**מבנה כללי** (בהתאמה עם התרשים)

* MainActivity (BillsMainActivity) – הוא ה host activity של כך ה fragments. מממש את כל ה interfaces של ה fragments כדי לתקשר איתם ולנהל אותם. כל fragment מגדיר את ה call backs אותם צריך לממש(interface) כדי לתקשר איתו.

כשעולה האפליקציה ישר מתבצע transaction ונקרא WelcomeScreen.

כל חזרה עם call back מאחד ה fragments גוררת קריאה ל fragment אחר ע”יtransaction.

* WelcomeScreen (WelcomeScreenFragment) – מגדיר 2 buttons שקוראים ע"י call back ל Camera ול Summarizer בהתאמה. המימוש שלהם ב MainActivity אבל ההצמדה ל buttons ב fragment.
* Camera (CameraFragment) – מגדיר call back שנקרא onCameraSuccess שקורא ל Analizer אחרי שהמצלמה מסיימת(onCameraFinished) עם התמונה שהתקבלה.

כמו כן Camera מייצר אובייקט של CameraRenderer המנהל את כל התקשורת עם המצלמה (מממש את ה interface שנקרא TextureView.SurfaceTextureListener). אובייקט זה גם דואג שהתמונה תצא ברזולוציה שהכי קרובה ל 3264x2448 כדי שהאלגוריתמיקה של העיבוד תמונה (שמתבצעת ע”יOpenCV) תעבוד תמיד.

* Analyzer (BillAnalyzerFragment) – מגדיר 2 call backs. במקרה שהניתוח של החשבונית נכשל אזי BillsLog כותב את השגיאה ל database ונקראonBillAnalyzerFailed עם התמונה ועם ה path ל storage אליו מעלים את התמונה. כמובן שחוזרים ל WelcomScreen והמשתמש מקבל toast עם השגיאה.

במקרה שהניתוח (הסרת הרקע ו TM לזיהוי האזור. קריאה ל OCR) מצליח קוראים ל onBillAnalyzerSucceed עם רשימה של BillRow המכיל בתוכו כמות, מחיר ותמונה של המוצר עבור כל שורה לצורך הצגתם ב Summarizer. כמו כן אנו מעבירים את ה pass code לצורך שיתוף של תוצאות הניתוח.

* Summarizer (BillSummarizerFragment) – לא מממש call backs. כשנקרא ע"י check passcode (שזה ה call back שנקרא start summarizer) אזי הוא מאותחל עם ה path ל db וה storage (ללא ה BillRows) וכמובן ה pass code ומתבצעת משיכה של המידע (כמות, מחיר ותמונה של המוצר). לעומת זאת בקריאה של onBillAnalyzerSucceed מתבצעת העלאה של המידע ל storage, db ואז קריאה ל summarizer עם השורות שנותחו. ה Summarizer כבר ידאג להעלות אותם ל path הנכון כך שיתר המשתמשים יוכלו למשוך אותם.

בנוסף יאותחל אובייקט של UiUpdater שינהל את ההתממשקות עם ה db, storage כשהיוסרים בוחרים את המוצרים שלהם.

**Analyzer**

* Init - מאתחלים ocrEngine ב thread נפרד כשנקראת הפונקציה Init.
* OnActivityCreated - מבצעים את הפעולות הבאות(thread נפרד):  
  ממירים את ה byte array של התמונה ל mat כדי שנוכל לעבוד עם opencv. בנוסף ממירים ל bitmap לצורך הצגה למשתמש ב DragRectView.

מאתחלים אובייקט של BillAreaDetector שהוא אחראי לזהות את ה edges של החשבונית ולהחזיר את הקואורדינטות כדי שנוכל לצייר אותם על ה DragRectView כך שהמשתמש יוכל לבצע תיקונים לזיהוי. כשמעדכנים את ה DragRectView בקואורדינטות שזוהו, משתמשים ב runOnUiThread כדי למנוע פניה לא חוקית של ה thread ל ui thread.  
בנוסף מגדירים button שברגע שה user מאשר את הזיהוי אזי תרוץ הלוגיקה שמנתחת את תוכן החשבונית ותבצע ocr על המחיר והכמות.

