# QuickBooks Self-Employed iOS Application

Braden Young
Computer Science Department
California Polytechnic State University
2014

# Abstract

The goal of this senior project is to build an iOS application for an Intuit product called QuickBooks Self-Employed. The product helps self-employed individuals do their taxes. The iOS application's purpose is to allow users to categorize their transactions on the go and keep them updated on their requirement to pay quarterly estimated taxes. I was able to collaborate with other members on my team at Intuit to bring the app from concept, to reality. We were able to design each piece of the application then implement it in Objective-C using Apple's iOS SDK and a third party library. The project requirements were met and we were able to release two versions to the iOS App Store.

# Table of Contents

# 1 Introduction

From July 2014 to December 2014, I had the opportunity to do an internship at Intuit. I was placed on a team to work on the iOS app for a new product called QuickBooks Self-Employed. My main role was to develop the app with two other teammates. Many others on our team worked on the web application and the service's backend that all of the frontend applications coordinate with. I also worked with our designers on the user experience of the app. The app was shaped by many others in a similar way, such as my teammates working on the Android application.

In this paper, I will discuss the iOS app that we built – all the way from design decisions to some implementation details. For the content on the app, I will only discuss parts that I wholly implemented or heavily contributed to. To start, let's take a look at what QuickBooks Self-Employed is all about.

## 1.1 QuickBooks Self-Employed

Over 53 million Americans, or 34% of the U.S. workforce, are freelancers[1]. This number only seems to grow as new avenues for freelance work, such as Uber, Etsy, and TaskRabbit, become more popular. These individuals are either making some money on the side of another job or they are successful enough to do this work full-time. However, the income that these people earn is still an interest of the IRS. The IRS requires these "self-employed" individuals to pay taxes on the income they earn. This means they have to fill out new forms and keep closer track of their finances. This is where our customers' problem comes from.

[1] Napach, Bernice. "Freelance Nation: One-third of U.S. Workers Are Freelancers." *Yahoo Finance*.

## 1.2 Understanding the Problem

To better understand why doing taxes is a problem for a self-employed person, let's take a look at a hypothetical example. Greg works at a local restaurant and is a driver for Uber. Greg has to do quite a bit to take care of his taxes. He will have to fill out a Schedule C, Schedule SE, Form 1040, make estimated tax payments using Form 1040-ES and receive a 1099-MISC. We won't get into the gory details, but in order to do this appropriately, Greg has to correctly report his income he made driving. If Greg wants to save money through tax deductions, he will have to keep track of how much he spends on gas and how many miles he drives for work. He will also have to keep records as proof (such as receipts), in case the IRS comes knocking at his door asking for it. Some people consider this to be quite the hassle and don't want to put in the time and effort to do all of this.

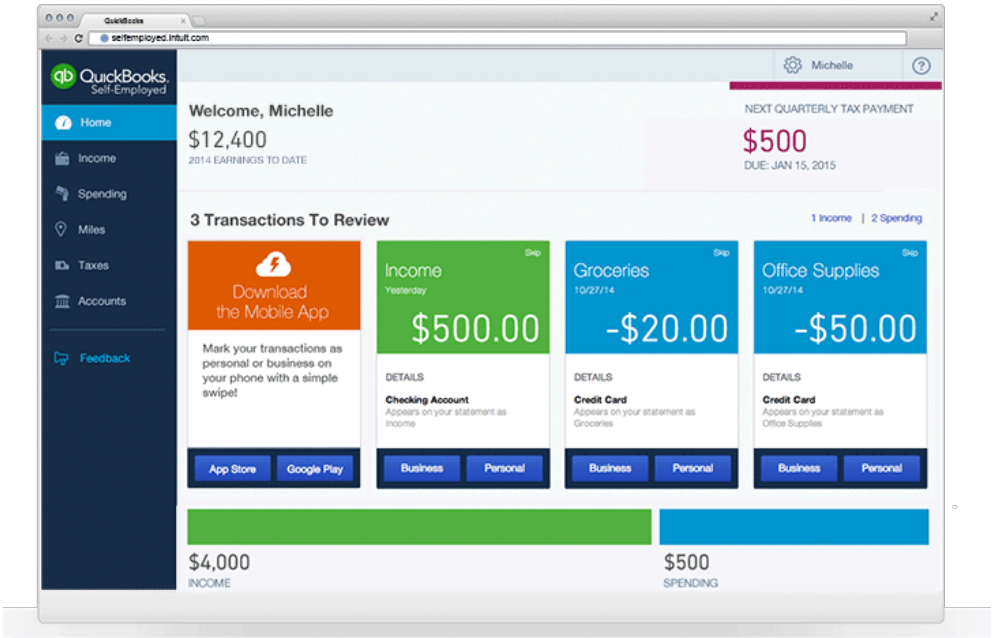
## 1.3 Developing a Solution

Some folks at Intuit recognized this as a problem that a lot of people had to tackle either on their own, or with products that only solved part of their problem. QuickBooks Self-Employed (QBSE) is a new product that seeks to solve the tax problems that self-employed individuals have to deal with. The idea behind QBSE is simple – users supply the service with the information it needs, it does all the calculations, and gives the user everything they need to do their taxes correctly. Of course we want to make the process as easy as possible, so we developed the service as a cross-platform application. Users can use QBSE through the web app, iOS app, or Android app. The mobile apps are currently designed to be "companion" applications to the web app. In other words, users cannot use all of QBSE's features from just the mobile apps. Before we get into requirements of the iOS app and the main topic of this

project, let's take a look at the features that the web app provides to better understand the product.

## 1.4 Web Application Overview

This section will give an overview of what users can accomplish with the QuickBooks Self-Employed web application. Users first sign up for a QBSE account that all of their data will be tied to. Next they connect their bank account(s) to QBSE. By doing so, all of the user's transactions will automatically appear in the app. The user doesn't have to manually enter in an amount whenever they earn some money as a self-employed person. Likewise, the user doesn't have to enter in each expense that would be a self-employed tax deduction. Instead, the app automatically separates the transactions into an income table and an expenses table. The user can simply scroll through all of their income/expenses. These tables are also frequently updated, so users can make a purchase and see their transaction in QBSE (usually) within hours. Additionally, the home screen will show transaction cards that the user can quickly review.



Users can then go through their list of transactions and review them as business or personal. Users should review an income or expense from their self-employed work as business and everything else as personal. QBSE uses this information to determine what needs to be put down on the tax forms. On the Schedule C, self-employed individuals report their business expenses that are tax deductible. However, the form requires that the expenses be broken up into specific categories. QBSE will automatically determine what category the transaction appears to belong in. The user can see this category upon classifying a transaction and change it if desired. When it comes time for the user to fill out their Schedule C, they can look at QBSE's taxes page to fill out the form with appropriate information.

Also on the tax page is the user's estimated tax payments. QBSE uses all the information provided from the user to determine how much they need to pay to the IRS quarterly. It also informs the user of when these payments are due. Once the user is ready to send a tax payment to the IRS, they can click "How to make a payment" to get specific instructions on how to send the payment correctly.

# 2 iOS Application Requirements

We came up with some high-level requirements that the iOS should accomplish. First of all, users need to be able to login to their account. The app coordinates with the QBSE backend for grabbing user data, so credentials are necessary.

The main requirement was giving users the ability to review transactions. This is the essential part of making QBSE work. We need to get the user to input what income/expenses are for business so we can help them with the tax forms and quarterly tax payments.

