Optical Character Recognition Accounting Application
Technical Guide

Author: Daniel Maguire
Student Number: 12528133

Supervisor: Dr Alistair Sutherland

Date of Completion: 21/05/2017

Abstract

The purpose of this project is to develop an application that will allow the user to capture and store receipt data in an online database, while providing an easy-to-use platform to access this data. The application provides the user with insight into their purchase habits and will allow the user to view their purchase history through various representations. The aim is to provide the end user with a tool that will allow them to educate themselves in their own spending affairs as well as providing some insight in the form of statistical representations of their receipt data. The application takes advantage of tesseracts optical character recognition technology, while applying error correcting algorithms for smoother input to capture obscured images. This application is designed for android mobile devices. It will run on Android devices running Android 5.0 or higher, and works in conjunction with Googles Tesseract API as well as the OpenCV API, to provide a simple and accurate experience for the user while capturing receipt data. The user can then save their captured receipt securely to their unique user account hosted by Amazon Web Services. The application's secondary feature is to allow the user to view statistical data on their spending habits through the use of various charts provided by the MPAndroidChart library.

Table of Contents

Abstract **2**

1. **Introduction** **4**
   - *1.1. Description* *4*
   - *1.2. Motivation* *4*

2. **Research** **5**
   - *2.1. Existing and similar applications* *5*
   - *2.2. Image Processing and OCR Tools* *5*
   - *2.3. Database and Cloud Storage* *5*
   - *2.4. User accounts* *5*

3. **Design** **6**
   - *3.1. User Authentication* *6*
   - *3.2. Main Functionality of the application* *6*

4. **Implementation** **8**
   - *4.1. Image Processing* *8*
   - *4.2. OCR and Text Extraction* *8*
   - *4.3. Training Tesseract* *9*
   - *4.4. Database and Cloud* *9*
   - *4.5. Statistics* *9*

5. **Validation** **10**
   - *5.1. Code Scanning* *10*
   - *5.2. User Testing* *10*
   - *5.3. Unit Testing* *11*

6. **Sample Code** **12**

7. **Problems and Resolutions** **14**
   - *7.1. Low accuracy while capturing receipts* *14*
   - *7.2. External Light Sources* *15*
   - *7.3. Tesseract and Creased Receipts* *15*

8. **Results** **15**

9. **Future Work** **16**
   - *9.1. Camera2 API* *16*
   - *9.2. Exporting .CSV Files* *16*
   - *9.3. Capturing more data from the receipt* *16*
   - *9.4. Capturing purchase information from email* *16*

1. Introduction

1.1. Description

This product has been developed as a personal accounting application. The user can use this application to store receipt data in a personal database accessed through the application. It allows the user to access this information in a monthly calendar view. The application provides the user with insight into their purchase habits and will allow the user to view their purchase history through many different representations. The application allows the user to manage and understand their spending habits in the hope of bringing them more control over these habits. The aim is to provide the end user with a tool that will allow them to educate themselves in their own spending affairs, as well as providing some insight in the form of statistical representations of their receipt data. The application takes advantage of tesseracts optical character recognition technology, while applying error-correcting algorithms for smoother input to capture obscured images.

This application is designed for android mobile devices. The application will run on Android devices running Android 5.0 or higher, and works in conjunction with Googles Tesseract API as well as the OpenCV API, to provide a simple and accurate experience for the user, while capturing receipt data. The user can then save their captured receipt securely to their unique user account hosted by Amazon Web Services. The application's secondary feature is to allow the user to view statistical data on their spending habits through the use of various charts provided by the MPAndroidChart API.

1.2. Motivation

I choose to create this application as when I searched for an application that allowed the user to quickly and easily store receipt/invoice data, there was none readily available. I found similar applications that allowed the user to enter their daily expenses. This turned out to be a slow manual task, with no option to store the receipt itself. This left room for error if not entered correctly. I was unable to find a user-friendly application that facilitated the storage of the receipt image, as well as the expense data, possibly due to the difficulty of working with receipts and OCR. I have also found a huge interest among working professionals during an initial questionnaire that I created to observe the level of interest in this application, as many people have stated that calculating their expenses at the end of each month can be a time-consuming task. From the evidence collected, it appeared that there was a gap in the market for an application of this nature.

2. Research

2.1. Existing and similar applications

I have researched applications of a similar service or functionality. The two applications that come the closest are Spendee and TextFairy. Although these applications are very different to each other, I have taken inspiration from both to create this application. Spendee is an Android application that allows the user to manually input how much they spend throughout the day, while keeping track of it. I observed the layout of the application, which provided me with ideas for developing this application. TextFairy is an image to text document converter for Android. I observed how TextFairy processes images to get good quality results, which aided me while developing a robust image processing algorithm.

2.2. Image Processing and OCR Tools

After researching image processing libraries, OpenCV seemed to cover all of the areas required. I choose this as it is also very well documented.
Leptonica - another image processing library, was also a candidate but it was not as clearly documented and I became familiar with OpenCV more easily.
To perform optical character recognition, I quickly became aware that Tesseract, Googles OCR API is considered the industry standard, and therefore opted to use this.