* Analyze - מבצעים את הפעולות הבאות(thread נפרד):

מבצעים המרה של הקואורדינטות מה DragRectView כדי לדעת כיצד לבצע crop לתמונה. לאחר מכן נשלח את ה mat ל WarpPerspective עם הקואורדינטות וכך נבצע חיתוך ומתיחה של התמונה באופן פרופורציונאלי.  
נשכפל את התמונה כדי לבצע עיבוד מקדים ע"י ImageProcessingLib עבור ה TM וה OCR.

לאחר מכן נשלח את התמונה פעם אחת ל TM לזיהוי המיקומים של הכמות, מחיר ומוצר ופעם שניה לצורך זיהוי ה components של הכמות והמחיר. בסוף נקרא ל onBillAnalizerSucceed.

**Summarizer**

* יש 2 init. אחת משמשת לקריאה מ StartSummarizer (בעצם check passcode) והשניה כשה Analyzer מצליח. בשניה יעברו גם התוצאות (רשימה של BillRow) של פענוח השורות.
* לאפליקציה יש 2 מודים: MainUser, SecondaryUser. משתמש ראשי מצלם את החשבונית וחולק אותה עם כל המשתמשים המשניים. הפונקציות של הראשי והמשני נקראות בהתאמה וה UiUpdater מתחיל את עבודתו בהתאם. כמובן שה UiUpdater עובד ישירות מול ה Firebase ומעדכן\מתעדכן בהתאם.

**UiUpdater**

* כיצד פתרנו את בעית ה race שהייתה כש 2 משתמשים בוחרים בו זמנית את אותו הפריט?

**תיאור הבעיה:** כש 2 בוחרים בוז אזי בא נסתכל מה יקרה אצל כל אחד מהם: תיקרא גם onClick וגם onChildChange. נזכיר ש onChildChange נקראת אצל המשתמש המשני ע"י push notification מה FB כאשר המשתתף הראשי בחר מוצר(ולהיפך). לכן תהיה גישה ל UI מ 2 פונקציות שונות(שמעדכנות את אותם משתנים ולכן ערך המשתנים לא צפוי).

**פתרון:** תחמנו את הקוד של 2 הפונקציות ב synchronized עם אותו מנעול שהוא private final מה שמונע גישה מה tread ל 2 הפונקציות בוז. אסור לשכוח שקיימת גם פניה ל DB אך אנחנו לא חוששים מהתנגשות(נניח ו 2 משתמשים פונים יחד ל DB) שם כיוון שהקריאות האלה מבוצעות ע"י trasactions מה שמבטיח פעולה שלימה ללא הפרעות. פעולת transaction יכולה להכשל או להצליח אבל התוצאות מוגדרות היטב (צפויות!).

* StartMainUser – מעדכנים את ה UI בתוצאות שפוענחו ע"י AddRowToUi. לאחר מכן מגדירים את ה ValueEventListener, ChildEventListener לרפרנס של ה DB. מכל ה interface שמתחייבים ה Listeners לממש, אנו נממש רק את onDataChange, onChildChange בהתאמה. כשמקליקים על מוצר נקראות הפונקציות הבאות בסדר הבא:  
  1. OnClick – בהקלקה על מוצר נעביר אותו מ’מוצרים משותפים’ ל ‘המוצרים שלי’. כמו כן נחסיר את הערך 1 מה DB (ב doTransaction – חלק מהטרזקציה הזאת היא קריאה ל2 הפונקציות הבאות). לאחר ש 1,2 ירוצו תיקרא onComplete ובה נבצע כתיבה ללוגים ועדכון של סופי של ה total sum של ‘המוצרים שלי’ ושל ‘המוצרים המשותפים’.

