Table of Contents

EatRight Final Documentation

Section 1 : Project Description

The statistics of weight gain and weight-related health issues in the United States show a growing dietary epidemic, particularly with the youth who are more likely to be tempted into the abounding fast food restaurants populating American street corners. Fast food restaurants are convenient and cheap - both desirable qualities in food-finding endeavors of busy students and adults - but for those who want to make an effort to stay healthy, it can be difficult for them to make informed choices at these establishments. When you enter a restaurant, there's a pressure to make a choice quickly, but the menus are small and evaluating trade-offs of food choices takes significant effort, so our project was to write an Android application that would allow a user to choose meal items from fast food restaurants in the area and visually display them in comparison to other meal items in terms of "healthiness" on a graphical, color-coded chart. Our client for this project was Professor Nakamura, from Food Science and Human Nutrition and it is under his supervision that we worked to complete the requirements he set for such an application.

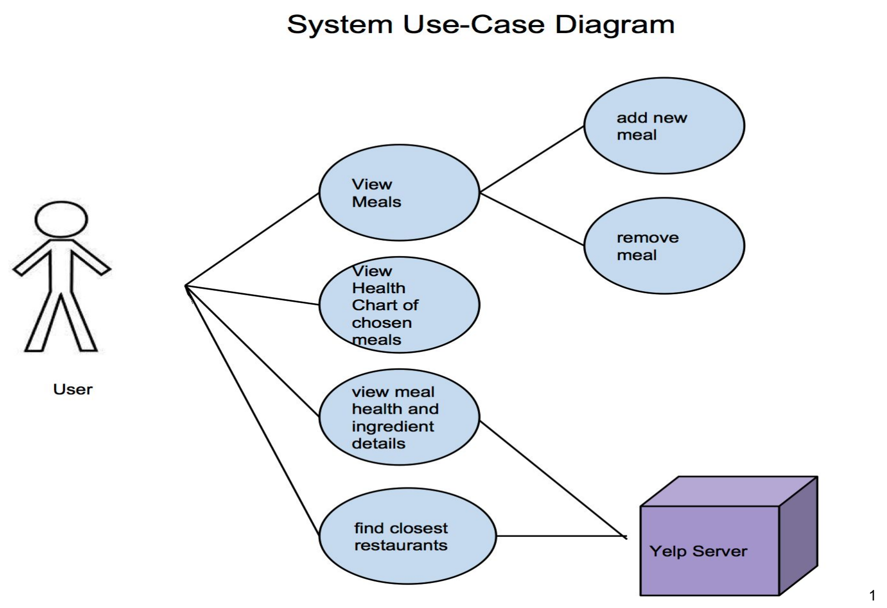
Section 2: XP Process

The software process that our team followed was the XP (extreme programming) software development process that we learned about from CS 427. It is a software development methodology that intends to improve software quality and responses to changing customer requirements. This is a discipline of software development based on values of simplicity, communication and feedback. The XP brand of software development has code released frequently per iteration and in very small development cycles. This has the advantage in that it allows there to be more readiness to adapt to any given situation if for some reason the requirements of the project change.

The XP process is a software development process that focuses on test driven development and the software process is one where there are a certain set of roles that the members of the group cycle through the different iterations. As a part of this particular software development cycle, at every iteration meeting for the group, the members of the group are assigned four different MARS roles, which are moderator, author, reviewer and scribe. Along with these four roles, for the duration of the iteration, three additional roles of planning leader, development leader and testing leader are also assigned. These steps are very essential for the success of the group because as a consequence of this setup, the members of the group can be held accountable for their work. The XP process places emphasis on the teams unified knowledge of the code base as well as following of consistent coding standards so that everyone can read everyone’s code, and keep the product system up and running at all times. This leads to everyone practicing coding in a previously agreed upon consistent style so that anyone can understand and implement any change in the code as needed. Extreme programming differs from other software development processes in that it always has a timeline-based sequence of software releases. This means that in each iteration cycle one or more user cases are handled and code is pushed. Thus, this places a higher value on the adaptability of the project to events. Therefore, a very natural aspect of the XP software development process is the ability to handle changes to requirements effectively by being able to adapt to them compared to some of the more traditional methodologies that are not able to respond to the same extent. The XP software development process also allows the software developers to often find out about possible edge cases and points of failure since it is a very test driven development process that forces the developers to think about what the product is intended to do which can further be tested. XP also has the added benefit of pair programming and thus reaching a more robust solution to the project faster than otherwise would have been the case. Finally XP teams focus on building simple software with adequate design, through rigorous testing and design improvement. This allows the software team to keep the design ready for the current functionality of the system.