2.3. Database and Cloud Storage

I needed a way of storing file metadata in a database, as well as the receipt image itself. Throughout my research, I have found various ways of achieving this. My first choice was MongoDB, as it allows storing images within the database. After conducting some more research, I decided to store the file metadata in Amazon's noSQL database, DynamoDB while storing the file in Amazon's S3 cloud storage. I then referenced the location of the file in the database. I decided on this approach as file storage is cheaper than database storage, so this combination appeared to be a good choice. Another deciding factor was that AWS services are always available and allow for an ease of scalability, while also minimising database administration.

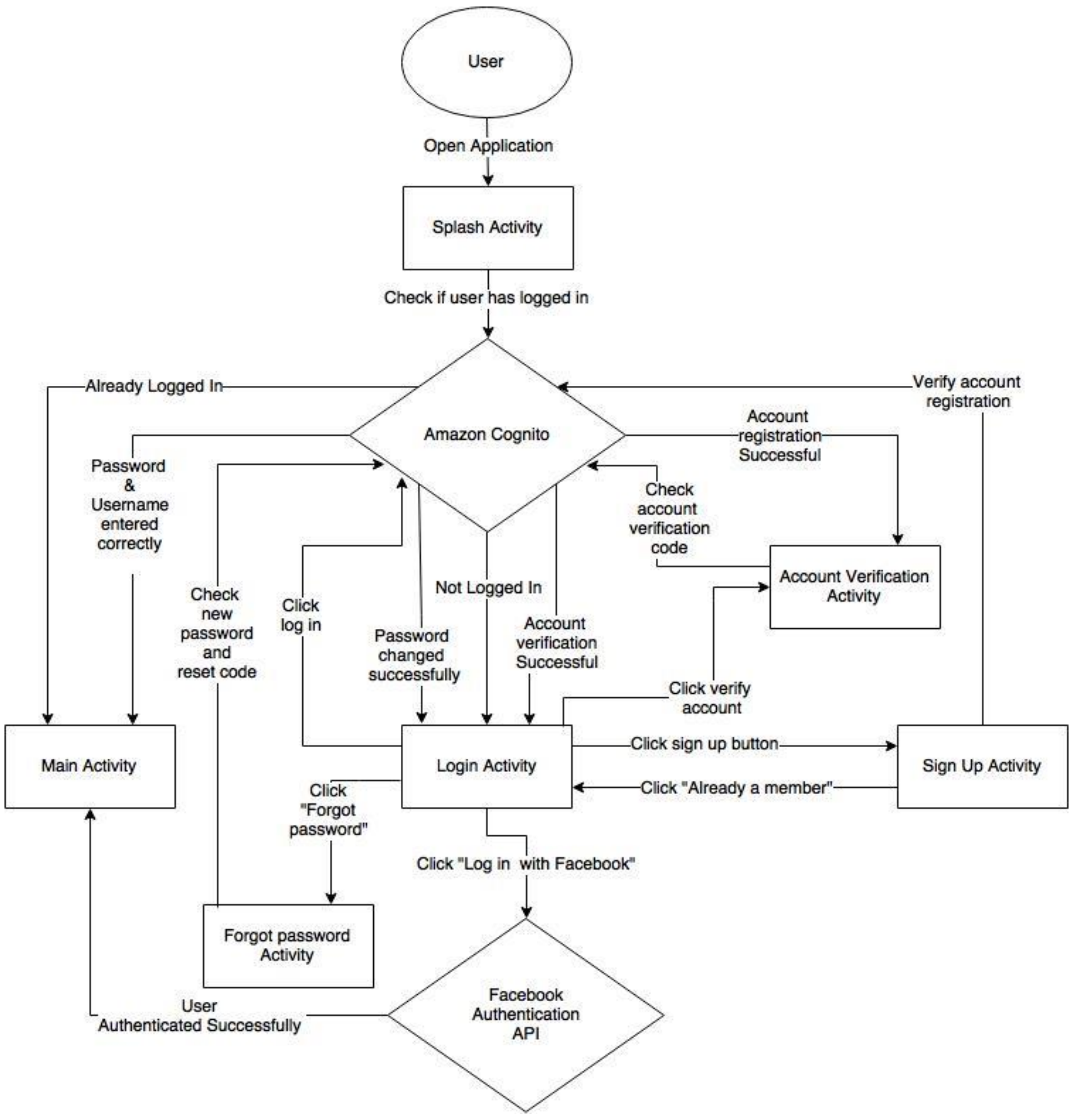
2.4. User accounts

To allow user authentication, I decided to use AWS Cognito as it provides a high level of security and confidentiality and integrates into the Android platform well. It also allows for easy administration of user accounts.

3. Design

3.1. User Authentication

To use the application, the user must have a user account. To achieve this, the user can create an account manually or log in using their Facebook credentials. To sign up through Facebook, the user can simply click "Log in with Facebook" and will be redirected to the Facebook application to authenticate. To manually sign up, the user must create an account and verify their account by entering a sign-up verification code sent to their email address. Once verified, the user can use their email address and password to log in to the application.