2. onChildChanged – נעדכן את ה total sum ב view ובמידה וה quatity של המוצר 0 נסיר אותו מה view.  
3. onDataChange – מקבלת מה DB את ה data snapshot ומעדכנת את ה UI בהתאם (גם את המחיר סה”כ). נקרא בפעם הראשונה שה DB עולה נראלי שכדי לעדכן את ה UI. מדוע צריך לעדכן פעמיים? הרי עשינו זאת ב onChildChanged?  
**תשובה:** ה ValueEventListener וה ChildEventListener עושות כמעט את אותו הדבר. אחד ההבדלים הינו שה ChildEventListener מאפשרת יותר גמישות. כלומר, יש ל interface שלה 4 פונקציות לעומת 2 ב ValueEventListener. בנוסף ChildEventListener מספק עותק בלתי ניתן לשינוי (immutable copies of the data) של המידע. יש עוד הבדל אחרון וניתן לקרוא עליו בלינק(כל התשובה מבוססת עליו):

<https://stackoverflow.com/questions/40764292/what-is-the-difference-between-childeventlistener-and-valueeventlistener-firebas/40776270?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa>

* StartSecondaryUser – כמו ב main user עם שינוי אחד קטן: נקראת הפונקציה onChildAdded (שלא ממומשת ב main user) והיא מושכת מה storage את ה product bitmaps ומוסיפה אותם ל UI. **פעם אחת מתבצעת המשיכה הזאת והתמונות נשמרות ב LIST גם אם משתמש אחר משך אותם מ UI! במידה ומישהו משחרר מוצר, המשתמש הנוכחי שולף אותו מה LIST (גם אצל המשתמש הראשי זה ככה רק שהוא מחזיק LIST עם התמונות שפוענחו מהצילום שלו)**

הערת שוליים: ניתן לבצע שינויים לפי ה**תשובה** אבל לא בצעתי אותם. בכל זאת ניתן לחשוב על שינוי חכם ולדבר עליו בראיון. כי יש שיכפול קוד ב 2 הפונקציות.

**FirebaseUploader**

* מגדיר interface עם 2 פונקציות(call backs) שכל מי שמייצר עותק של ה class מתחייב לתת מימוש: OnSuccess, OnFail. הרעיון הוא שבמקרה של כישלון(למשל בעיית תקשורת) נרצה להדפיס שגיאה ללוגים או להתנהג באופן מסויים.
* מעלה ל DB ול storage את ה price, quantity, product bitmap בהתאמה (גם מעלה את כל התמונה של ה bill במקרה של כשלון). העלאה מתבצעת רק בפונקציה UploadRows באופן הבא.  
  תחילה נעלה את תמונת ה bill. נגדיל storage reference עם שם הקובץ שנרצה ונאכלס אותו ב bytes. לאחר מכן נגדיר upload task של FB ו call backs של הצלחה וכשלון. לאחר מכן נעלה את התמונה ואת ה metadata שלה (מה זה ה metadata)?  
  כעת נעלה את ה rows באופן הבא. נקצה buffer ל product bitmap ונעתיק לתוכו את ה bytes. לאחר מכן ניצור metadata שם נשמור את ה width, height, type של התמונה וגם את ה price, quantity. בסוף נעלה את ה data באותו אופן שהעלנו את התמונה של ה bill. בסוף קוראים ל updateChildren עם ה dbItems שמאכלס בתוכו את כל המידע.