Another requirement was showing the user's upcoming quarterly tax payment. The goal here is to remind users that the work they are putting in reviewing transactions is actually serving a purpose.

We also wanted to give the user the ability to change transactions from business to personal/personal to business and to change the Schedule C category it belongs to. This replicates the functionality on web. Our auto-categorization feature unfortunately isn't perfect and editing these are sometimes necessary.

The requirement we kept in mind during the entire development process was making the app enjoyable to use. We wanted users to use this on a daily basis. Our theory is that it is easier for a user to spend 30 seconds a day going through their transactions than to wait until it is tax time and do them all at once. We think it'd be easier to remember a transaction from yesterday rather than 3 months ago. We want to promote this behavior because we think it makes the whole self-employment tax process easier.

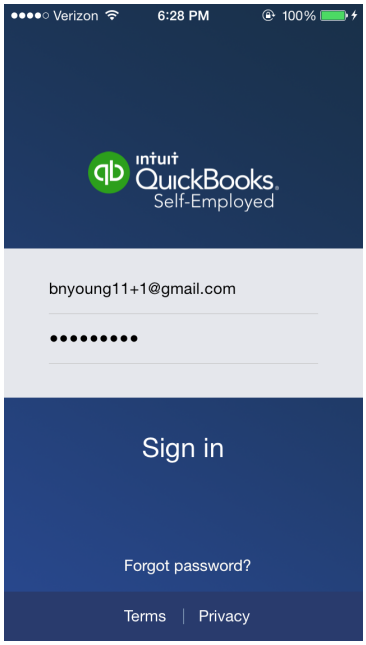
In summary, users need to login to their account, review transactions, see their upcoming tax payments and adjust transaction categorizations – all through an enjoyable user experience that keeps them coming back daily. Next we will take a look at how we planned to satisfy these requirements through our design of the user experience.

# 3 UX Design

In this section we will walkthrough the iOS application, looking at the design of each screen, one by one. We will cover the user experience decisions we made and why we made them.

## 3.1 Login Screen

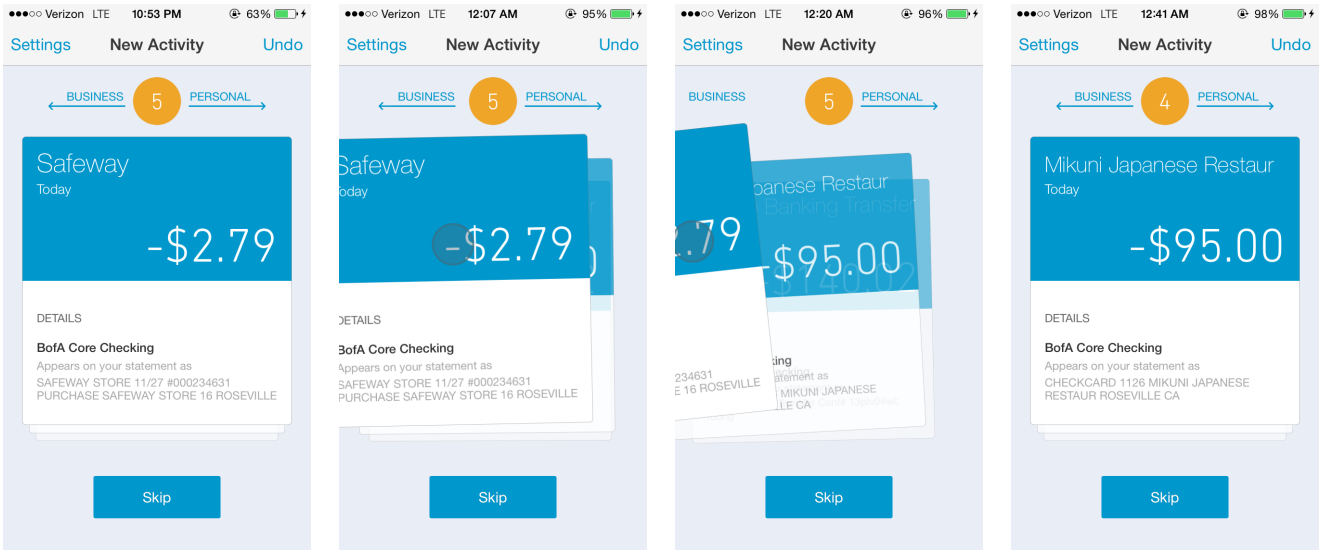
The login screen is the first screen that the user sees when opening the application. Here they supply their QBSE username and password in order to login to the web service. We followed the typical pattern of hiding the password text as the user enters them. The background and color scheme follow Intuit's internal design guidelines. Most of the colors throughout the app follow these guidelines.




## 3.2 New Activity Screen

Once logged in, the user moves on to the main functionality of the app. The purpose of this screen is to give the user the ability to review transactions as business or personal. The screen displays a deck of cards, with each card being a transaction. Above the deck are two arrows pointing away from the deck. The one pointing left is labeled "BUSINESS" and the one pointing right is "PERSONAL". Between the arrows is a little sun (right now just an orange

circle) that holds the number of transactions left in the deck. Below the deck is a skip button. After swiping a card, an undo button appears on the right side of the navigation bar.




To review a transaction as business, the user simply swipes the card to the left. Likewise, swiping a transaction to the right will classify it as personal. After swiping to review a card (or skipping a card), the card underneath is animated “upward” to the top of the deck. If the user is unsure of how a transaction should be reviewed, he/she can press the skip button to send it away to be reviewed later. The user can press the undo button in case they make a mistake and the just-reviewed card will fly back onto the top of the deck.

### 3.2.1 Why a Card UI?

We decided to present the transactions in a card user interface for several reasons. First of all, we didn’t want to present the user with too much information at once. The user is presented with one transaction at a time and is able to focus on determining if that single transaction is for business or not. This way the user is unlikely to overlook a transaction. If we

