A Seal Data Entry Android App

Samuel Beecher
CS470 – Project Writeup

April 28th, 2011

Table of Contents

Abstract-- 3
1. Introduction--- 3
2. Project Overview--- 3
    2.1 Data Files--- 3
    2.2 Prior Data Entry Process-- 5
3. Project Requirements-- 6
    3.1 Functional Specifications--- 6
        3.1.1 General Specifications-- 7
        3.1.2 Observation Form Functional Specifications------------------------------ 7
        3.1.3 Count Form Functional Specifications----------------------------------- 7
    3.2 System Specifications--- 8
4. System Design--- 8
    4.1 Android Architecture-- 8
    4.2 User Interface Design--- 9
    4.3 Data Structures--- 13
    4.4 System Architecture--- 16
    4.5 Algorithms--- 18
5. Software Development Process--- 19
    5.1 Testing and Debugging-- 19
    5.2 Prototyping Process-- 20
    5.3 Work Breakdown--- 20
6. Results-- 21
    6.1 Future Steps-- 21
7. Summary and Conclusions--- 21
8. References-- 22

Abstract

Dr. Testa, working with the National Marine Mammal Laboratories, collects Fur Seal statistics on the Probilof Archipelago Islands of St. George and St. Paul. These statistics are collected either by performing *Observations* or *Counts.* They are then used by the National Marine Mammal Laboratory to help determine why the fur seal population has been in decline since 1988. The Fur Seal Project was developed to replace the paper and pen system currently used to record *Observations* and *Counts*. This data is then exported in a CSV file for import to a Microsoft Excel document that is used to perform the statistical analysis. The User Interface is meant replace the pen and paper method for data entry currently employed on the field. This will hopefully produce more accurate results for use in statistics.

1. Introduction

This project was developed for Dr. Ward Testa, a Wildlife Biologist for National Marine Mammal Laboratory (NMML). The NMML has been monitoring Fur Seal populations on the Probilof Islands Archipelago of St. George and St. Paul. These islands account for 70% of Northern Fur Seal population in the world. Northern Fur Seals were declared depleted in 1988. These studies are to help identify why the population continues to decline. More detailed monitoring began around the year 2000. NMML monitors Northern Fur Seal population size, age, sex and natural mortality rate. This data is collected through two forms: *Observations* and *Counts*.

2. Project Overview

The goal of this project is to develop an Android App that can be used on the field to collect data. The application will replace the current pen and paper recording system developed by Dr. Testa. Ideally, the application will contain all the necessary protocols of the paper and pen system while adding additional convenient features, such as error checking. However, analysis of this data was not included in the projects scope.

2.1 Data Files

Dr. Testa created and manages all Fur Seal data via a single Microsoft Excel Sheet. This sheet is separated into different sheets that contain more specific data. Of these seven sheets, only four applied to this project. The four applicable sheets are described below:

- MasterResighting. This sheet contains all *Observations* over the current year.
- RawCounts. This sheet contains all *Counts* over the current year.
- Counts. This sheet contains the count totals for each day. Values are derived by summing values for each day in “RawCounts”.

All data, with the exception of “Counts”, is obtained through either *Observations* or *Counts*.

A second Microsoft Excel document was created during the course of the project to manage data internal to the phone. This document was also separated into five sheets, described below:

- IDLookup. This sheet contains all existing seals and their corresponding tags. Each seal has at least one tag. Tag ID's are a concatenation of type, number and the first letter of color. For Example, Alflex Large (AL) tag with number 100 that is white would have an ID of "AL100W".
- Rookeries. This sheet contains all Island – Rookery – Section combos. Islands have rookeries, which have sections.
- Codes. This sheet contains all existing codes: Visibility, Pup, and Loss. It contains the code type, code values, and description for each value.
- TagTypes. This sheet contains all existing tag types and their possible colors. Each tag type has a corresponding set of colors.

From this Data Management document, a desktop application generates four Xml documents: Seals.xml, Codes.xml, TagTypes.xml, and Islands.xml. These documents are then placed into the Phone's SD card where the Phone will have access to them.

Seals.xml is the parsed IDLookup sheet. It is formatted so that each Seal element contains its perspective tags as children.

```
<SEALXML>
    <Seal id="AL100W" lastSeen="1012609933" sex="F">
        <Tag side="left" color="White" number="100" type="AL"/>
        <Tag side="right" color="White"  number="100" type="AL"/>
    </Seal>
</SEALXML>
```

Codes.xml is the parsed Codes sheet. It is formatted so that each Code has values and descriptions.

```
<CODEXML>
    <Code name="Visibility">
        <Item description="Excellent" value="1"/>
        <Item description="Fair" value="2"/>
        <Item description="Poor" value="3"/>
    </Code>
</CODEXML>
```

TagTypes.xml is the parsed TagTypes sheet. It is formatted so that each TagType has a specified set of colors and a description of each type abbreviation.

```
<TAGTYPEXML>
    <TagType description="Narrow Allflex" name="AN">
        <Color name="Pink"/>
    </TagType>
</TAGTYPEXML>
```

Finally, Islands.xml is the parsed Rookeries sheet. It is formatted so that Islands have rookeries which have sections.

```
<ISLANDXML>
    <Island name="St. Paul">
        <Rookery name="Lukanin">
```

```
                <Section name="1"/>
                <Section name="2"/>
            </Rookery>
        </Island>
    </ISLANDXML>
```


2.2 Prior Data Entry Process

Observations are recorded on a printed double sided Microsoft Excel Sheet. Each row of the sheet represents a single *Observation*. On the back side of this sheet is a reference table for all codes, such as visibility or pup codes. These fields are described below.