**Firebase**

* ה storage וה database של האפליקציה הם בעצם ה Firebase.
* database – קיימות 2 תיקיות. הראשונה userIds שממפה id של כל משתמש ל pass code שלו ול path הרלטיבי ב database (ה path הנוכחי ששייך ל session הפעיל האחרון). התיקיה השניה היא users שמחזיקה את כל המידע עבור כל משתמש. לכל משתמש קיימת תיקיה של session (כאשר הוא מצלם) ו 2 תיקיות של logs. אחת השייכת ל session אליה יכתבו כל המשתתפים ב session והשניה (ישירות תחת users) המשמשת ללוגים ספציפיים של ה user. לאורך כל ריצת האפליקציה המשתמש כותב לתיקיית הלוגים שלו. כאשר אנו בתהליך של ‘משיכת’ מוצרים אזי המשתמש יכתוב גם לתיקיה שלו וגם לתיקיית הלוגים של ה session אליה כמובן יכתבו גם המשתתפים האחרים במקביל. כל שורת לוג ב session מחזיקה גם user id כדי שנדע לשייך אותה ולעקוב אחרי כל התהליך במקרה של שגיאה. ברגע שה main user נכנס לאפליקציה מאותחל עבורו private session פעם אחת **וכל** הלוגים נכתבים לשם. כשהוא יצלם את החשבונית יאותחל עבורו common session אליו יכתבו גם המשתתפים האחרים. בכל צילום מחדש יאותחל common session חדש(בעצם timestamp חדש). כאשר secondary user יכנס לאפליקציה יאותחל עבורו private session כמו אצל ה main user. כאשר הוא יקליק על check passcode אזי יתחבר ל common session ויכתוב לוגים לשם.  
  בנוסף קיימים ה rows. ברגע שמשתמש ‘מושך’ מוצר אזי ירד ה counter של המוצר. כאשר מתבצעות 2 טרנזקציות במקביל על אותו מוצר אזי רק אחת תצליח והמשתמש יקבל דיווח על שגיאה.
* storage – מחזיק תיקיה לכל משתמש לפי ה id שלו. בתוך כל תיקיה כזאת נשמרים כל ה sessions של המשתמש. לכל session אנו שומרים את כל התמונות ה products ואת התמונה ב bytes. את המוצרים נצטרך כדי שכל המשתתפים יוכלו ‘למשוך’ אותם, ואת החשבונית כדי שנוכל לשחזר באגים.
* FirebaseLogger – מממש interface שנקרא Ilogger אותו מתחייב לממש כל class שרוצה לכתוב לוגים. בפונקציה Log מתבצעת טרנזקציה (doTrasaction, onComplete) שכותבת לוג ל path המתאים ב database. במקרה שהיא נכשלת **ניתן** לשלוח מייל (עדיין לא מומש).
* FirebaseUploader – משתמשים בו ב 2 מקרים כדי להעלות מידע ל storage. הראשון הוא להעלות את ה bill image במקרה שה Analyzer נכשל (מה לגבי מקרה של הצלחה?). השני הוא להעלות את ה rows שפוענחו אחרי שה Analyzer מצליח. העלאה מתבצעת ע"י onFailureListener, onSuccessListener.
* **Sessions** – עבור ה MainUser - מאתחלים private session רק פעם אחת כשהמשתמש עושה sign in ב onCreate של ה main activity. שם גם נייצרUUID ו timestamp. כמו כן נשמור מיפוי של sessionId ל ILogger ב BillsLog כדי לדעת לאן לכתוב בהינתן sessionId מסוים (הפונקציה Log ב BillsLog). במקרה הזה אנו כותבים ל FirebaseLogger (שכמובן מממש את ה interface שנקרא Ilogger). באתחול ה session גם מייצרים את ה log/storage paths כדי שנוכל לחלחל אותם לשאר הקופוננטות.

עבור ה Secondary User – מבצעים את אותו תהליך עם אתחול של private session אבל דורסים אותו עם אתחול נוסף של private session for second user. באתחול זה מגדירים את תיקיות הלוגים עבור ה common/user logs. כמובן שנעזרים ב pass code כדי לגלות את ה path ל common וכדי לדעת את ה timestamp לתיקית הלוגים של המשתמש.  
עבור Logs – לכל Common Session תיווצר תיקייה חדשה תחת ה user. בעצם כל פעם שהמשתמש יצלם חשבונית. כמו כן תיווצר תיקיה אחת עבור לוגים פרטיים של המשתמש תחת Logs של לוגים פרטיים. לתיקיה זו תהיה אותה חותמת זמן כמו לזאת של ה Common. כמו כן יווצרו תתי תיקיות תחתיה עבור כל פעם שהמשתמש מצלם מחדש. התיקיות האלה מקבילות לתיקיות שנוצרות עבור common session (אותה חותמת זמן! ראה תרשים).

**Passcode**

* PassCodeResolver, GetPassCode - הפונקציה StartCamera משתמשת ב PassCodeResolver ולכן מתחייבת לממש את ה interface שלו (onPassCodeResolve, onPassCodeResolveFail) ולשלוח את המימוש ל GetPassCode. במקרה של הצלחה StartCamera תקבל את ה path היחסי ל db/storage שנוצר ואת ה passcode שנוצר(לצורך שיתוף עם המשתתפים).
* PassCodeResolver, GetRelativePath – מחזירה את ה path היחסי בהינתן passcode כמובן לפי הרשימה של ה userIds שנמצאת ב db. כמו כן ברור ש passcode יחודי למשתמש.