Section 3: Requirements and Specifications

The requirements for this project are captured in the use case diagrams for Project EatRight. (Using the use case diagrams from HW 2 and HW3). We initially built the use cases for this project to help us plan out the work that would be required for us to complete the application. They are listed as follows: 1) User can view meals and restaurants, 2) User can view health chart of chosen meals, 3) User can view meal health and ingredients details, 4) User can find the closest restaurants from the given user location, 5) User can customize preferences, 6) User can add new meals, 7) User can remove meals, 8) User can combine meals into menu, 9) User can plot selected meals into a chart and finally 10) User can view a summary screen of all meals that have been added to see the visualization. Some of the use cases and user stories were refined and changed during the course of this software engineering project. The division of the project into these different use cases helped us split up the work by the main three categories of visualization, API, Locu and parsing and finally the data group.

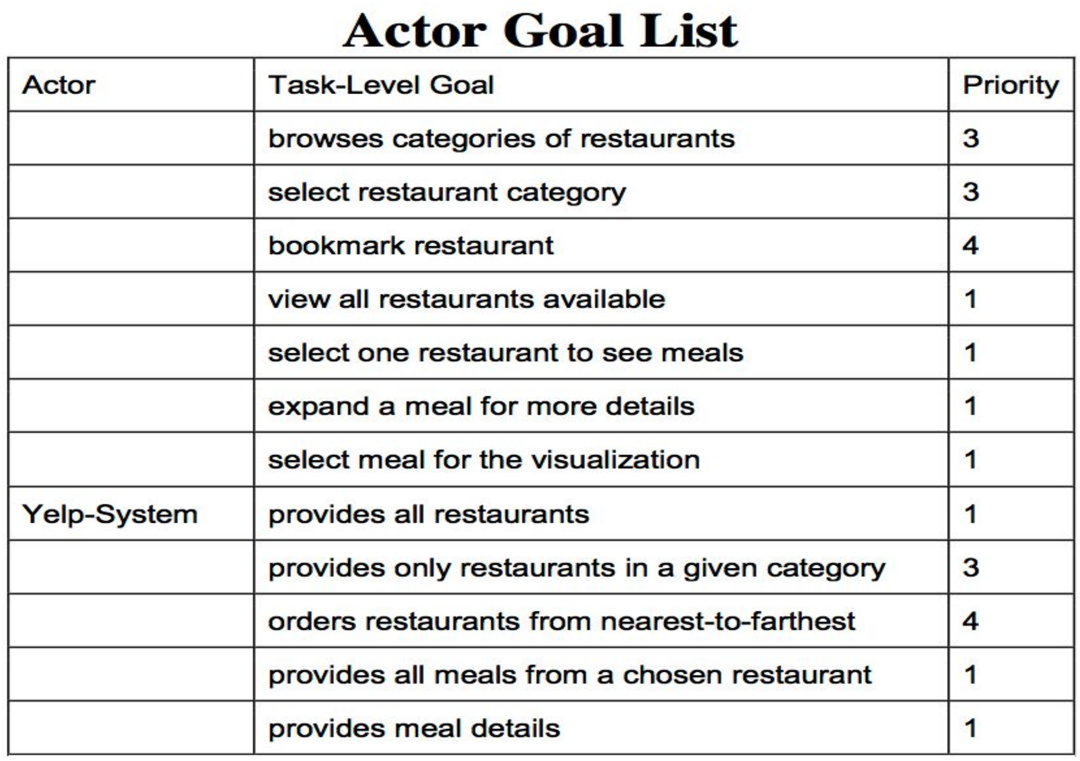


[1]