- Location. The Rookery in which the observer is working in.
- Section. Rookeries are separated into sections.
- Time. The Time of the observation in hours and minutes.
- Seal Number. A reference number given to the each seal. This is used as an identifier of the seal for reference in future observations. Integer value starting at some multiple of 100 and goes up incrementally. For Example: Observer starts with 100 and following *Observations* are 101, 102, 103, etc… whereas another observer starts at 200, and following *Observations* are 201, 202, 203, etc….
- Detected/Read. Read: Tag was seen and number recorded. Detected: Tag was seen, but unable to record number.
- Distance. Distance to the seal in meters.
- Angle. Angle of observation relative to the seal.
- Optics. Type of optics used for observation.
- Visibility. Integer code for visibility at time of observation.
- Side. Side of the seal the tag was detected, recorded as **L**eft or **R**ight.
- Type or Loss. Tag type or integer loss code if tag not found. Tag types are specified by a one to two letter code.
- Tag Number. The number displayed on the observed tag. A mixture of characters and numbers.
- Tag Color. Color of observed tag. This is normally recorded before Tag Number (above).
- Confirm. Observer was able to visually confirm all previous tag information.
- Second flipper seen. **Y**es: Other flipper was seen, but not necessarily other tag. **N**o: Other flipper was not seen.
- Second Tag Seen. **Y**es: Second tag was seen but not necessarily recorded. **N**o: Second tag was not seen.
- Pup code. Integer code for female seals relation to pups.
- Comments. Additional information as necessary.

Counts are recorded on a spreadsheet. Each row in the sheet represents a single count entry. A *count* entry consists of a tally of all seals in a group. These fields are described below:

- Island. The island on which the count is occurring.
- Rookery. The rookery of the current island.
- Section. The section of the current rookery.
- Observer. The initials of the current observer.
- Date. The Date of the *count*.
- Start. The time at which the count began.

- End. The time at which the count ended.
- Counts Fields: Idle Bull, Territorial Bulls, Harem Bulls, Females, Pups and Dead Pups. These are non-negative values for the count.

At the end of each day, the observer manually transfer this handwritten data to the Microsoft Excel Sheet provided by Dr. Testa. *Counts* are transferred to "RawCounts" and *Observations* are transferred to "MasterResighting", as seen below.

	A	B	C	D	E	F	G	H	I	J	K
1	Record number	ID	Island	Rookery	Section	Obs.	Date	Time	Seal	Detected/ Read	Dista (met
2	1	ANP413P	ST. PAUL	POLOVINA CLIFFS	7B	JWT	7/1/2010	21:55	1	D	
3	2	ANP413P	ST. PAUL	POLOVINA CLIFFS	7B	JWT	7/1/2010	21:56	1	R	
4	3	ALX053W	ST. PAUL	POLOVINA CLIFFS	7C	JWT	7/1/2010	22:43	2	D	
5	4	ALX053W	ST. PAUL	POLOVINA CLIFFS	7C	JWT	7/1/2010	22:45	2	R	
6	5	AL70W	ST. PAUL	POLOVINA CLIFFS	7B	JWT	7/1/2010	23:31	3	D	
7	6	AL70W	ST. PAUL	POLOVINA CLIFFS	7B	JWT	7/1/2010	23:33	3	R	
8	7	ALX053W	ST. PAUL	POLOVINA CLIFFS	7B	LFT	7/2/2010	20:23	1	D	
9	8	ALX053W	ST. PAUL	POLOVINA CLIFFS	7B	LFT	7/2/2010	20:23	1	R	
10	9	ANP413P	ST. PAUL	POLOVINA CLIFFS	7B	LFT	7/2/2010	20:49	2	D	
11	10	ANP413P	ST. PAUL	POLOVINA CLIFFS	7B	LFT	7/2/2010	21:02	2	D	
12	11	ANP413P	ST. PAUL	POLOVINA CLIFFS	7B	LFT	7/2/2010	21:06	2	R	
13	12		ST. PAUL	POLOVINA CLIFFS	7B	LFT	7/2/2010	21:03	3	D	
14	13	AL70W	ST. PAUL	POLOVINA CLIFFS	7B	LFT	7/2/2010	21:25	4	D	
15	14	AL70W	ST. PAUL	POLOVINA CLIFFS	7B	LFT	7/2/2010	21:38	4	D	

Dr. Testa then uses this data to perform his statistical analysis of the fur seal population.

3. Project Requirements

The requirements set forth by Dr. Testa were very clear from the beginning. Dr. Testa wanted to replace the current process with a digital version that, as much as possible, would retain all the current functionality while adding several convenient features. Initially, I was given the requirements in English by Dr. Testa. Although the initial requirements captured the general idea for desired functionality, they did not provide many of the necessary implementation details. This resulted in a prototyping design methodology which allowed for the further clarification of requirements.

3.1 Functional Specifications

The given specifications can be separated into the three major categories: General, Observation Entry, and Count Entry. These are described below.

3.1.1 General Functional Specifications

1. The system must export *Counts, Observations,* and *Count* totals separate CSVs where they may be accessed by the computer.

3.1.2 Observation Form Functional Specifications

1. Use entered tag numbers to validate tag side, seal sex and second tag information. This data is based on the "IDLookup" sheet which contains all tagged seals.

2. Application must have capability to insert new *Observation* or modify previous *Observation*.

3. Form must Validate Tag – Color combo. Tags may only be certain colors.

4. Form must Validate Island – Rookery – Section combo. Islands have specific rookeries which have specific Sections.

5. All Codes, Tag Types, Colors, Islands, Rookeries, and Sections must be managed from a central location.

6. Application must have a visual lookup table, displaying all previous *Observations* recorded that day.

7. Application must inform user of errors or inconsistencies in form upon submission, but allow observer to override these errors. In either case, error is recorded in comments.

8. Application must have a visual error validation form that is displayed when inconsistencies in form and database data occur.

3.1.3 Count Form Functional Specifications

1. Must have the following count categories: Lone Bulls, Territorial Bulls, Harem Bulls, Female Bulls, Pups, Dead Bulls, Dead Females, and Dead Pups.

2. Must allow negative counts for any category.

3. Harem Bulls must have females.

4. There can be no more than one Harem Bull per count entry.

5. For each count category, must be able to either manually enter a value, or increment/decrement it.

6. Application must have a visual lookup table, displaying all previous *Observations* recorded that day.

3.2 System Specifications

The application must be designed for field use on the Motorola Defy, running Android 2.1 Update 1 OS. The device must have a memory card of at least 1 GB.

4. System Design

Android applications are predominantly programmed with Java. As a result, I utilized Object Oriented Design patterns. Primarily, a Model – View – Controller (MVC) design pattern, which is strongly encouraged by the Android Architecture. However, Singleton, Adapter, and others are also used.