**Tests**

קובץ TestBench מכיל את ההגדרות של ה Test אותו רוצים להריץ וכמו כן מייצר TestBill לכל חשבונית שירוץ ב thread נפרד. הרצת וניהול ה threads תתבצע ע"י thread pool. כשנקצה אותו נגדיר מספר פרמטרים בניהם max pool size.

למעשה TestBill מבצע את כל ה flow של ה Analyzer. כל ה threads כותבים את התוצאות(אחרי שמשווים ל expected txt) ל ConcurrentLinkedQueue (שפותר מאחורי הקלעים את בעיית ה producer-consumer), ובסוף ה main thread מסכם ומדפיס לקובץ.

משפט חשוב שמסביר את הצורך בthreadpool כמו כן מדוע הוא עדיף על ליצור בעצמי threads כשמדובר בכמות גדולה:  
A thread pool helps mitigate the issue of performance by reducing the number of threads needed and managing their lifecycle.

**החלפת tess two ב ABBYY**

לצורך ביצוע ocr השתמשנו ב Open Source שנקרא tess-two מהלינק הבא:  
<https://github.com/rmtheis/tess-two>

למעשה מדובר ב Android API שמאפשר להשתמש ב Tesseract 3.05 על אנדרואיד. לאחר בדיקות שהרצנו על כ 10000 חשבוניות גילינו שהוא לא עובד ככ טוב ולכן החלטנו לנסות לשלב SDK של חברה מוכרת בשוק שנקראת ABBYY. למעשה התכנון היה להמשיך איתם עד ש tess-two יתמכו ב

Tesseract 4 שמשלב Neural Networks ולכן משפר משמעותית את הזיהוי עם הזמן.

**הכל על DL**

החלטנו להחליף את ה BillAreaDetector ב DL כיוון שהזיהוי לא עבד טוב ככ (בניית גרף לפי צבעים – מפורט בשלב מאוחר יותר). לצורך זה מצאנו Pretrained model שעושה edge detection שנקרא HED ועשינו לו training עם תמונות שלנו (מה שנקרא transfer learning). השתמשנו ב TF כדי לאמן ולבדוק אותו. לאחר מכן ביצענו המרה שלו ל tf Lite עם converter של גוגל שנקרא toco (<https://www.tensorflow.org/mobile/tflite/>).

כעת יש לנו API ש"מדבר” עם kernels שנכתבו ב C++ שרצים על ה GPU. ההמרה לtflite לא משנה את הגודל של קובץ המודל. ניתן להקטין אותו פי 4 אם ממירים ל quantized model (הכל קטן ל int8 מ float) ואז ל tflite.

בעצם ה API ב Java מבצע הקצאה של buffer אותו הוא שולח לשכבת ה C++ דרך JNI ומקבל בחזרה מערך של buffers עם התוצאות (בעצם tensors). במקרה שלנו מספר תמונות בינאריות של edges שיצאו ממספר יציאות בגרף. נבחר את התמונה הטובה ביותר ועליה נריץ connected component algorithm כדי למצוא את המרובע הגדול ביותר.

**Camera (billslib)**

מממש את ה interfaces של Runnable, TextureView.SurfaceTextureListener. מי שממש את ה Runnable מתכוון להשתשמש בפונקציה run כדי להריץ פיסת קוד ב thread נפרד. ה interface השני הוא כדי "לשלוט” בפונקציונאליות של המצלמה (flash mode, capture size atc).

מימשנו Custom Camera כדי לשנות את ההנהגות הרגילה של המצלמה: רצינו לשנות את הרזולוציה של התמונה למשהו קבוע (או לקרוב ביותר אליו ע”י gcd וחלוקה בו וחישוב כלשהו). השינוי הזה עוזר לנו להריץ אלגוריתמיקה קבועה על התמונה שתעבוד ברוב המקרים(preprocessing).

**Contracts (billslib)**

מורכב מ 2 תיקיות:

תיקית enums שכל אחד בקובץ נפרד.