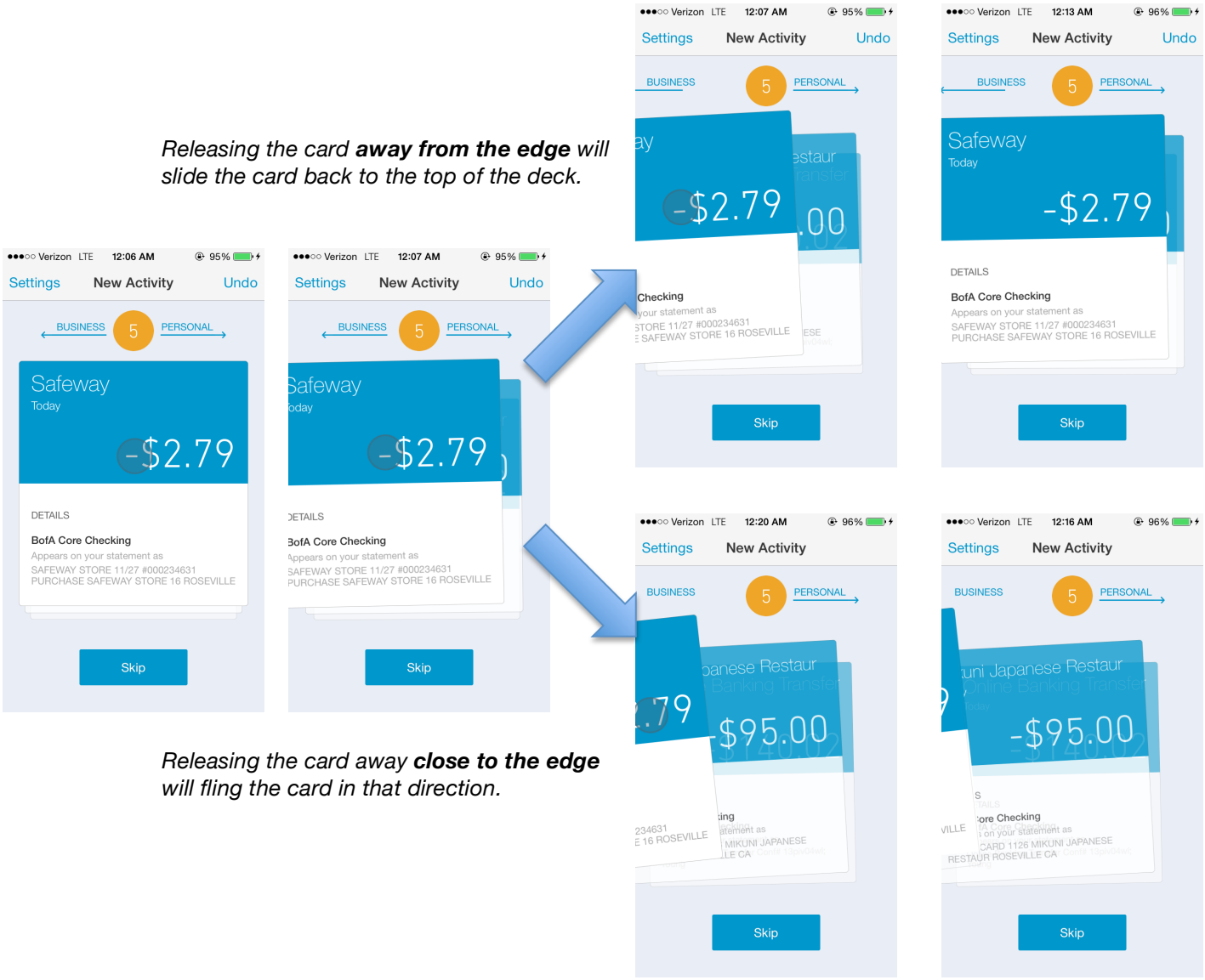
went with a list format for presenting un-reviewed transactions, it would be easier for the user to not see a transaction while scrolling through the list.

Additionally, we are presenting the user with a decision that has two possibilities (business or personal). Having the user decide by swiping the card left or right seemed natural. We also felt it would be a more interactive and fun way to classify the transactions.

### 3.2.2 Essential Card Animations

This screen is filled with animations. First of all, the top card will follow the finger of the user as they drag the card across the screen. This is perhaps the most important animation with the cards. The immediate feedback shows the user that they can directly interact with the card. Releasing the card close enough to the edge will fling the card in that direction. Otherwise, it will slide back to the top of the deck.

The top card can also be flung off the screen by performing a swiping gesture left or right. The gesture motion doesn't need to be completed by the end of the screen either – doing a quick, short swipe will fling the card. Like the panning animations, the swipe animations are used to illustrate the user's direct interaction with the card. This visual feedback shows the user that they successfully reviewed the transaction.




### 3.2.3 Accessory Card Animations

There are some animations we wanted to add in order to give a little extra “feeling” and style to the cards. We added these to help satisfy the requirement that the app should be enjoyable to use. If you look closely, you will notice that the middle card is partially transparent and the bottom card is even more transparent. This is to add a stronger feeling of depth to the deck. After the top card is swiped, the middle card is animated to the top location, the bottom card is animated to the middle, and a new card is animated to the bottom location. However,

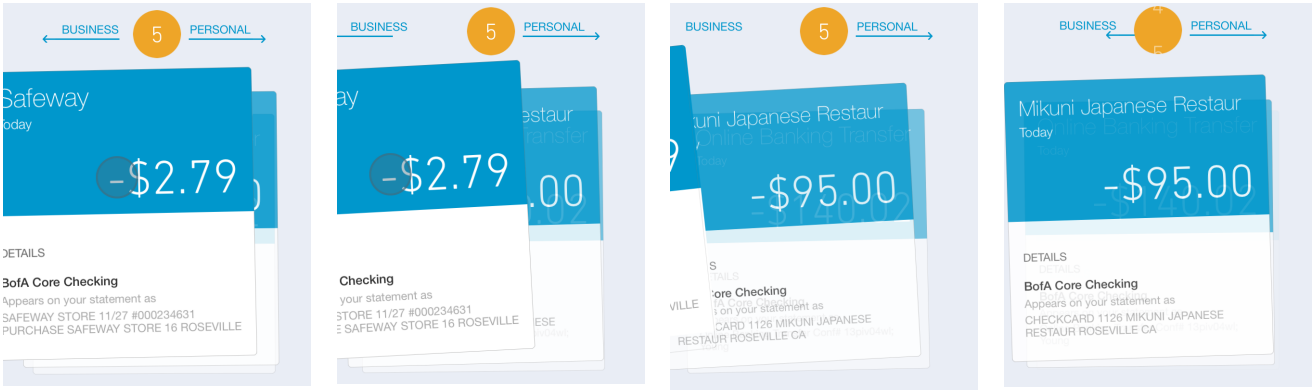
the alpha values are also animated. So as the middle card becomes the top card, it smoothly goes from a semi-transparent look to a completely opaque look.


As the user swipes the top card, the middle and bottom card are slightly pulled in the same direction. Upon release of the top card, the middle and bottom cards bounce back to the middle. This creates a spring-like feeling with the deck. Again, this was added to make the app feel more responsive and enjoyable to use.

### 3.2.4 Review Label Animations

As the top card is panned or swiped, the review labels and arrows will follow the card. So moving the card to the left will also move the business label and it's arrow. If the card is released and it moves back to the top of the deck, the label and arrow will slide back to their original positions. However, when a card is flung off the screen, the arrow will also fling off the screen while the label slides back to the center. After being flung off the screen, the arrow will grow out of the sun counter in the center.

By animating the labels and arrows, we are emphasizing to the user that swiping this transaction is actually reviewing it as either business or personal. The meaning of the swipe follows the card itself so the user can be certain what their action implies.