4.1 Android Architecture

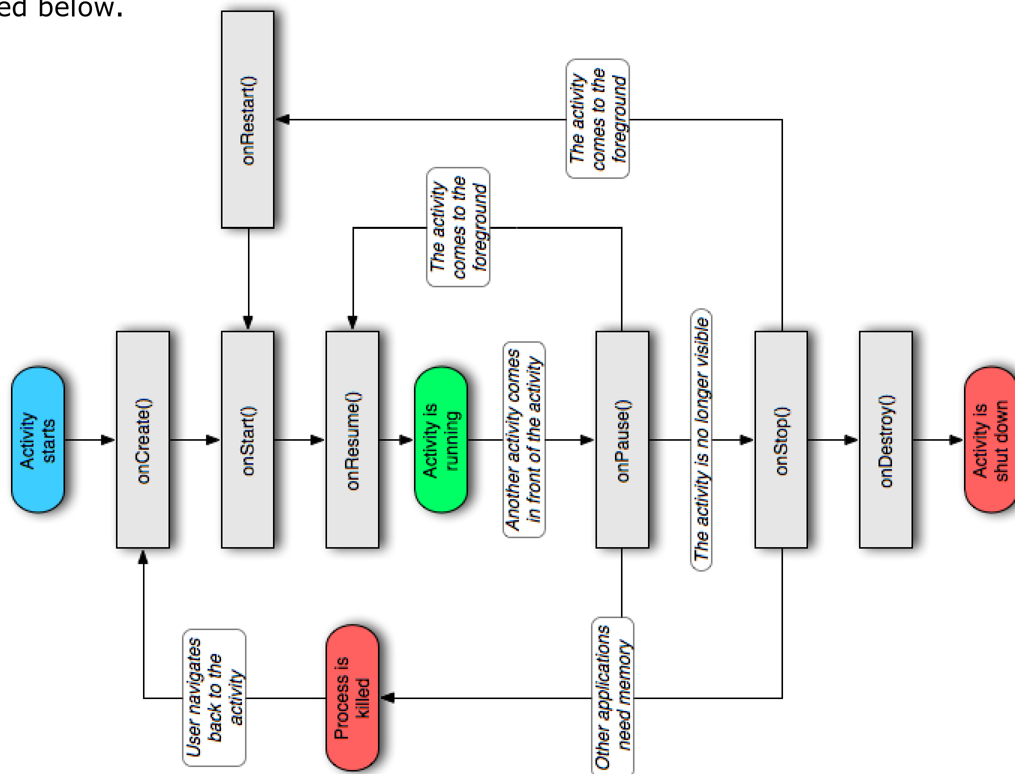
Because system design is based on the Android Architecture, it will be briefly covered here so that in the next sections certain aspects will be clear. The primary concept in the Android Architecture that I will be explaining is the Android Activity.

According to Android documentation,

> "An `Activity` is an application component that provides a screen with which users can interact in order to do something…"

Android views each activity as a separate application. For each individual application, or context, an activity stack is maintained. Each activity is placed at the top of the stack, and when that activity has finished, it is popped from the stack and the one below it resumes its operations. This feature is built into the architecture.

Furthermore, Activities have a specified lifecycle. This is best described by the image displayed below.



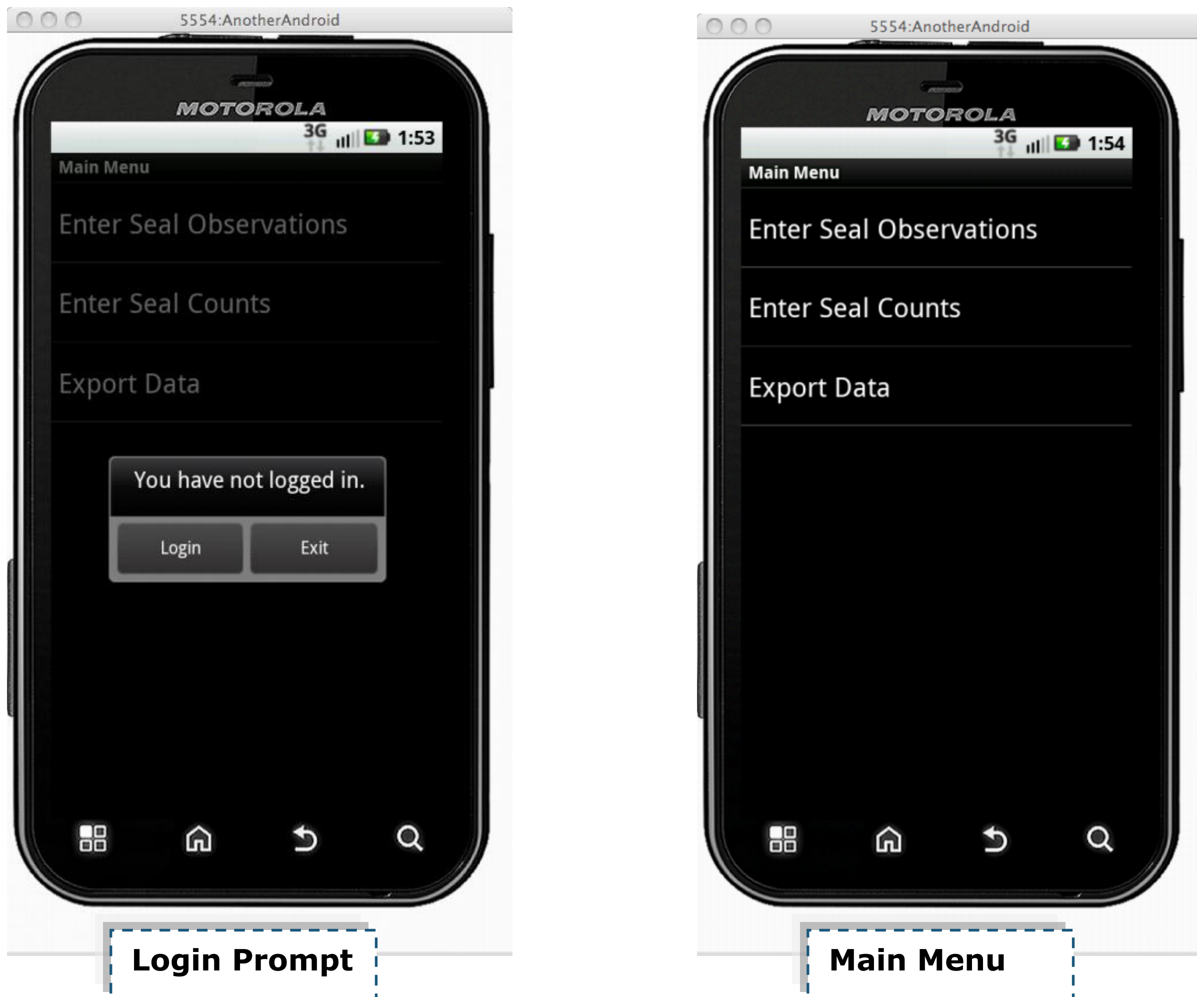
Although an Activity's view can be customized in the code, Android predominantly utilizes Xml to populate views, store primitive static data that is shared among views or activities.