User
Open Application
Splash Activity
Check if user has logged in
Already Logged In
Amazon Cognito
Verify account registration
Account registration Successful
Password & Username entered correctly
Check new password and reset code
Click log in
Check account verification code
Account Verification Activity
Not Logged In
Password changed successfully
Account verification Successful
Click verify account
Main Activity
Login Activity
Click sign up button
Click "Already a member"
Sign Up Activity
Click "Forgot password"
Click "Log in with Facebook"
Forgot password Activity
Facebook Authentication API
User Authenticated Successfully

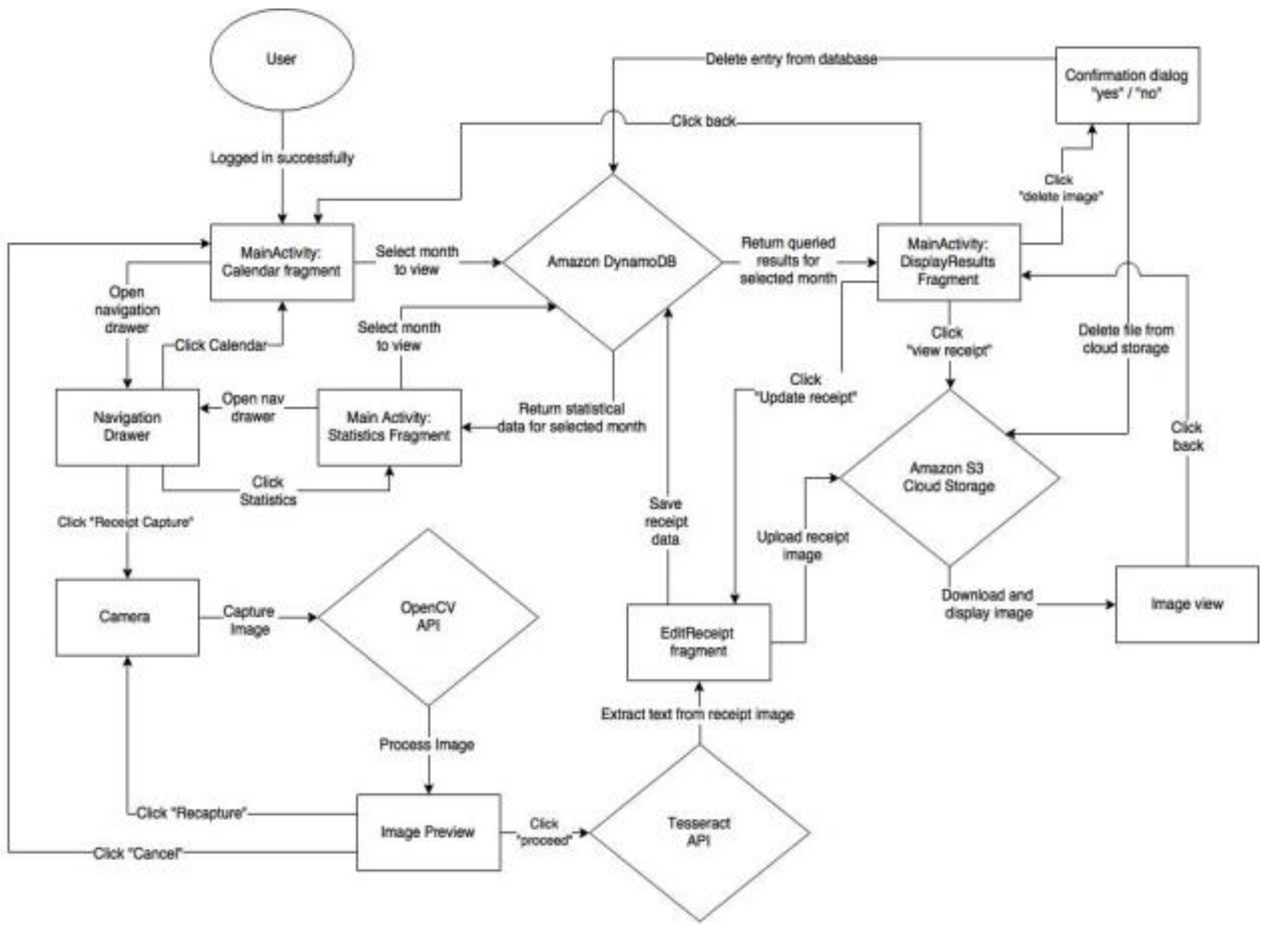
3.2. Main Functionality of the application

The main functionality of the application includes capturing and storing receipt images and meta-data, storing the image file privately online using Amazons S3

cloud storage, while storing the receipt meta-data in Amazon’s DynamoDB database. The meta data includes:
User ID – The users unique ID.
Receipt Name – This functions as the sort key used in the database.
Friendly name – The name of the receipt displayed and editable to the user.
Category – The expense category that the receipt falls into e.g. Food or Travel.
Date – The date visible to the user in which the receipt purchase was made.
File path – This is the path to the receipt image in the users’ cloud storage.
Formatted Date – Date equal to the “Date” entry but in the format yyyyMMdd to be used by the application.
Total – The total value of the receipt.

The receipts can then be accessed through the applications Calendar view by clicking the month and year in which you want to view. The application will then query the database and return all receipts for the corresponding month and year. The user can then view, update or delete the receipt from this view.
The user also has the option to view statistics on their saved receipts by accessing the statistics view and inputting the month and year they want to view.

The following diagram acts as a visual representation of the design of the main functionality of the application. The representation below also assumes that the user has logged in to the application previously.