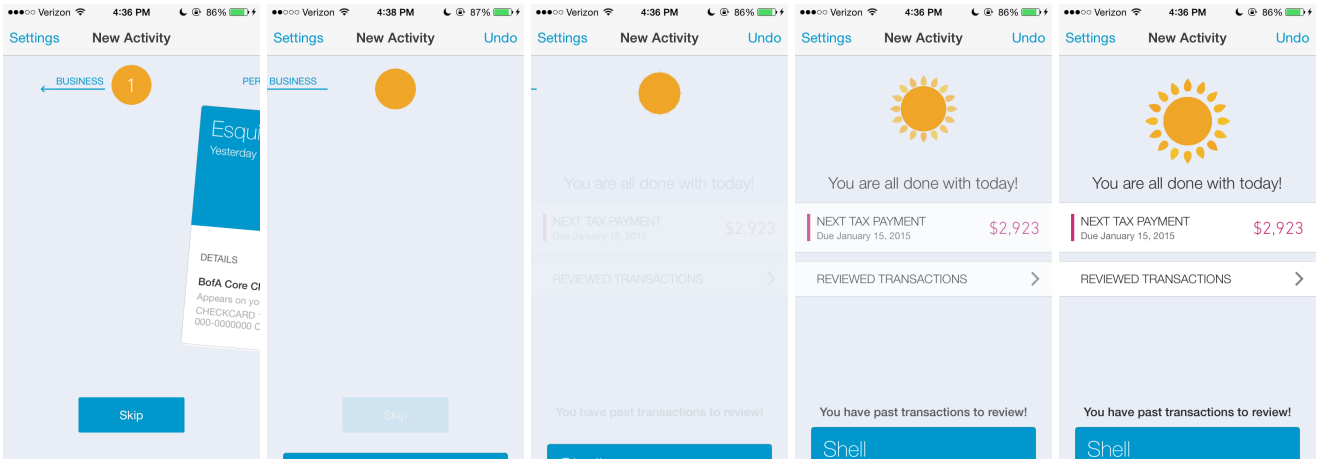

### 3.2.5 Sun Counter Animation

The sun counter displays the number of cards remaining in the deck. Once a card is swiped, the counter is decremented. The next number label slides from above the sun to the center. Meanwhile, the old number label moves from the center to below the sun.

This creates a spin animation similar to slot machines. It's a neat animation that is used to show the user that they are making progress on their work of reviewing transactions.

### 3.2.6 Transition Animations

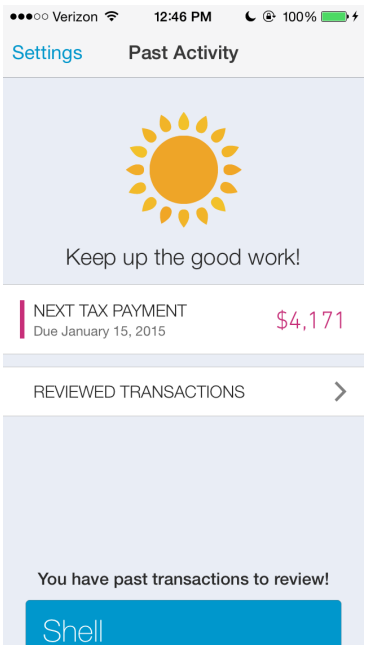
As soon as the last card is swiped, the app transitions from the new activity screen to the home screen. The animations that occur are shown below.



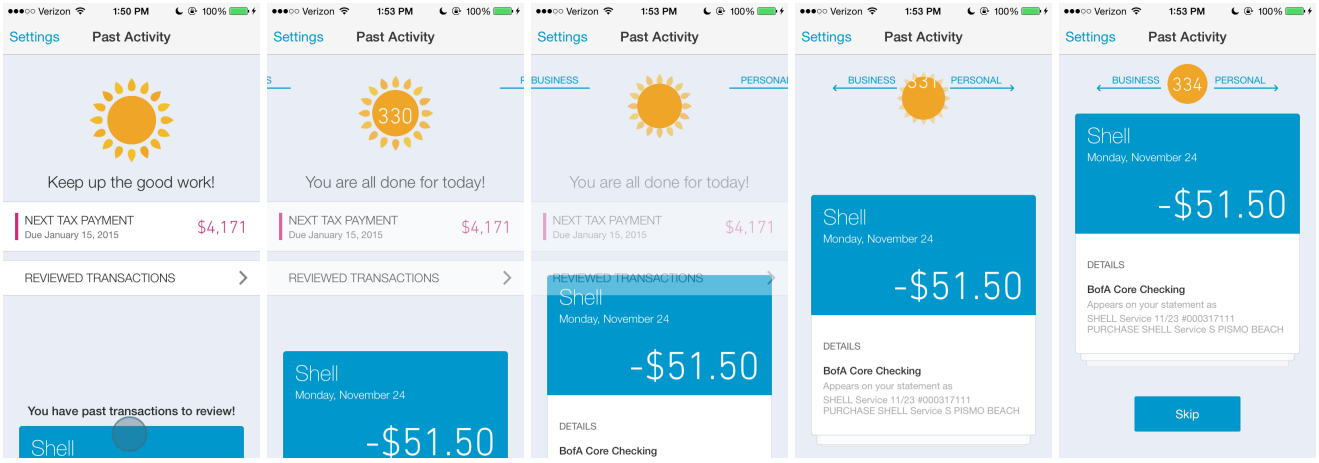
## 3.3 Home Screen

After swiping away all of the transactions from today, the home screen is shown. There is an animating sun image near the top of the screen. Right below the sun is a label with some motivating phrase. Below that is a table (a list). In the table is a row for the user's next quarterly tax payment. The current estimated amount and payment due date are included in this row. If the user has an overdue tax payment, an additional row is shown with that amount. The estimated tax rows are not clickable. Below the estimated tax rows is a row labeled "REVIEW TRANSACTIONS". Clicking this row will show the reviewed transaction table screen.

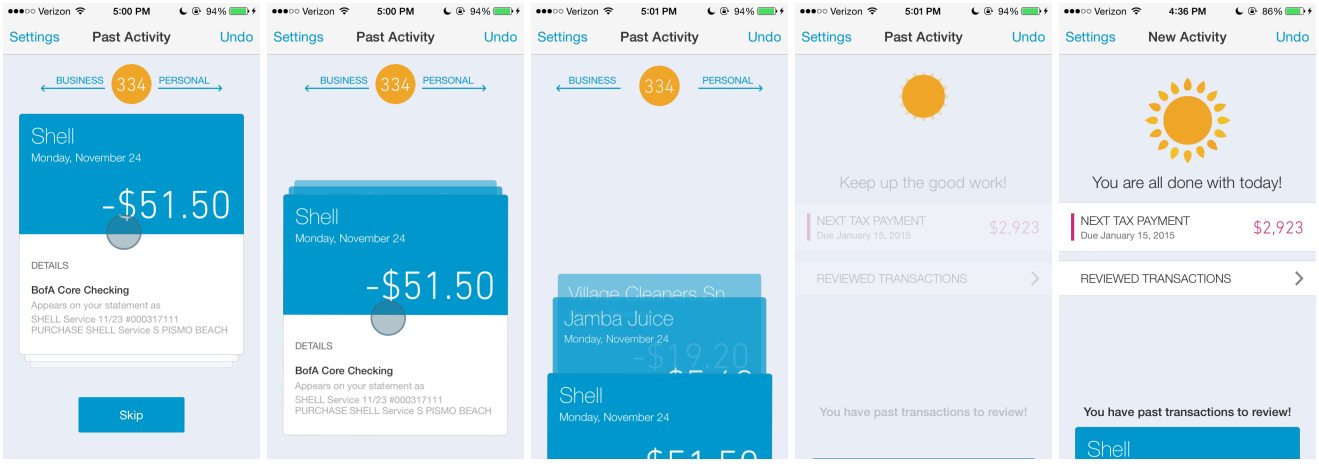
If the user has old transactions that they still need to review, there will be a new deck of cards at the bottom. Right above this deck is a label explaining to the user that they still have some business to take care of. The user can click or swipe up on the deck to bring it onto the screen. They can then swipe these transactions just like before.



Swiping up the past deck will start transition animations from the home screen to the past activity screen. The past activity screen is the same as the new activity – users swipe their transactions. The only difference is that these are old transactions that they haven't gotten around to yet.