Therefore, this projects form layouts were done in Xml, using Android built in features to populate the view from these Xml files.

4.2 User Interface Design

Understanding the Activity Stack will help clarify my choices for User Interface design.

The starting view is a loading or "splash" screen. This initial view is the initialize for all internal data and displays a waiting notification until its operations are complete. Because of the Android Architecture, this is the first activity on the stack. Once all necessary data is uploaded, the Main Menu is displayed.



However, as part of the requirements, each observer must first login and select an island and rookery. This is given as a prompt at the start of the Main Menu Activity. Which, when selected, starts the Login Activity.

Login information is used throughout the application, and therefore requires entry before application will continue. Should the user select "cancel" or "back" he will be prompted again to login or exit.

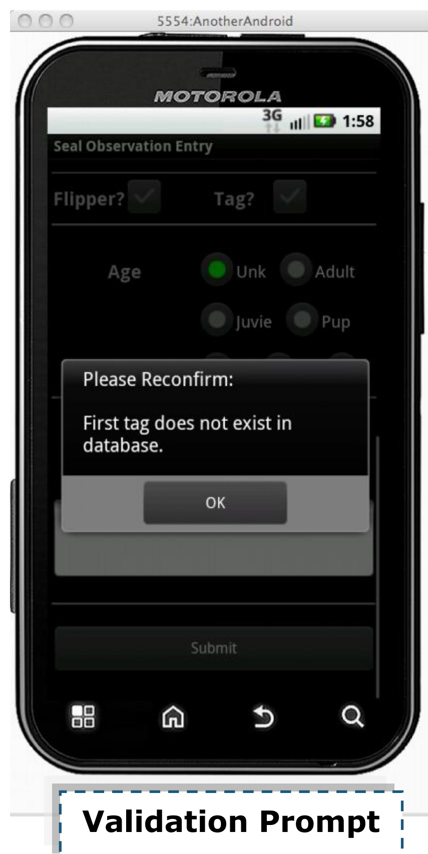
The user may return to this Activity through the "Options" menu provided by Android. In this way, the observer may adjust this data at any time.

The Main Menu is the base for all navigation. As seen above, from here the user may choose one of several options. Selecting either Count Entry or Observation Entry will start a new Android Activity, CSV Export will allow the user to export the .csv files regarding a selected date.

The Observation Entry Form is the most complex form in the application. Because space is limited in a handheld device, many fields are conditionally visible.

Observation Form

Furthermore, this form's data is checked for errors and consistency with database. In the general cases, a simple confirmation dialog is displayed.



Validation Prompt

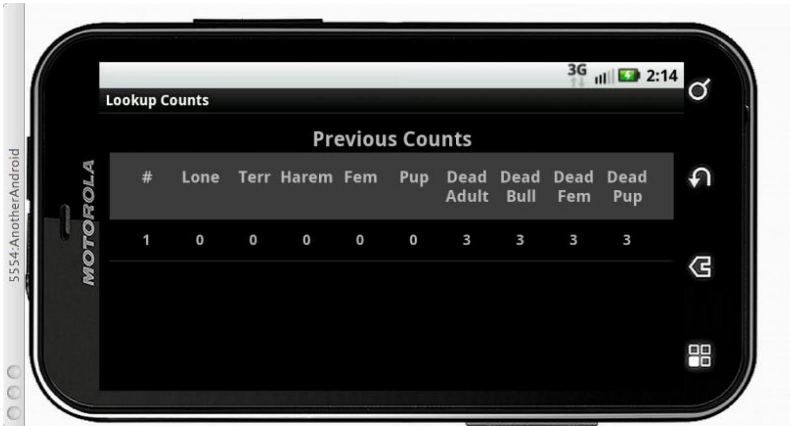
However, for many scenarios, a more specific form is used to require validation. This is the Observation Validation Form and is started as a separate Android Activity on the stack. Passed to this form are the necessary fields to validate and returned from it is the result of such validations: Changed, Unchanged, or Canceled. This form is not directly accessible, but rather is a result of inconsistencies between entered data and database data.

Observation Validation

Count Form

The Count Entry Form also needed to present itself clearly in the limited screen space. Hence, fields within the Count Entry Form are also conditionally visible.

Finally, both Count and Observation Entry Forms share the ability to start a final Activity, The Lookup Activity. This activity, unlike the Validation Observation Activity, is directly accessible from either Count or Observation Entry Forms. Its displayed values are dependent on the starting form, but in either case, it will display a list of all successfully submitted data for that particular form.