4. Implementation

4.1. Image Processing

To increase the level of accuracy when performing OCR, I needed to process the image in various ways. All of the processing tools used have been from the OpenCV library. The first stage of the processing pipeline occurs before the user captures the image. The camera captures each frame, which it then passes to the drawBoundingRectangle method in the ImageProcessing class, which searches for the largest contour. This is the outline of the largest shape in the image, which should be the outline of the receipt if detected properly. It then draws a bounding box around the receipt on the frame and passes the frame to the camera preview window.

When the user clicks the capture button, only the image within the bounding box is captured by locating the corners of the bounding box and cropping the image based on these coordinates. The cropped matrix representation of the image is then passed to the cleanImage method. Here it undergoes denoising, resizing, binary thresholding, rotation, artefact removal, and once complete, writing to external storage. The user will then be shown a preview of the cleaned image which, if satisfied with the result, can proceed to optical character recognition.

4.2. OCR and Text Extraction

Optical character recognition is performed using TessTwo, Tesseract's Android library. The OCR task runs on a background thread when image processing task has completed, with the file path of the processed image passed to the OCR task. The OCRImage method is called and the trained tesseract file is checked to make sure that it is present in the tessdata folder and if not, it is added. The startOCR function is then called, encoding the file into a bitmap image. The orientation is then checked to make sure that the image is correctly rotated. The bitmap is then passed to extractText where the tesseract library is initialised and the text extraction process executes. The extracted text is then saved in a .txt file in the applications data folder.

While still executing on the background thread, the text extraction method is called. The text extraction's main functions are extracting the title, date, category and total from the text file containing the captured receipt text. While extracting the title, date and total, the most likely candidate is chosen from a list of potentials. The category is chosen by looking through the text for key words exclusive to a receipt of a specific category, for example, if the word ‘shoes’ is present on the receipt then the receipt is most likely a clothing receipt.

4.3. Training Tesseract

Building a custom training file was necessary due to the structure of a receipt. Tesseract’s default training file is trained with a dictionary file incorporated and works best with blocks of text. There is also the issue of many different fonts used on receipts, making OCR difficult.

The first step that I performed to train tesseract to recognize receipt text more accurately was to capture numerous receipts and clean them. I used my image processing algorithm in my application to accomplish this. I then used Adobe Photoshop to clean any artefacts or noise still present in the image. Once there was only text left in the image, I then extracted it and put it all on one line. This step was necessary due to line height issues that become an issue later in the process if this step is not completed. Tiff format was recommended for these files but caused errors, so instead, I used png format which resulted in clearer images with no errors. The next step was to create box files for each of the png files I had created. These box files contain a line for each character in the image in the format: P 228 22 243 57 0, where the leftmost character is the character that tesseract assumes the character is, with the rest of the digits being coordinates on the image, along with box heights and widths.
I used a java program called jTessBoxEditor to correct any mistakes made by Tesseract. I then created training files for each of the image-box pairs, followed by generating the unicharset file from the box files and a font properties file. The shape prototypes file, a file containing the number of expected features for each character, and the character normalization sensitivity prototypes, are then generated. The final step is to compile the data into the trained data file, ready to be used by the application.

4.4. Database and Cloud

The application takes advantage of AWS DynamoDB for storing receipt data and AWS S3 for storage of the receipt images. When the user chooses to save a receipt, the receipt is renamed from a temp file and named with a time stamp to give it a unique name. The file is then uploaded onto the user’s private storage on the S3 servers. The receipt’s data is also stored in the database with the location of the corresponding file also saved in the entry.

4.5. Statistics

To provide the user with statistics on spending data, I used the MPAndroidChart library to display various charts. The data is queried from the database depending on the month the user selects. This has been implemented in a custom list view to display various charts based on the data provided.

5. Validation

5.1. Code Scanning

I have used Lint, a code scanning tool, to help improve the structural quality of the code. Using this tool, I have improved usability, increased performance and fixed issues, allowing the application to conform to accessibility standards. I have found many areas where the code needed to be refactored and issues that could result in errors in the future. Accessibility has been improved by adding descriptions to the UI elements of the application. Performance has been improved as Lint detected areas of code that may result in using large amounts of memory. I then modified these areas to prevent this from occurring. It also helped locate methods that could possibly result in null pointer exceptions.

5.2. User Testing

I have conducted user testing on two separate occasions. Below is a summary of the usability document which is a combination of both occasions.

Audience Type

Student	7
Employed	4
Unemployed	1
TOTAL (participants)	**12**

Phone Usage

0 to 2 hrs. daily.	0
3 to 4 hrs. daily.	5
5+ hrs. daily.	7
TOTAL (participants)	**12**

Age

18-25	10
26-39	2
40-59	0
60-74	0
TOTAL (participants)	**12**

Gender

Women	6
Men	6
TOTAL (participants)	**12**

What participants did:
During the usability evaluation, participants were asked to complete 8 scenarios of “real-life” tasks on the app. The tasks were presented in linear order.

#	**Task**
1	Sign up
2	Log in
3	Capture and save two receipts
4	View the receipt in the calendar view
5	Navigate to the statistics screen and view the charts
6	Delete a receipt
7	Edit a receipt
8	Log out

Findings:

User testing 1

- Displaying "passwords do not match" notification falsely.
- Not allowing users to activate account.
- Case sensitive email addresses.
- Receipts being displayed mirrored when captured.
- Some fragment overlaying issues.