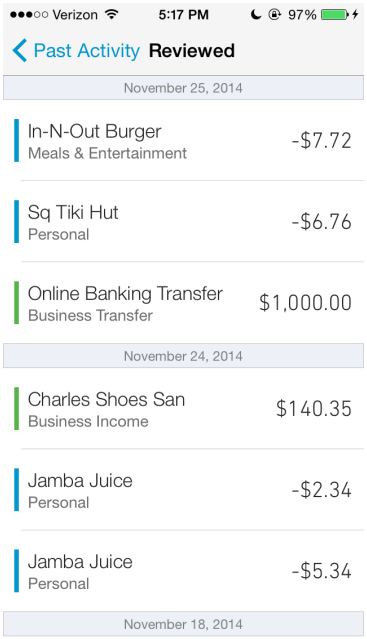



To get back to the home screen, users can pull down or swipe down on the deck. The app will transition back to the home screen.



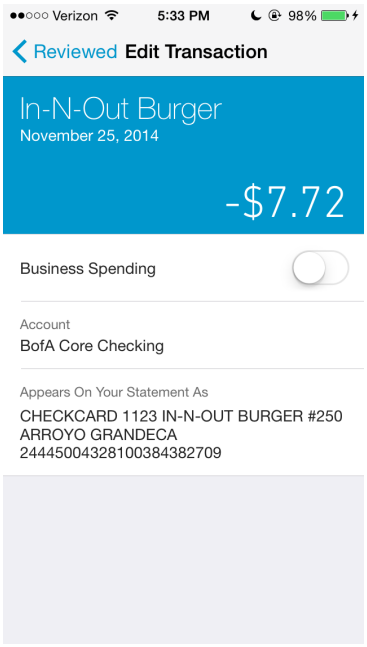
## 3.4 Reviewed Transactions Screen

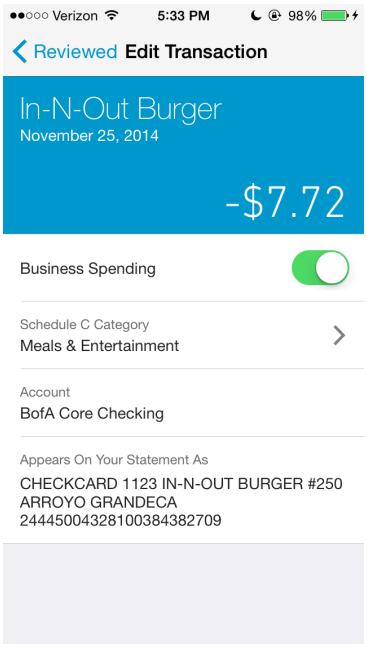
After reviewing a transaction, the user may want to change the category it was assigned by QBSE. Or they may have made a mistake and reviewed a transaction as personal when it should've been business (and vice versa). The reviewed transactions screen simply shows a list of reviewed transactions. Each row shows the title, the amount, and the category. Selecting a transaction brings up the edit transaction screen where the user can adjust the category and if it's for business.



## 3.5 Edit Transaction Screen

Here the user can edit how a transaction is reviewed. Toggling the Business Spending switch is how users can change how the transaction is reviewed. If the switch is in the “on” position, there will be a row with the transaction’s schedule C category. Clicking this row will bring up a screen to select from a list of schedule C categories.



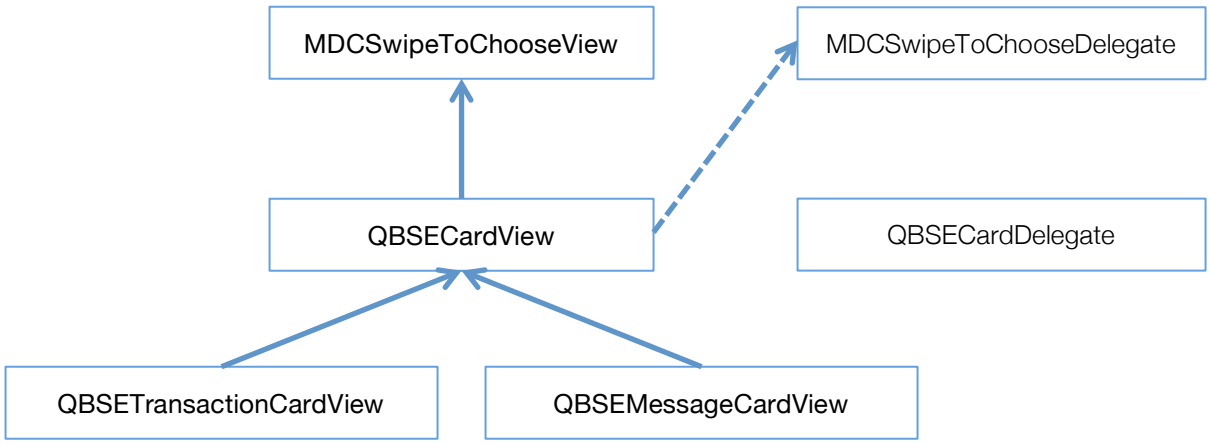



# 4 New Activity Screen Implementation

In this section, we will cover how we implemented the new activity screen. First we will look into some details on how the cards work. Then we will go over how the deck of cards is managed. The review labels' functionality will be described after that. Lastly, we will look at the parent view controller that manages the communication between all of the pieces.

## 4.1 Card Implementation

The card UI provides the main functionality of the app. The following section describes the implementation of the cards themselves. Another class is responsible for managing the deck functionality and we will look into that later. The diagram below illustrates the class hierarchy for the card views. Note that QBSEMessageCardView is only used in the first time user tutorial in the app, which is not a topic in this paper. Therefore, we're not going to look into its implementation details any further.



### 4.1.1 MDCSwipeToChooseView

We utilize an open source library called MDCSwipeToChoose to serve as the base functionality of the card UI. Out of the box, this class handles touch events and animates the panning and swiping accordingly. We modified it to support vertical swiping and adjusted the animations so cards would follow a path instead of moving freely. The actual appearance of the card is not defined here. Instead, the card UI is defined in its super classes.

### 4.1.2 MDCSwipeToChooseDelegate

The methods in this delegate provide the initial interface for responding to card events. We modified some of these methods to provide extra functionality. One example of an event that is called to the delegate is *view:wasChosenWithDirection:.* This method is called whenever a swipe is recognized by the view. It reports the direction that the view was swiped.

### 4.1.3 QBSECardView

This class extends MDCSwipeToChooseView to inherit the touch event handling and card animations. Here the background of the card is set. This includes the grey border of the cards. QBSECardView implements MDCSwipeToChooseDelegate and sets itself as the delegate property from its super class. So here is where we respond to all of the card events that MDCSwipeToChooseView calls. However, we still want to respond to these events outside of the card view. QBSECardView has a QBSECardDelegate property that it will relay the messages to. By relaying messages to a whole new delegate, other classes that interact with the cards will not have to know about the library that the card view utilizes. This way we can drop the reliance on the library later and the only changes will be in QBSECardView.