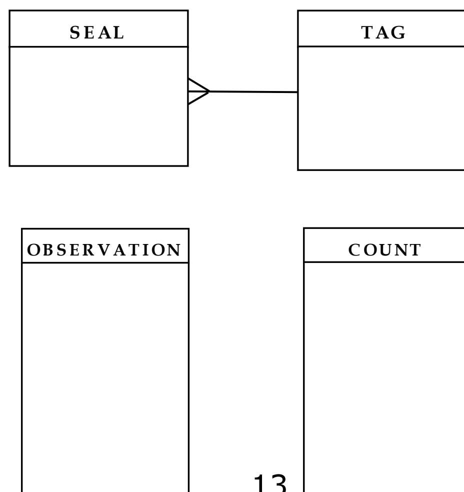
Lookup Form

4.3 Data Structures

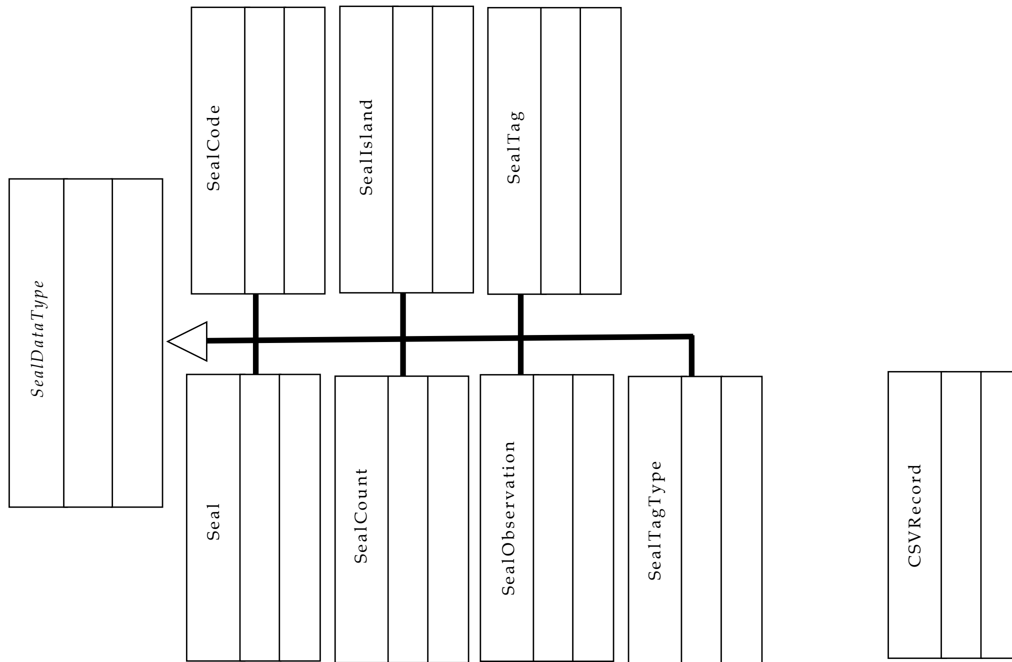
Two forms of data are stored in the application, Xml and SQLite database. Xml is used to store any data used in the UI, whereas SQLite is used to store the heavier sets of data, such as IDLookup table, Observation entries and Count Entries.

Storing dynamic data in Xml format removed any restrictions for the use of a relational database. This allowed for more rapid development, as database design and setup is time consuming. Furthermore, the data was already being put into Xml format for transfer to the phone from the Microsoft Excel sheet, so storing this data in internal memory was merely a matter of copying it directly. Hence, of the data files passed to the phone, only Seals.xml is written into the phone's database (For Xml Format refer to Section 2.1).

The database ER Diagram is displayed below.



To simplify database interactions with the application, an Object Oriented class structure was created to somewhat parallel the database. Below is the UML of this diagram.

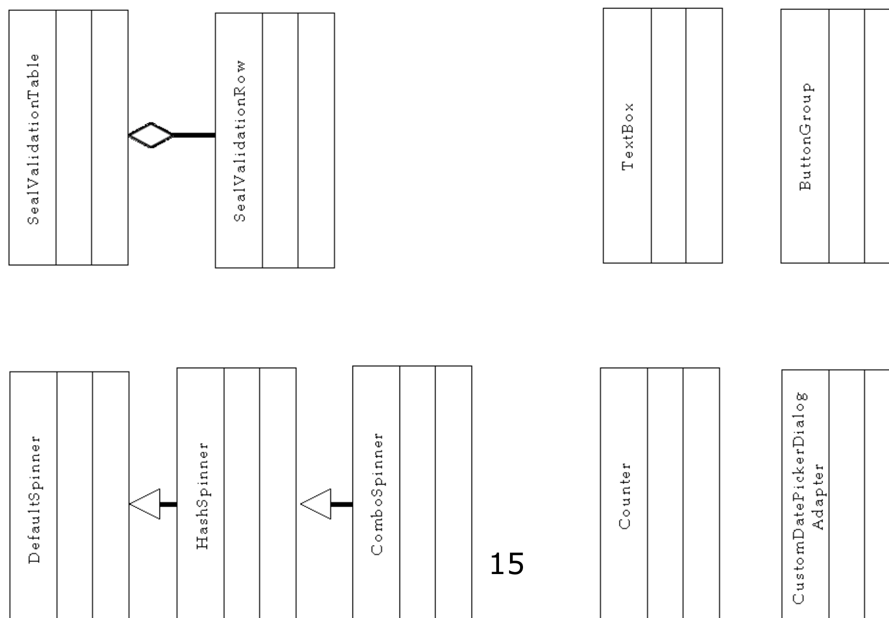


It can be seen, the database and UML are not exact matches. The Object Oriented Design, the seal holds references to its tags, whereas in the database, the tags hold reference to the Seal. This is largely due to lookup behavior. An observer enters a Tag Number, which in turn means that the seal must exist. However, to find that seal, the tag number is used as the lookup variable. Once the number is found in the database, the seal can be identified. However, from an Object Oriented approach, if the tag exists, the Seal must exist first. Hence, a seal is created, and then given tags, not the other way around.

Several smaller data structures were also created to manage visual elements. In the simple case, a drop down list, or Spinner, merely returns its displayed value. However, this proved ineffective for code display. So, a custom Spinner was created that allowed use of a Hash Table for storage of data. This data structure allowed for Spinner display of code and its corresponding description, while returning one or the other. This also applied for setting values. Although I use the term Hash Table, in reality it is two parallel arrays.

This proved effective for most cases, with the exception of Tag Types. In the original printed form, either a tag type was entered or a loss code. This value was entered in the same field. In an effort to capture this behavior, a Combo Spinner was created. This added the functionality of specifying two separate Hash Tables that were concatenated together, one atop the other. Now I was able to query what value was selected, whether it was in the top list or bottom and modify the form as necessary.