User Testing 2

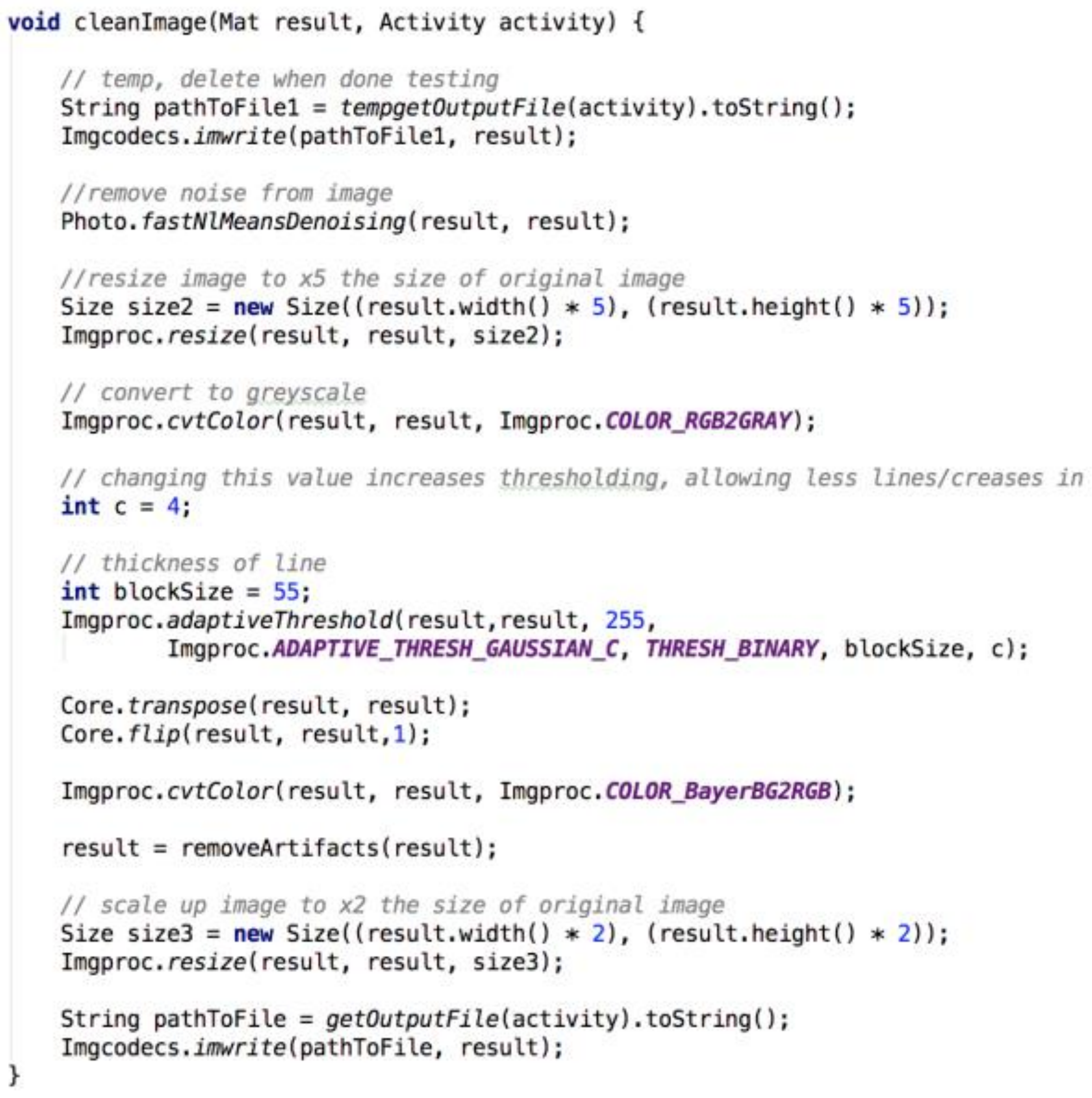
- The date database entry updated when user updated the date field but the formattedDate entry did not.
- The statistics activity crashed when there were no entries in the database.
- The category field did not update in the receipt edit activity
- The data shown on the pie chart in the Statistics activity was not made clear what it was displaying.
- List of receipts disappearing when user clicks the back button on edit receipt fragment.

5.3. Unit Testing

I have performed both unit testing and instrumentation testing on the application to cover a range of possible problems. My instrumentation testing is concentrated on the testing of the UI elements of the application. My unit tests are focused mainly on the text extraction class, as it is the area that has the most variable results. All written tests have passed successfully.