### 4.1.4 QBSECardDelegate

These methods are used to relay messages about what events are occurring with the card. This is the interface that the rest of the app uses to listen for card events. Here's the method that is called when the card is swiped:

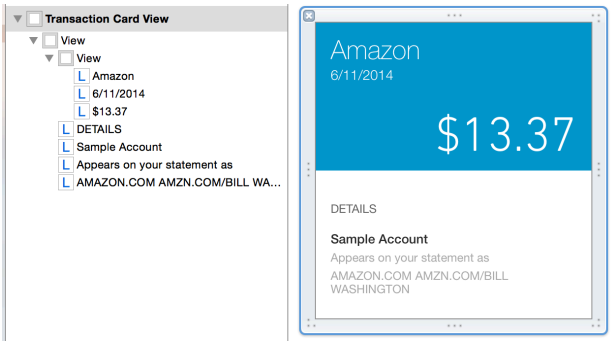
```
/**
 *  The card just registered a swipe and animated off the screen. This will only be
 *  called if card:willBeSwipedInDirection:translation: returns 0.
 *
 *  @param card      The card that was swiped
 *  @param direction The direction the card was swiped in
 */
- (void)card:(QBSECardView *)card wasSwipedInDirection:(QBSECardDirection)direction;
```

The following shows the method being called in QBSECardView. The QBSECardView object receives the *view:wasChosenWithDirection:* message, converts the passed in MDCSwipeDirection enum value to a QBSECardDirection enum, and calls the method.

```
- (void)view:(UIView *)view wasChosenWithDirection:(MDCSwipeDirection)direction
{
  if (_delegate)
     [_delegate card:self wasSwipedInDirection:QBSECardDirectionFromMDCDirection(direction)];
}
```

### 4.1.5 QBSETransactionCardView

The UI that is specific to transaction cards is created here. All of its labels are set to the appropriate text from the QBSETransaction model that is supplied in its initializer. Most of the UI is created, sized and arranged from a nib file shown here:




## 4.2 Deck Implementation

QBSETransactionCardView handles its own touch events, animates in response to those touch events, and constructs its own interface. QBESDeckViewController is responsible for managing the QBSECardViews that make up the deck of cards.

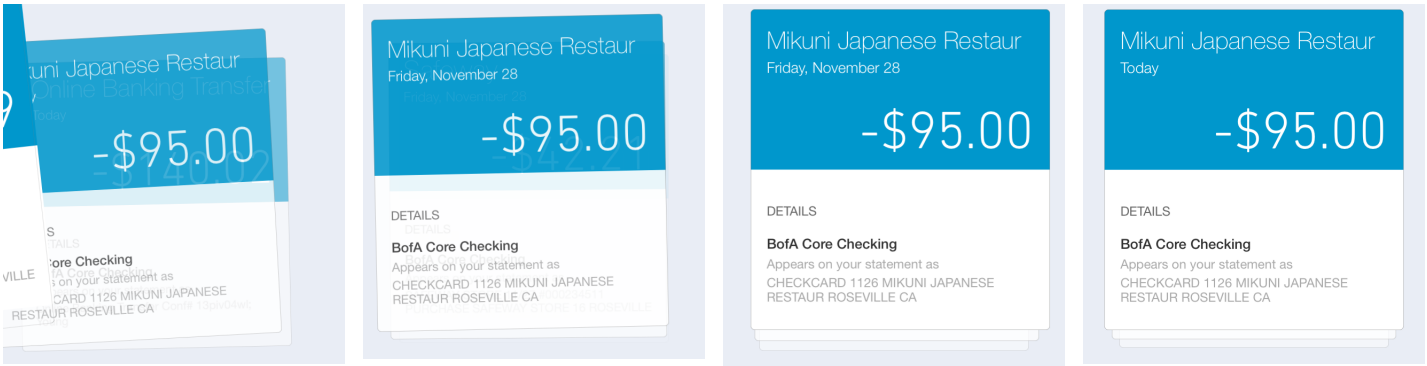
### 4.2.1 QBSEDeckViewController

QBSEDeckViewController handles all of the deck animations and positions the cards appropriately. The deck can be in either an expanded state (deck is in the middle of the screen and the user can swipe cards) or a collapsed state (deck is on the bottom of the screen and the user can swipe up to bring them on screen). This controller will make sure the cards are in the correct position and restricts some touch events depending on the state of the deck (e.g. users can't swipe a card left or right while the deck is in a collapsed state).

QBSEDeckViewController implements the QBSECardDelegate protocol because it needs to respond to the card events. It also has a QBSEDeckDelegate property that is set in its initializer. QBSEDeckDelegate is a protocol that implements QBSECardDelegate and adds a method to let the delegate know that the deck was pulled down to the collapsed state. It relays the card events to this object, but not always – it will intercept some if the touch event shouldn't be permitted. For example, QBSECardView will call *card:willPanInDirection:* as soon as the card is being panned left or right (the user put their finger on the card and began to move it). If the deck is in a collapsed state, the user shouldn't be able to pan the card. By returning NO, QBSEDeckViewController can tell QBSECardView that a pan should not occur. Additionally, the deck will only call the *card:willPanInDirection:* on its own delegate property if its in an expanded state. The code below shows how this works.

```
- (BOOL)card:(QBSECardView *)card willPanInDirection:(QBSECardDirection)direction
{
    // only allow a pan when the deck is shown on screen.
    if(!_isCollapsed)
    {
        if ([_delegate respondsToSelector:@selector(card:willPanInDirection:)])
            return [_delegate card:card willPanInDirection:direction];
        return YES;
    }
    return NO;
}
```

This class also contains public methods so a parent view controller can interact with the deck. For example, *addCardToBottom:* is used to add a card to the bottom of the deck. When this is called, the deck animates the middle card to the top position, the bottom card to the middle position, and the new card to the bottom position. This occurs after a swipe. As shown below, the middle and bottom card are pulled slightly with the swipe in the swipe's direction. Then they slightly "overshoot" where they should end up (3rd image from the right) and settle to their correct positions.




The code that makes these animations happen is below. Note that *topTransform* and *middleTransform* are the objects that define the position and size of the cards. The view animations are handled using the UIView class method *animateWithDuration:delay:options:animations:completion:*. Changing specific properties of a UIView in the *animations* block and specifying duration for these properties to change is all it takes to make an animation happen.