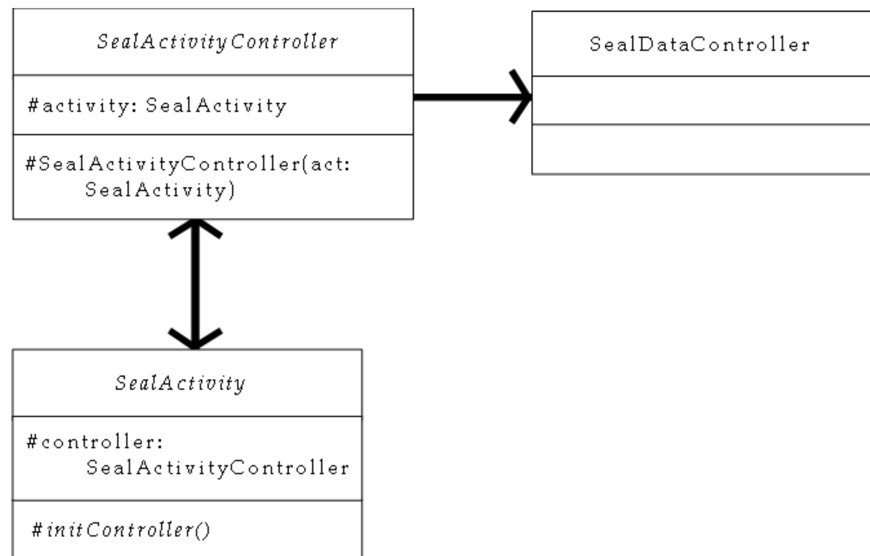
Many other custom visual elements were created to manage specific scenarios. As seen below.



4.4 System Architecture

At a Top Down look at the project, it is primarily designed using MVC, utilizing a parallel class hierarchy to fully utilize an Object Oriented design. However, a closer look reviews a singleton for data access, adapters for Xml Parsing, database creation and access, and Csv creation.

The MVC Pattern is best described by the below structure.



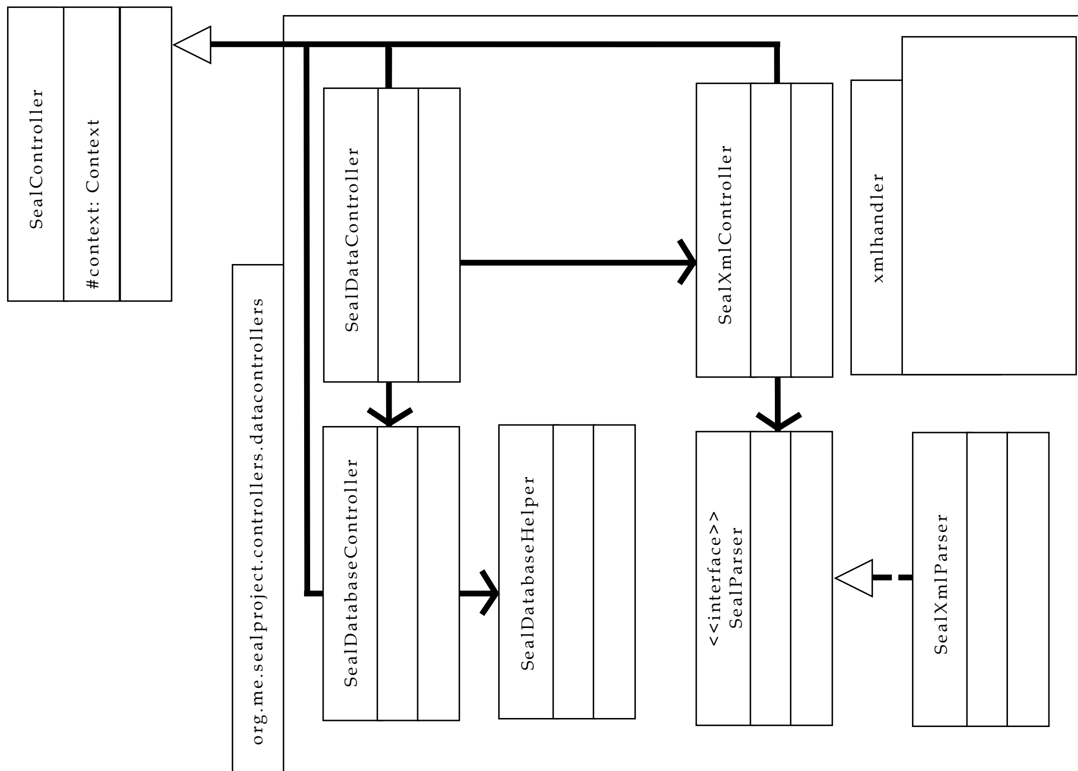
Built as a parallel class hierarchy, activities and their corresponding controllers are built. This allowed for convenient encapsulation of the code. Activities are responsible for handling view setup and changes, which includes returning contained data. Controllers are responsible for managing the activity's data, which includes database insertion and or giving the appropriate data to the view.

Because of the Android Architecture, the activity is initialized before the controller. This created a Slave-Master relationship between the activity and controller. When the activity is started, it calls "initController()" to start the controller, with a reference to self. The slave wakes up the master.

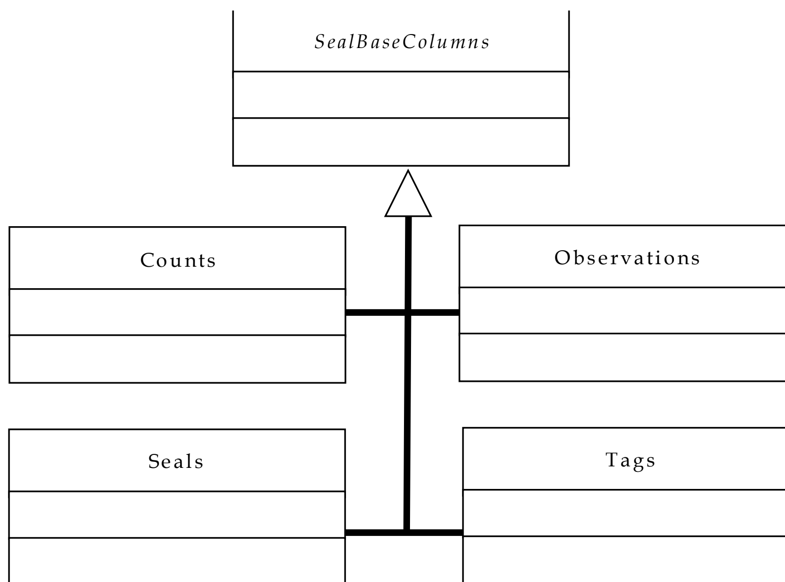
At the base of this architecture, all activity lifecycle events are passed to the base controller class. This allows for the controller to behave as another activity, if desired, reacting to the same events as the activity without the parallel activity being aware of or needing to pass these events forward.