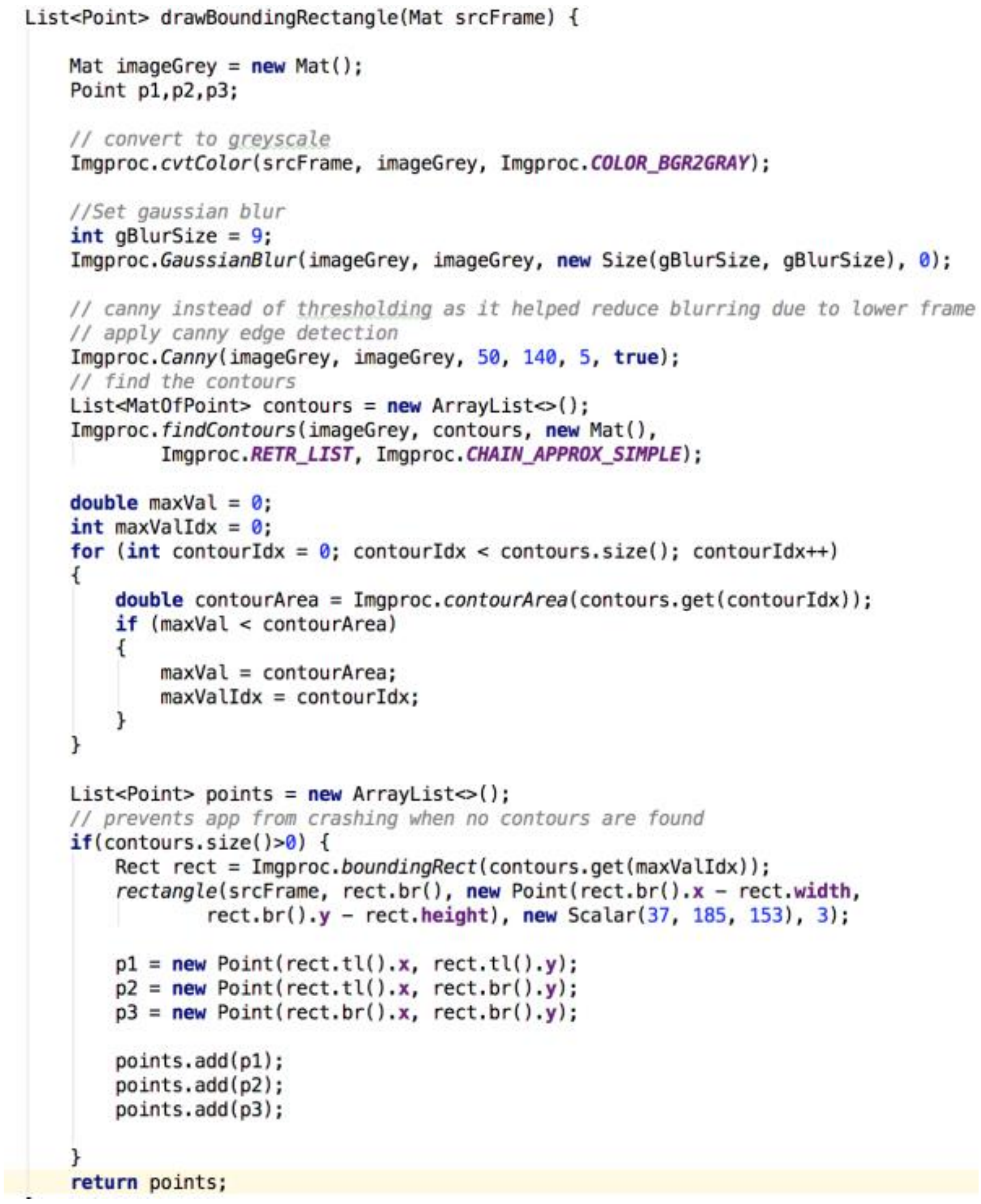
6. Sample Code

Method to clean the captured image and write it to storage:



