListener() {
      @Override
      public void onClick(View view) {
            …
            comm.setupPreview();
            …
      }
});
```

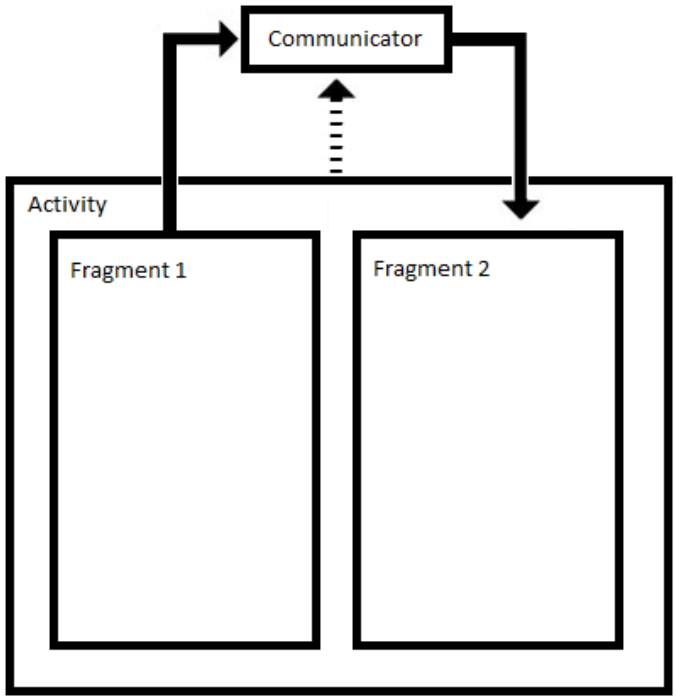


Figure 18, The process of communication between Fragments by use of the Communicator which is implemented by the Activity

Conclusion

This application allows for a form or questionnaire to be digitalised. This results in a method of data collection which is free of errors and ensures structured data. Human effort and time is also saved by use of this application as it can reduce the amount of time that a user spends giving or collecting details while offering a simple way to entered information into the device.

The implementation of this application on a smart device allows for several advantages over paper-based systems such as control over the interface of the device; the ability to make use of the media capabilities such as the camera and the audio recorder; the ability to use the data collected from the device's inbuilt sensors for information such as time, date, location, movement or temperature; the fact that there is usually constant connectivity to networks; and that smart devices offer a way to police the data being entered for inconsistencies or illegal values.

This solution is very versatile. It can be used to digitise any sort of form due to its capability to handle all types of questions. A likely scenario is the administrator of a questionnaire sharing the XML specification file with their chosen participants by email, which saves the XML specification file into the participants' Dropbox accounts. From there, the participants only need to open the application and load the specification file from their Dropbox account and complete the questionnaire.

The paper-based system is eliminated as a direct result of this solution. This reduces the need for printing, distribution and collection which can be hindrances in terms of cost, effort and time. The data received and sent is done so over the Dropbox cloud service which provides safe and secure access to the data stored.

This application, originally intended to solve a singular problem can change the way that data collection is performed. There is a need to eliminate paper-based systems due to all the inherent flaws and this is what can be achieved.

Future Work

There are still some features which have not been implemented as of yet. One of these is one which allows for users to 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 context is the photo editing functionality, as a user may wish to undo whatever they have drawn and try again.

Adaptations which could be made include modifying the application such that the resultant sheet is uploaded onto the servers of the administrators or the questionnaire as opposed to a Dropbox account. This allows for a seamless process where administrators can send the questionnaire specification to participants who can dynamically generate them and subsequently have their answers sent back to the administrators of the questionnaire.

Another addition which could be made is one which allows for the questions to be grouped together and categorized as dependent on one another. An example of this is one which groups three fields together where each of these fields is to have numeric data entered into them. The difference between the first two fields is to be used to calculate and populate the third field.

Questions specified in the XML file can possibly be adapted to include a specification of how much of the screen width they are to occupy. The current implementation delegates each question the whole width of the screen, which can be a potential waste of space if the questions are one which ask very short questions and require very short answers.

The way in which the spreadsheet tab which is displaying the preview is operating can be modified. The way in which the preview is being updated has the potential to cause large lags in operation. This is because the whole spreadsheet is being read and then the whole preview is being loaded and displayed again. This can be avoided if the numbers of rows read and displayed are kept count of. Then only the new rows can be added as opposed to replacing the whole preview with much of the same data.

Bibliography

A. ALLENBY, D. M.-C. (2002). The application of computer touch-screen technology in screening for psychosocial distress in an ambulatory oncology setting.

A. M. V. Kumar, B. N. (2013). Efficient, quality-assured data capture in operational research through innovative use of open-access technology.

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

Google. (n.d.). *Android Fragments*. Retrieved October 24, 2014, from Android Developer: http://developer.android.com/guide/components/fragments.html

Google. (n.d.). *Android Horizontal Paging*. Retrieved October 24, 2014, from Android Developer: http://developer.android.com/training/implementing-navigation/lateral.html#horizontal-paging

Google. (n.d.). *Android ViewPager*. Retrieved October 24, 2014, from Android Developer: http://developer.android.com/reference/android/support/v4/view/ViewPager.html

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

KA Kupzyk, M. C. (2014). Data Validation and Other Strategies for Data Entry. 1-11.

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

L. Beretta, V. A. (2007). Improving the quality of data entry in a low-budget head injury database.

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

Stephen Joel Coons, P. C. (2009). Recommendations on Evidence Needed to Support Measurement Equivalence between Electronic and Paper-Based Patient-Reported Outcome (PRO) Measures: ISPOR ePRO Good Research Practices Task Force Report.

Appendix

formcreator.xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="form">
      <xs:complexType>
         <xs:choice minOccurs="0" maxOccurs="unbounded">

               <xs:element name="time"  >
                  <xs:complexType>
                     <xs:attribute name="format" type="xs:string"/>
                  </xs:complexType>
               </xs:element>

               <xs:element name="date" >
                  <xs:complexType>
                     <xs:attribute name="format" type="xs:string"/>
                  </xs:complexType>
               </xs:element>

               <xs:element name="location"  />

               <xs:element name="question" >
                  <xs:complexType>
                     <xs:sequence>
                        <xs:element name="questionQ" type="xs:string"/>
                        <xs:element name="item" type="xs:string" minOccurs ="0
maxOccurs="unbounded"/>
                     </xs:sequence>
                     <xs:attribute name="type" type="xs:string"/>
                  </xs:complexType>
               </xs:element>
```

```
                <xs:element name="photo" >
                   <xs:complexType>
                      <xs:sequence>
                         <xs:element name="questionQ" type="xs:string"/>
                      </xs:sequence>
                   </xs:complexType>
                </xs:element>

                <xs:element name="display_text" type="xs:string" >
                </xs:element>

             </xs:choice>
       </xs:complexType>
    </xs:element>
</xs:schema>
```