On Activity creation, the view is populated with data from an Xml file. On an activities start, the activity retains a reference to each of the needed Xml elements, which are accessible via getters and setters. The Controller utilizes these references to obtain or set data for the view as necessary.

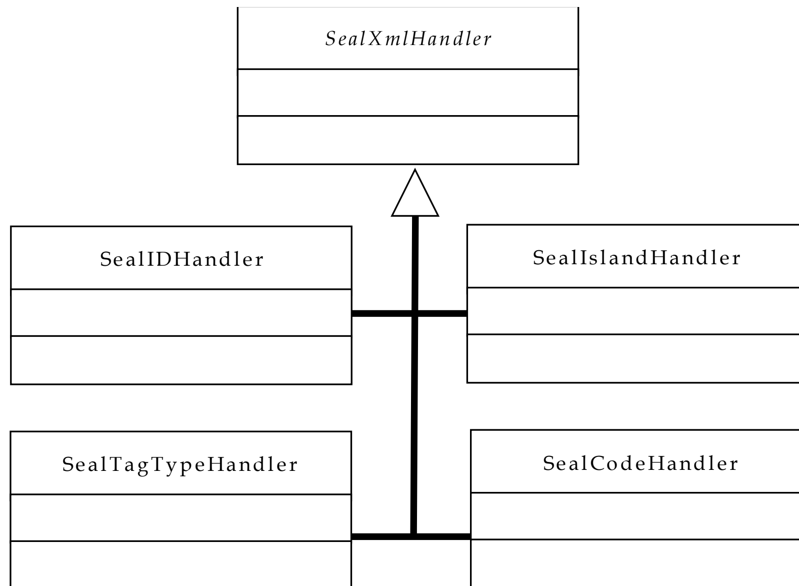
The Data Controller is a singleton that manages the Database and Xml Controllers. Because this is referenced at the base activity, all Activity Controllers have access to the Data Controller.



The Database Controller is an adapter for the SQLite database. It utilizes several convenience classes to create, insert and update the phones internal SQLite database.



The Xml Controller handles all custom Xml Parsing. Upon startup, the initial activity or "StartupActivityController", checks for and parses any extrenal Xml files found on the Phone's SD card, returning this data in an array for further use. Since each Xml file is in a different format, an adapter pattern is used.



Each Xml file type has a corresponding "handler" type that can be used to parsed it's contents appropriately. The Xml Controller provides convenience methods to access this data, so that the choice of handler is transparent to the Data Controller, when used.

4.5 Algorithms

Most algorithms in the project were fairly straightforward. The most difficult algorithms proved to be proper update of the form upon upload of previous observations and correct comparison of form data with database.

Visual updates to the form proved difficult because of dependencies within the form. When a user selects a tag type, the colors are populated with that particular tag types color selections. This update was performed by an event. However, when attempting to populate the entire form with data, this event driven method did not properly update the color, instead leaving it either selected, but not display the selection or not selected at all. The problem arose in that the event driven architecture was acting before the view or after the view had changed. This resulted in inconsistent results for display. To correct this problem, the event responder on the element is replaced with a temporary responder. This temporary responder, which is called at some arbitrary time during the change process, sets the responder to what it was originally. Under this setup, the visual updates proceeded correctly, in spite of delays.

Validation of the Observation Activity proved to be the most difficult algorithm. The Observation Form, upon submission, is compared with database data. As described in section 4.2 (User Interface), this results in either a simple dialog box or a validation activity. The issue resides in the fact that the entered data is allowed to disagree with the database. For Example, the user may enter two tags, with one differing from database, and still

submit it as an observation. This scenario means the validation routines must be conditional on the need to validate the form.

To appropriately handle this scenario, the Observation Controller retains state information in the form of two Booleans. The first, formHasChanged, is based on current form data. If any data in the form changes, it is revalidated. The second, dataValidated, is whether the data has been processed by the form and the user chose to validate. This allowed for controlling of the controllers state.

5. Software Development Process

Because the UI was not well defined and the large learning curve for Android, I used the prototyping development methodology. I developed a prototype for Dr. Testa every two weeks, which was reviewed, changes suggested, new requirements added, which I then implemented and repeated the process.

5.1 Testing and Debugging

Many of the bugs were merely lack of knowledge in working with Android. These were generally worked out through better understanding of the architecture and how to utilize it correctly. Some of the most difficult bugs involved storage of information in the database and visual display of tags.

One of the major bugs in the project was visual update when uploading data to the program. This is largely described in section 4.5 (Algorithms), under the visual updates. The program arose in that setting the Observation Activity to display a new Observation would at times display both tags correctly, and at others only display one while either hiding the other or display only its tag type, but not color. This bug turned out to be a problem with the event driven architecture. Events were being queued and run at the end with old data, this meant that the event called to update the color display was being given the wrong data, thereby hiding it.

Another rather troublesome bug had to do with display of the Count Forms visual elements. The Counters, which are custom elements, would take up too much space in the screen, pushing the other Counters out of view. I was unable to determine why the automatic layout feature was acting in such a way and ended up forcing the element size manually, storing these values in an Xml file for future editing. As a result, were this application to be transferred to another device with a different screen size, I am not sure the display would correctly fit the screen.

The project proved to be much larger than anticipated. Being primarily visual, most testing was done by creating shortcuts to a specific screen, in which I would check for all scenarios pertaining to my feature. Furthermore, whenever I met with Dr. Testa, he would walk through the application and inevitably, find some error or bug that needed to be fixed. Although I found this frustrating, it proved to be a very profitable setup as I was able to work through these bugs before the next meeting and repeat.

Testing that was not related to the UI was done by using Boolean variables to either enable or disable test settings, these varied from printing files to the desktop to deleting the database and setting it up once again. Over time, this simplified to an interface that held a

Boolean test variable, when set the project runs through any setup test routines and/or print statements.

5.2 Prototyping Process

Prototyping proved extremely useful in the project design. Because of my lack of experience with Android, the first three prototypes, although changing marginally as far as visual elements were concerned, were architecturally unique. As I learned more of Android, I found a design pattern that best fit the given built in Architecture.

Initially, my first prototype was built around the idea of a console application: A single controller that held the view stack and popped or pushed these views as necessary. I quickly found this to be unrealistic, as it would require quite a bit of work to get around Androids architecture of separate Activities.