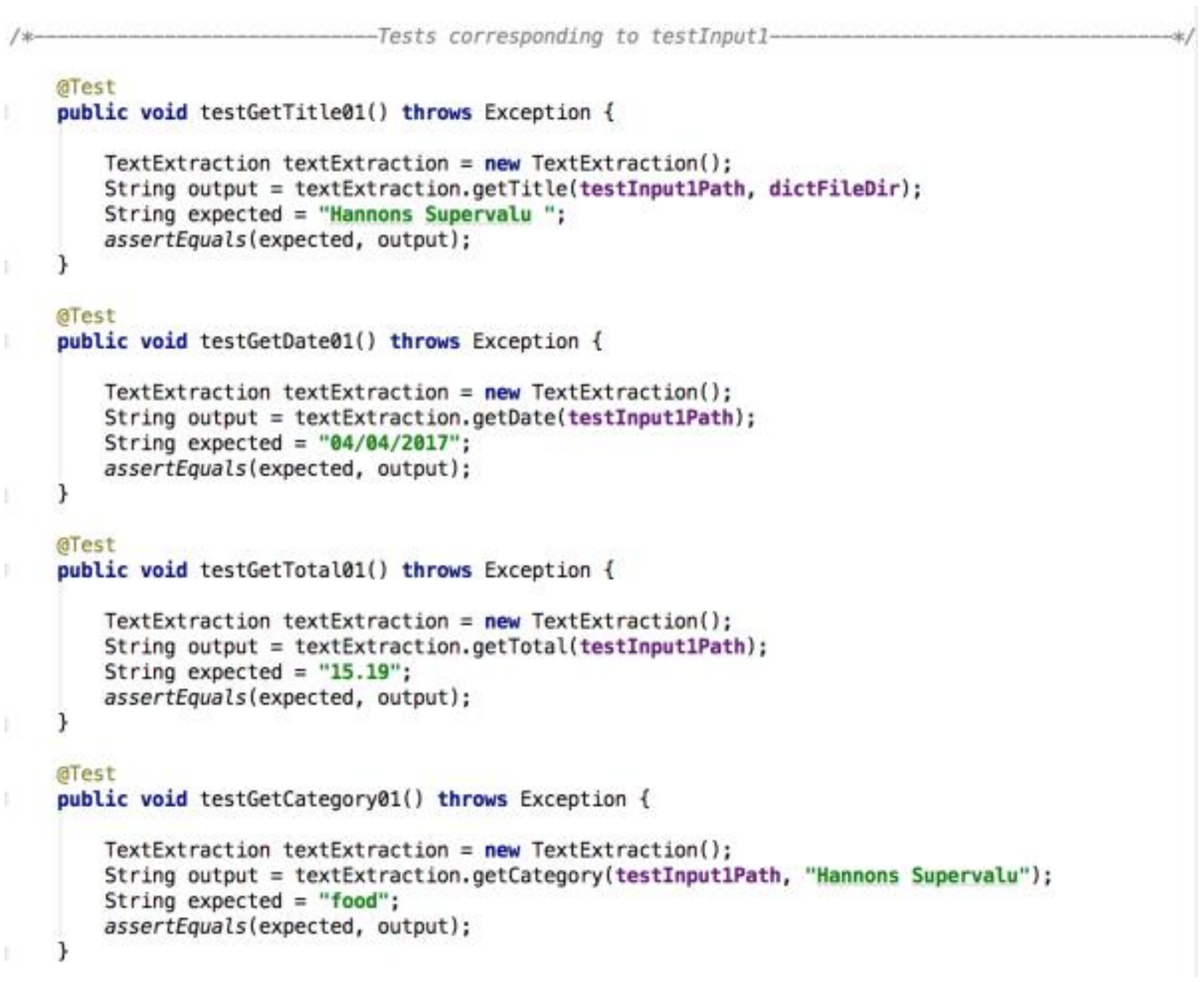
```
void cleanImage(Mat result, Activity activity) {

    // temp, delete when done testing
    String pathToFile1 = tempgetOutputFile(activity).toString();
    Imgcodecs.imwrite(pathToFile1, result);

    //remove noise from image
    Photo.fastNlMeansDenoising(result, result);

    //resize image to x5 the size of original image
    Size size2 = new Size((result.width() * 5), (result.height() * 5));
    Imgproc.resize(result, result, size2);

    // convert to greyscale
    Imgproc.cvtColor(result, result, Imgproc.COLOR_RGB2GRAY);

    // changing this value increases thresholding, allowing less lines/creases in
    int c = 4;

    // thickness of line
    int blockSize = 55;
    Imgproc.adaptiveThreshold(result,result, 255,
            Imgproc.ADAPTIVE_THRESH_GAUSSIAN_C, THRESH_BINARY, blockSize, c);

    Core.transpose(result, result);
    Core.flip(result, result,1);

    Imgproc.cvtColor(result, result, Imgproc.COLOR_BayerBG2RGB);

    result = removeArtifacts(result);

    // scale up image to x2 the size of original image
    Size size3 = new Size((result.width() * 2), (result.height() * 2));
    Imgproc.resize(result, result, size3);

    String pathToFile = getOutputFile(activity).toString();
    Imgcodecs.imwrite(pathToFile, result);
}
```

Method to draw a bounding box onto the frame before it is displayed to the user:



```java
List<Point> drawBoundingRectangle(Mat srcFrame) {

    Mat imageGrey = new Mat();
    Point p1,p2,p3;

    // convert to greyscale
    Imgproc.cvtColor(srcFrame, imageGrey, Imgproc.COLOR_BGR2GRAY);

    //Set gaussian blur
    int gBlurSize = 9;
    Imgproc.GaussianBlur(imageGrey, imageGrey, new Size(gBlurSize, gBlurSize), 0);

    // canny instead of thresholding as it helped reduce blurring due to lower frame
    // apply canny edge detection
    Imgproc.Canny(imageGrey, imageGrey, 50, 140, 5, true);
    // find the contours
    List<MatOfPoint> contours = new ArrayList<>();
    Imgproc.findContours(imageGrey, contours, new Mat(),
            Imgproc.RETR_LIST, Imgproc.CHAIN_APPROX_SIMPLE);

    double maxVal = 0;
    int maxValIdx = 0;
    for (int contourIdx = 0; contourIdx < contours.size(); contourIdx++)
    {
        double contourArea = Imgproc.contourArea(contours.get(contourIdx));
        if (maxVal < contourArea)
        {
            maxVal = contourArea;
            maxValIdx = contourIdx;
        }
    }

    List<Point> points = new ArrayList<>();
    // prevents app from crashing when no contours are found
    if(contours.size()>0) {
        Rect rect = Imgproc.boundingRect(contours.get(maxValIdx));
        rectangle(srcFrame, rect.br(), new Point(rect.br().x - rect.width,
                rect.br().y - rect.height), new Scalar(37, 185, 153), 3);

        p1 = new Point(rect.tl().x, rect.tl().y);
        p2 = new Point(rect.tl().x, rect.br().y);
        p3 = new Point(rect.br().x, rect.br().y);

        points.add(p1);
        points.add(p2);
        points.add(p3);

    }
    return points;
```