תיקית interfaces. למשל IOcrEngine שמממש אותו TesseractOCREngine. הרעיון הינו שבמידה ונרצה להחליף את ה tess-two במשהו אחר נוכל לממש את ה interface הזה ולא נצטרך לבצע שינויים גדולים בקוד.

בנוסף קיים class שנקרא PreparingEnvironmentUtil שמעתיק את הקבצים הרלוונטיים מ assets למכשיר לפני שהאפליקציה רצה (tess data למשל). במקרה של העתקה לאמולטור יש טיפול אחר למשל.

**Core (billslib)**

**BillAreaDetector:**

נתאר את האלגוריתם של GetBillCorners:

נמיר את התמונה ל gray ונחשב היסטוגרמה שלה ולפי הצבעים של התמונה נחפש מרובע לבן ונחזיר את הערכים של הפינות שלו.

**BillsLog:**

מחזיק מיפוי של UUID ל ILogger כדי שנוכל להדפיס ל Path הנכון בהינתן session. בעצם FirebaseLogger מממש אותו ומחזיק בתוכו את ה paths. למשל, ניתן להחזיק 2 מיפויים ל 2 FirebaseLogger. כל אחד מהם נוצר בתחילת session. נניח ונוצר session חדש ועדיין ה session הקודם לא סיים לכתוב את הלוגים, אזי המיפוי פותר בעיה זו.

**ImageProcessingLib:**

* WarpPerspective – משתמשת בפונקצית OpenCV המקבלת mat ו 4 נקודות ומבצעת חיתוך ו"מתיחה” של התמונה. חיתוך בנקודות שזוהו ע"י BillsAreaDetector. “מתיחה” לרוחב\גובה המקסימלי של רוחב עליון לעומת תחתון (או גובה ימני לעומת שמאלי). כמובן שהחישוב הכבד הזה מבוצע ב thread נפרד.
* PreprocessingForTM/PreprocessingForParsing – נבצע AdaptiveThreshold על התמונה עם פילטר GAUSSIAN\_C כדי לקבל תמונה בינארית ונקיה. ניסינו פילטרים אחרים כגון MEAN/GLOBAL THRESHOLDING אבל לא התקבלה תמונה נקיה ככ.

לאחר מכן נבצע Erode/Dilate כדי להצר את הפונטים ולחבר פונטים שנחתכו. הגענו למסקנה שכדאי להשתמש בפילטר VERTICAL\_LINE ב Erode

ו HORIZONTAL\_LINE ב Dilate. השימוש בקו אנכי ב erode נובע מהרצון להפטר מקווים דקים ואופקיים בחשבונית.

**MainActivityBase:**

רצינו לרשת מ Activity בסיסי לצורך ביצוע פעולות שחוזרות על עצמן בכל יצירה של אפליקציה חדשה (היו לנו אפלקציות משנה למשל testsdebug). הפעולות שבוצעו שם הם בקשת הרשאות בזמן ריצה על כל מכשיר שמריץ אנדרואיד 6 (מרשמלו) ומעלה. כמו כן הרצה של PreparingEnvironmentUtil.*PrepareTesseract.*

**TemplateMatcher:**

למעשה ה class הכי מורכב שיש לנו באפלקציה. מקבל את 2 התמונות שעברו preprocessing ומבצע Match על הראשונה כדי לזהות את ה bill area, למצוא את המיקום של המילים\מספרים באיזור זה ולשמור אותם למערך של Rect. לאחר מכן ה Parsing יעבור על המיקומים (רק של ה price, quantity), יפענח אותם בעזרת OCR וישמור את המספרים שהתקבלו.

קצת פירוט על ה Match:

* משתמשים ב GetWords כדי לקבל את המיקומים של כל המילים בחשבונית מה OCREngine.
* ממלאים LinkedHashMap עם מיפוי של Rect-→Rect עבור כל מילה שמקבילה למילה בשורה הבאה (הצד הימני שלה. בנוסף אפשרנו חריגה של 0.4 כפול הגובה של המילה). בעצם ננסה ליצור מין גרף (לא קשיר כנראה) של מילים מחוברות.
* ננסה למצוא את הרצף הגדול ביותר של שורות מחוברות שבכל אחת יש 3 חיבורים. נבחר את הרצף הגדול ביותר.
* נבצע זיהוי של עמודת ה product ע"י חישוב ממוצע על אורך השורות בעמודה. מה שיעבוד ברוב המקרים. נשמור למערך של Bitmap את כל ה products ולמערך של Rects את כל המיקומים של ה price, quantity.