By the second prototype, my design better resembled what the final design would be. However, it was still not complete. As Dr. Testa and I fine tuned the requirements and visual design, my code base grew rapidly. This required an architecture that better encapsulated the differing functions of each activity. Furthermore, Dr. Testa added the requirement that there be a lookup form for previous activities, and a way to effectively validate the Observation data. I could see that I needed a structure more conducive for adding elements. From this was born the idea of a parallel class hierarchy of Activities and their Controllers.

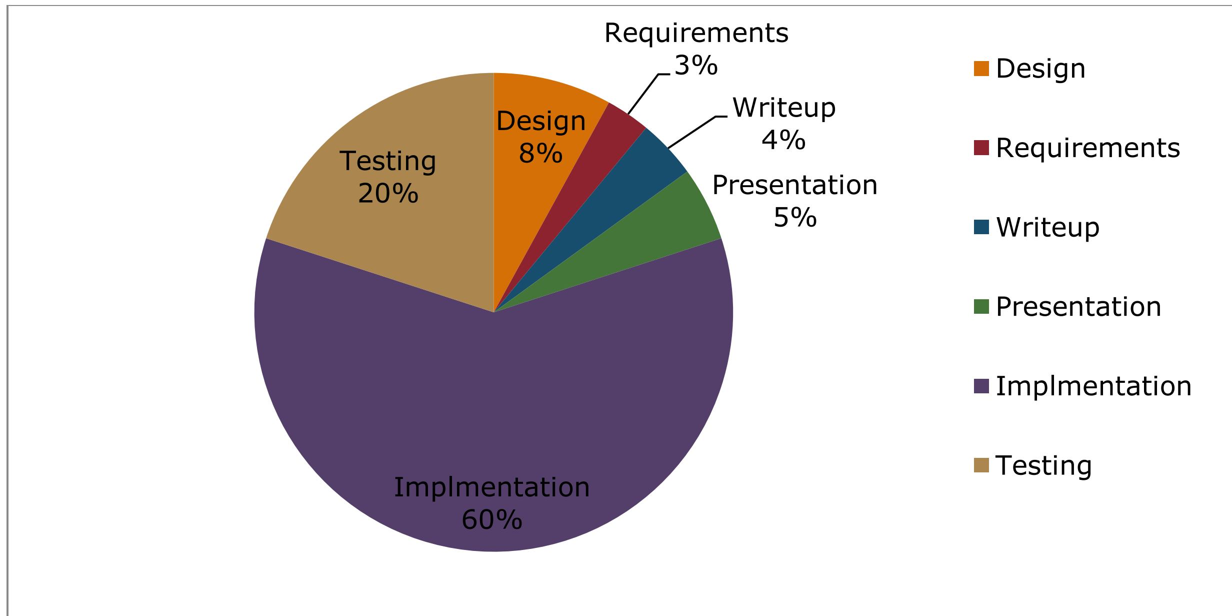
The next three prototypes were centered on visual display. I became comfortable discussion different options with Dr. Testa and addressing what could and could not be realistically accomplished. Furthermore, I gained a better understanding of different elements in the project. Such as how tags were recorded.

At several points in the project, I felt frustrated with the number of features requested by the client. However, as the project continued, I became comfortable with presenting options of what could be accomplished within the given time and what could not. I began to present a list of priorities, what he would like most to be accomplished immediately.

One of the greatest disadvantages to this project was the lack of accessibility to the phone. Dr. Testa only possessed one Phone for the project, and with the large amount of features and learning curve, testing of the project was limited. It would have been highly valuable to have left a running application with Dr. Testa that he would have walked through while I worked, further rooting out any bugs that were present. As a result, there are still bugs that I will continue to work out after I graduate.

5.3 Work Breakdown

Initially, the project was planned to take around 110 hours over the course of the semester. However, by the end of the semester, I had spent over 220 hours on the project, this rounds out to about 20 – 22 hours be week. As seen below is the work break down in time spent in each segment.



6. Results

The project was much larger than anticipated and as a result, is not yet complete. All major features have been implemented and Dr. Testa is looking at first use this summer. However, this is somewhat dependant on budget limitations of NMML for purchasing multiple phones. Only one requirement has not been met to specifications, ability to insert a new seal record under the same seal number.

6.1 Future Steps

Fixing bugs in the application is the most immediate need. As a result of having only one device, the two weeks of the project have been the only opportunity for Dr. Testa to work with the phone. Hence, for a period of time the work done on this application will be to debug what errors are found during his testing.

As mentioned above, the last feature still needs to be implemented. However, along with that, I would like to implement a progressive load view to be used when the database is first initialized and/or repopulated with seals and tags. Also, I would like to allow modification of previous counts, rather than just a lookup table. There has been mention of future work with the application as necessity arises, but no specific requests for features have been given yet. I would be interested in continued work with Dr. Testa should he be willing to hire me for my services over the summer.

7. Summary and Conclusions

The Fur Seal Project was developed for a handheld Android Device, running Android 2.1 Update 1 operating system with the goal of replacing the paper and pen method of entering observations and counts of Fur Seals. The Application was not completely finish and bugs are still be discovered and worked through, but it is anticipated to be done very soon. Dr.

Testa is very satisfied with the user interface and has brought forward the option of paid work in the future to add features or make modifications.

Overall I learned an amazing amount working on this project. Having never work with a client before, I had to become comfortable with setting appropriate limits and communicating clearly any needed information. I was new to Android and its included features, such as SQLite, and now feel I have broken the ice on Android Apps and look forward to future developments on the platform.

I found the project requirements vague, but the bi-weekly meetings proved and effective method for developing a UI that the client was pleased. This regular involvement proved to be a great advantage in the project lifecycle and I found it very beneficial.

If I were to do this project again, I would focus much more so on understanding the Android Architecture before setting up large code bases. I found that much of my time was spent trying to do something too quickly, and therefore incorrectly, instead of learning the architecture to design solid code that worked consistently.

8. References

[1] Darcy, L. & Conder, S. (2011). *Android wireless application development*. Boston, MA: Addison-Wesley

[2] Testa, J. W. (editor). 2005. *Fur seal investigations, 2002-2003*. U.S. Dep. Commer., NOAA Tech. Memo. NMFS-AFSC-151, 72p

[3] *Activities* (2011) retrieved from http://developer.android.com/guide/topics/fundamentals/activities.html