Sample of unit tests designed for the TextExtraction class:



```
/*----------------------------Tests corresponding to testInput1---------------------------------*/

    @Test
    public void testGetTitle01() throws Exception {

        TextExtraction textExtraction = new TextExtraction();
        String output = textExtraction.getTitle(testInput1Path, dictFileDir);
        String expected = "Hannons Supervalu ";
        assertEquals(expected, output);
    }

    @Test
    public void testGetDate01() throws Exception {

        TextExtraction textExtraction = new TextExtraction();
        String output = textExtraction.getDate(testInput1Path);
        String expected = "04/04/2017";
        assertEquals(expected, output);
    }

    @Test
    public void testGetTotal01() throws Exception {

        TextExtraction textExtraction = new TextExtraction();
        String output = textExtraction.getTotal(testInput1Path);
        String expected = "15.19";
        assertEquals(expected, output);
    }

    @Test
    public void testGetCategory01() throws Exception {

        TextExtraction textExtraction = new TextExtraction();
        String output = textExtraction.getCategory(testInput1Path, "Hannons Supervalu");
        String expected = "food";
        assertEquals(expected, output);
    }
```

7. Problems and Resolutions

7.1. Low accuracy while capturing receipts

An issue that I encountered was low accuracy when capturing the receipt image. My first implementation relied on finding the four corners of the receipt, cropping the remaining image out and rotating accordingly. This was problematic as all four corners of the receipt are not always well defined. For example, if the receipt was torn at one corner. There was also no way of knowing if this was performed correctly until after the user had already captured the receipt, which resulted in returning to the camera and trying to recapture. I resolved this problem by modifying the capture activity to perform real-time image processing by surrounding the receipt in a bounding box. The user can now easily see if the receipt has been identified before capturing the receipt, allowing them to achieve a more accurate result. The only drawback to this approach is that the image is of lower resolution when first captured, which I have adjusted by scaling up the image accordingly before performing OCR.

7.2. External Light Sources

An issue that caused many problems was external light sources, which affected the quality of the captured image. My first solution for this was to force the device's flash to initiate when capturing the image. This resolved the problem. However, it caused an issue with creases in the receipt becoming very prominent in the processed image, making the OCR very inaccurate. I decided to roll back on this idea and adopt a different approach. I adjusted the OpenCV binary thresholding algorithm to a level where it worked relatively well for most of the external light sources, allowing darker regions of the image to be passed through (image text) and removing all other lighter regions (creases or artefacts) that may have been present. I have since tested with the device flash re-enabled, which resulted in counter-intuitive results.

7.3. Tesseract and Creased Receipts

Very creased receipts have been an issue. I have tried to counteract this problem as much as possible in the same way I have resolved the external light sources problem, by adjusting the thresholding level to a point where it will let the darkest regions of the receipt through. This still did not resolve the problem as prominent creases remained evident in the processed image. Instead, I implemented an algorithm that masks areas it recognises as text, removing anything else present on the receipt. This method works relatively well, but can be problematic if large creases run through areas of text. This can cause that area of text to be removed along with the crease.

8. Results

The final result of the project has been successful. The application can access the devices camera. It can capture and process images into text. It successfully connects to the database. The processed receipt data can be stored in the user's database. The application successfully enhances and corrects the receipt. The application allows the user to register and allows the user to log in, and successfully displays graphical representations of the users purchase history. The final result satisfies nine out of the eleven functional requirements present in the functional specification document, with all of the highly critical requirements satisfied.

Due to the fact that I have built an application based on optical character recognition, a 100% success rate in always correctly extracting text from receipt is difficult. This is why I have allowed the user to make adjustments to the extracted text if text has been incorrectly read. Overall, the feedback from the user testing conducted has been very positive.

9. Future Work

9.1. Camera2 API

I plan to extend my application by implementing the newer Camera2 API, which is present in the Android library. My reasons for doing so is that the current implementation of the OpenCV Camera is based on the deprecated Camera API and does not function as well on newer devices, which camera hardware's have been built with the Camera2 API in mind. Another reason for this is that Camera2 is far more versatile and allows much more customisation and control over the camera. The issue of a low frame rate while processing in OpenCV is common in many implementations of OpenCVs Camera, as well as a low maximum resolution when capturing frames, which I hope to resolve when upgrading to Camera2.

9.2. Exporting .CSV Files

Allowing the user to export receipt data to a .CSV file is another plan for the future of the application. This will allow the user to view or store their captured receipt data along with previous financial records they may have. This removes the fact that the user's purchase information is tied exclusively to the application which, may deter some users.

9.3. Capturing more data from the receipt

At present, the information that can be captured from the receipt is quite basic. To extend this, information such as time of purchase could be captured and stored. This could be worked into the statistics section of the application to allow the user to observe trends, such as their most active spending hours. V.A.T could also be captured from the receipt to allow users with small businesses to easily calculate what percentage of tax they can claim back. The location of the purchase may also be captured from the business location on the receipt.

9.4. Capturing purchase information from email

One of the initial requirements in the functional specification was to allow the application to optionally scan emails for online purchases and prompt the user to store the purchase data in the database. Unfortunately, I did not achieve this requirement in the time provided, however, I feel that it would be a very useful feature in the application and I hope to include it in the future.