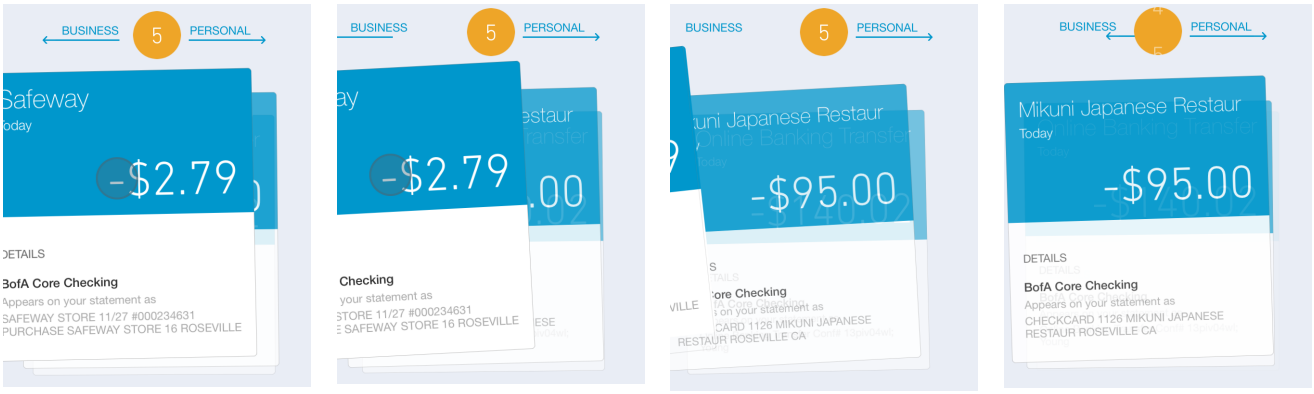
```
- (void)animateCardsMovingUpFromDirection:(QBSECardDirection)direction completion:(void (^)(BOOL
finished))completion
{
    // Card was just swiped left or right. We need to animate the cards from their current position
    // (left or right of center) to their new positions (bottom becomes middle, middle becomes top).
    // But we want to overshoot a little and come back.

    static CGAffineTransform leftMiddleOvershootTransform, leftBottomOvershootTransform;
    static CGAffineTransform rightMiddleOvershootTransform, rightBottomOvershootTransform;
    if (leftMiddleOvershootTransform.tx == 0)
    {
        const CGFloat overshootHorizontalDistance = 4, overshootVerticalDistance = -2;
        leftMiddleOvershootTransform = CGAffineTransformTranslate(_topTransform,
                                                                  overshootHorizontalDistance,
                                                                  overshootVerticalDistance);
        leftBottomOvershootTransform = CGAffineTransformTranslate(_middleTransform,
                                                                  overshootHorizontalDistance * .5f,
                                                                  overshootVerticalDistance * .5f);
        rightMiddleOvershootTransform = CGAffineTransformTranslate(_topTransform,
                                                                   -overshootHorizontalDistance,
                                                                   overshootVerticalDistance);
        rightBottomOvershootTransform = CGAffineTransformTranslate(_middleTransform,
                                                                   -overshootHorizontalDistance * .5f,
                                                                   overshootVerticalDistance * .5f);
    }

    CGAffineTransform middleOvershootTransform, bottomOvershootTransform;
    switch (direction) {
        case QBSECardDirectionLeft:
            middleOvershootTransform = leftMiddleOvershootTransform;
            bottomOvershootTransform = leftBottomOvershootTransform;
            break;
        case QBSECardDirectionRight:
            middleOvershootTransform = rightMiddleOvershootTransform;
            bottomOvershootTransform = rightBottomOvershootTransform;
            break;
        case QBSECardDirectionDown:
        case QBSECardDirectionNone:
        case QBSECardDirectionUp:
            middleOvershootTransform = _topTransform;
            bottomOvershootTransform = _bottomTransform;
            break;
    }

    [UIView animateWithDuration:0.15f
                          delay:0
                        options:UIViewAnimationOptionCurveEaseOut |
                                UIViewAnimationOptionBeginFromCurrentState |
                                UIViewAnimationOptionAllowUserInteraction
                     animations:^{
                         if (_topCard) {
                             _topCard.alpha = 1;
                             _topCard.transform = middleOvershootTransform;
                         }
                         if (_middleCard) {
                             _middleCard.alpha = MIDDLE_CARD_ALPHA;
                             _middleCard.transform = bottomOvershootTransform;
                         }
                     }
                     completion:^(BOOL finished) {
                         if (!finished) return;
                         [UIView animateWithDuration:0.2f
                                               delay:0
                                             options:UIViewAnimationOptionCurveEaseInOut |
                                                     UIViewAnimationOptionAllowUserInteraction
                                          animations:^{
                                              if (_topCard)
                                                  _topCard.transform = _topTransform;
                                              if (_middleCard)
                                                  _middleCard.transform = _middleTransform;
                                          }
                                          completion:completion];

                     }];
}
```

## 4.3 Review Label Implementation

The review labels and arrows instruct the user which direction is which when swiping. They animate to follow the path that the cards take as the cards are swiped. The QBSEReviewLabelViewController class manages the labels and arrows.

### 4.3.1 QBSEReviewLabelViewController

The review labels correspond with the card's movement. As the card is panned left, the label and arrow should also pan. When the card reaches the edge of the screen, the label remains completely visible while the arrow follows off the screen. On a swipe, the arrow goes completely off the screen then "grows" back out of the sun to its original position.

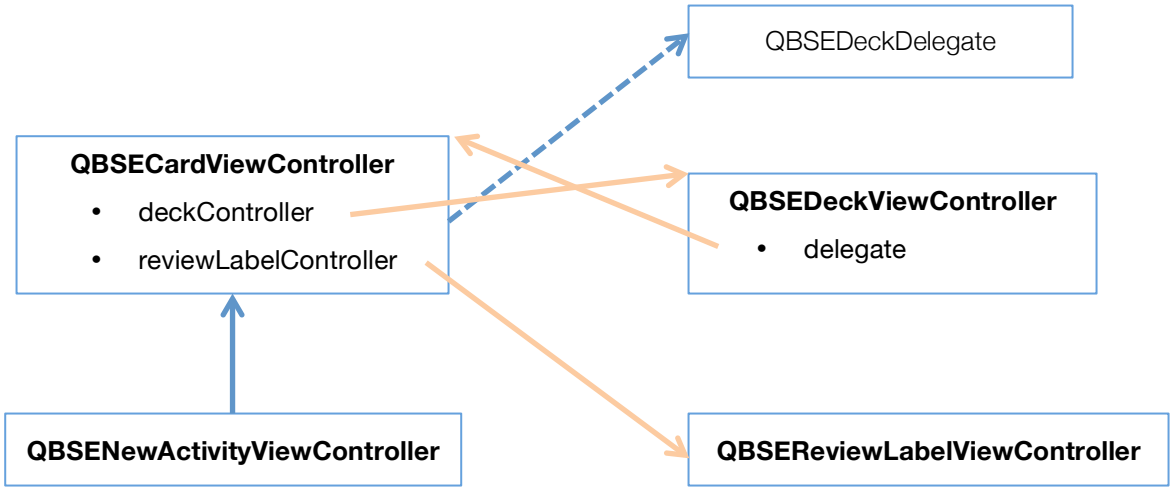



QBSEReviewLabelViewController doesn't implement any protocols, it just has public methods so some other class can tell it what to do. One such method is *notifyLeftMovementWithDelta:*. The caller passes in the horizontal distance that the card has moved from its original position. This method will then move the left label and arrow accordingly. Note that the x position for the label is the max between the left side of the screen (minCenterX) and the corresponding position of the card. It's the maximum because the x values are decreasing as the label moves left.

```
- (void)notifyLeftMovementWithDelta:(CGFloat)delta
{
    if (isInHiddenState) return;

    // move the left label the same corresponding distance as the reported delta
    if (delta > minLeftLabelDelta) {
        CGFloat xPos = MAX(minCenterX, initLeftLabelCenter.x - delta + minLeftLabelDelta);
        _leftLabel.center = CGPointMake(xPos, _leftLabel.center.y);
    } else {
        _leftLabel.center = initLeftLabelCenter;
    }

    // move the left arrow too
    if (delta > minLeftArrowDelta) {
        _leftArrow.center = CGPointMake(initLeftArrowCenter.x - delta + minLeftArrowDelta,
                                        _leftArrow.center.y);
    } else {
        _leftArrow.center = initLeftArrowCenter;
    }
}
```