קצת פירוט על ה Parsing:

* ניקח את המיקומים שזיהינו ב Match ו ונעבור על התמונה (שעברה את ה preprocessing השני) ונפענח בעזרת OCR את המספרים.

**TesseractOCREngine:**

מממש את ה interface שנקרא IOcrEngine. בנוסף מממש פונקציות כמו SetImage, GetWords, GetUTF8Text, MeanConfidence, SetNumbersOnlyFormat.

למעשה מממש פונקציונאליות שחשובות לנו כדי להתמודד עם תמונת טקסט.

בהתחלה השתמשנו בו גם לצורך הזיהוי של המילים וגם בפענוח. בשלב מאורח יותר החלפנו רק את ה OCR ב ABBYY. התכנון העתידי היה להתמש במודל DL שעושה זאת

**Views**

**DragRectView:**

יחד עם fragment\_drag\_rect\_view.xml.

למעשה יצרנו FrameLayout שמחזיק בתוכו CustomView שנקראDragRectView (שעושה extends ל View) שלתוכו אנו "שופכים” את המידע בזמן ריצה. בנוסף הגדרנו button כדי שהמשתמש יאשר את הזיהוי של ה bill.

למעשה ב extends ל View עשינו Override לפונקציות הבאות: onTouchEvent, onDraw, setBackground.

למשל ב onTouchEvent גרמנו לפינה של המרובע לזוז. כמו כן הגבלנו תזוזה אסורה. הפונקציה הזאת מקבלת MotionEvent.

כשמשתמשים ב DragRectView אז עושים setBackground כדי שה View יכיל את התמונה של החשבונית. עליה מציירים את ה lines ואת ה circles ע"י canvas.

**fragment\_welcome\_screen.xml:**

כמו בקודם FrameLayout תוחם את הכל.

מורכב מ ImageView שמגדיר את הרקע ו LinearLayout שמגדיר את התכולה.

ב LinearLayout יש עוד ImageView עבור ה icon של ה bills ועוד LinearLayout שמגדיר בתוכו את ה buttons ו EditText כדי להזין את ה passcode.

ב WelcomeScreenFragment מבצעים findViewById כדי למצוא את ה buttons ואז מגדירים onClickListener עם קריאה לcall back הנכון.

**fragment\_bill\_summarizer.xml:**

למעשה ה view הכי מורכב שיש לנו.

מה שתוחם כאן את הכל זה RelativeLayout. המבנה הכללי נראה כך:



נסביר קצת. חילקנו את המסך ל 2 איזורים של ScrollView. אחד עבור ה’מוצרים שלי’ והשני עבור ה’מוצרים המשותפים’. מה שמפריד בניהם הוא ImageView של spliter שניתן להזיז למעלה\למטה. ה LinearLayout התחתון הינו עבור ה Tip. כשנקליק על ה Tip יפתח NumberPicker כי הגדרנו onClickListener על ה EditText של ה Tip.

**Exeception Handling**

לאורך כל האפליקציה אנו כותבים לוגים ל FB וגם לוקלית במקרים של חוסר תקשורת. ברגע שתהיה תקשורת נשלח את הלוגים ל FB (אנו לא שומרים לוקלית רשמתי כתכנון עתידי).   
במקרה שהאפליקציה קורסת נרצה לקבל לוגים על הקריסה. נעשה זאת באמצעות פונקציה שנקראת SetDefaultUncaughtExceptionHandler. בפונקציה ניצור thread ממנו נשלח מייל עם פרטי המכשיר\מערכת הפעלה ומשתמש. כמובן שגם במקרה שהאפליקציה קורסת, זאת הפעולה האחרונה שהיא תבצע.

**Utilities**

מספר פונקציות שהשתמשנו בהן בהרבה מקומות בקוד. למשל: קריאת תמונה מ byte array file, שמירת קובץ תמונה, יצירה של timestamp, המרה של mat ל bmp וכו