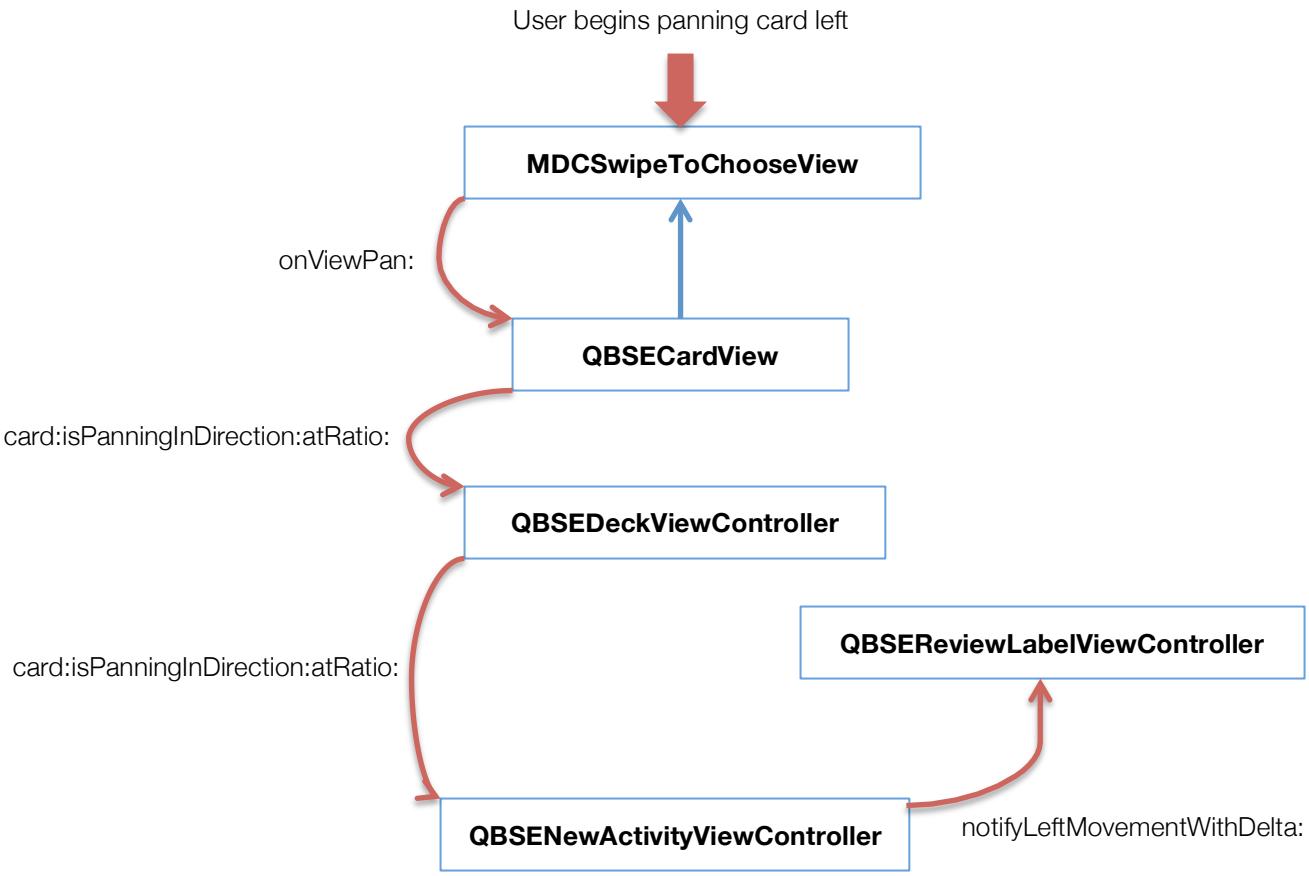
## 4.4 Putting It All Together

QBSENewActivityViewController is the parent view controller that positions and sizes instances of QBSEDeckViewController and QBSEReviewLabelViewController. It also acts as the communicator between them. However, some of the communication is handled by the class it extends QBSECardViewController. The following diagram illustrates all of their relationships:



Let's take a look at how the communication occurs when the card is being panned to the left:

1. An instance of QBSECardView detects a pan occurring. It calls *onViewPan:atRatio:* on the MDCSwipeToChooseDelegate property (from MDCSwipeToChooseView inheritance), which happens to be it self.
2. This same instance receives the message that a pan left is occurring. It then tells its QBSECardDelegate property by calling *card:isPanningInDirection:atRatio:* on it.
3. An instance of QBSEDeckViewController is the QBSECardView's delegate property. It receives the event that the card is panning left. It does some animations in response and tells its QBSEDeckDelegate property that a pan is occurring by calling *card:isPanningInDirection:atRatio:* on it.
4. An instance of QBSENewActivityViewController is the QBSEDeckViewController's delegate property. It receives the event that the card is panning left. It tells its QBSEReviewLabelViewController property what's going on by calling *notifyLeftMovementWithDelta:*.
5. The instance of QBSEReviewLabelViewController that is QBSENewActivityViewController's reviewLabelController property animates its left label and arrow.




QBSENewActivityViewController is also responsible for the communication and management of QBSESunViewController (the sun counter that also animates on the home screen), QBSEPromptViewController (manages the content of the home screen, including the table), and QBSEOverlayViewController (the loading screen and error screen which are not covered in this paper). The communication with each of these pieces is very similar to what was just demonstrated.

# 5 Retrospective

This project wasn't an easy task by any means. One of the biggest hurdles I had to overcome was learning Objective-C and how to develop for iOS. I had to read a ton of documentation on how to build every little component of the UI. Reading blogs and articles online helped with the process. But they only helped so much because Apple is constantly updating the frameworks for iOS – many posts discussed outdated ways to develop for the system. Additionally, the updates that Apple make on the iOS platform sometimes make it difficult to develop for. For example, UIViews in iOS can now be sized and positioned using the Auto-Layout system. It was more work to learn how to make this system operate the way I wanted it to. But I'd definitely say it was worth it because it makes the app more responsive to various screen sizes and Apple is continuously releasing different sized products.

It also took quite a bit of trial and error for me to figure out what architectural decisions work well. I had to keep in mind that this is an on-going project so writing stable code that is easy to maintain was a huge priority. Additionally, more people will join the project as time goes on – the code needed to be as readable as possible. These requirements resulted in me doing a few rounds of refactoring during the development process. But as the project went on, it became easier to make good decisions that took these into account.

The way we decided to design the user experience imposed challenges on the implementation side too. We didn't want users to finish swiping cards and a new screen to simply slide in. We wanted each individual element to be a part of the transition so everything would be smooth. This required carefully compartmentalizing each component (deck is it's own component, sun is its own, review labels, etc.). Luckily, developing the architecture in this manner went hand in hand with making the code maintainable and readable.

One other issue I ran into was getting the second version of the app released to the store. We released the first version at the end of August. Another developer I worked with on the app handled the submission process. However, I was the only one still working on the iOS app in the fall – when it became time to submit the second version, I took care of it. Figuring out how to sign the archive appropriately took some time. Also preparing the image assets (screenshots) and app descriptions for the App Store was tedious. Seeing it released on the store made all the work that much better though!

# 6 Conclusion

We were able to build an app from the ground up in a decently short amount of time. So I'd definitely consider it to be a successful project. As of now, the App Store has our second version of the app (version 1.5).

I've really enjoyed working on this project. I was able to learn a ton about iOS development, design, building a product, and working with a team in industry. I will continue to learn from this project, as it will continue for me. I will keep on working with the QuickBooks Self-Employed team. I'm excited to continue this development process to try and make this product better and better.