Also the specifications of the project, required us to build an actor-goal list. For our project we had two main actors which were the user of the app (hungry person) and also the Yelp-System. The first actor had various goals which were the task level goals: 1) browsing restaurant categories, 2) select restaurant category, 3) bookmark restaurant, 4) view all available restaurants, 5) select one restaurant to see meals, 6) expand a meal for more details, 7) select a meal for visualization, 8) input dietary preference and allergies, 9) apply preferences to filter.

Similarly for the Yelp level system we had task level goals that were: 1) provides all restaurants, 2) provides only restaurants in a given category, 3) orders restaurants from nearest to farthest, 4) provides all meals from a chosen restaurant, 5) gives meal details, and also 6) provides filtered meals based on user preferences that have been previously specified.

[1] Figure 1 - System Use case Diagram



Actor Goal List

Actor	Task-Level Goal	Priority
	browses categories of restaurants	3
	select restaurant category	3
	bookmark restaurant	4
	view all restaurants available	1
	select one restaurant to see meals	1
	expand a meal for more details	1
	select meal for the visualization	1
Yelp-System	provides all restaurants	1
	provides only restaurants in a given category	3
	orders restaurants from nearest-to-farthest	4
	provides all meals from a chosen restaurant	1
	provides meal details	1

[2]

The next step that we took in the project after writing down the use cases and user stories was to split them up into the different iterations. This was done by ranking the use cases based on the estimated time to complete a particular case and also by dividing the use cases into the three groups of data, visualization and parsing (API's Yelp and Wolfram). These are all visible and easy to access through the different child pages in the Project EatRight homepage.

Section 4: System Environment

There are many ways to write an Android application, but by far the most efficient way is to use the Android Studio platform. This platform allows for quick setup of UI screens and better management of resources, xml screens, emulation, testing, and content assists. Android Studio makes the process of designing Android applications easier. The application is free to use and simply required downloading [1]. There is a nice feature in Android studio that allows the programmer to link the application directly to version control, such as GitHub. Any new updates can be merged and any new changes can be pushed to the repository without ever leaving the studio.

There is an emulator available in Android Studio that allows for testing the application as if it were on a real device. The programmer is able to choose which version of the Android operating system they wish to test on, which generally is decided by the API level of the program they are writing. New APIs are not available on the older phones, the features are not portable, so the general strategy is to write programs for the lowest API level as the features you need to implement.

[2] Figure 2 - Actor Goal List

To make the emulator work, in BIOS, the Intel virtualization technology must be turned on. This can be done at startup of the machine the programmer is working on. Also, SDKs need to be downloaded and the HAX which will make the native Android elements and the simulator available to the programmer, respectively. The environment setup is time consuming. It is best to follow a tutorial and may take a few days for a newcomer to get working correctly. Once the environment is setup, the programmer can begin. The process for writing an application is generally as follows.

First, you create `Activity` classes, which are typically UI screens. These typically contain the control flow of an application, dictate how an end user will move to other screens through buttons and what data will be passed between screens with `Intents`. An `Activity` will also contain methods that dictate what will occur when the activity is first created or resumed or where a back button will lead. These methods are aptly named `onCreate()`, `onResume()`, and `onBackPressed()`.

Once you have an Activity, it is then associated with a view, generally some form of layout that will then be exposed to the end user. Most often, the view will be an xml file in the resources folder of the Android project and the Activity will define how the objects in the view act on user inputs, be they key presses, button clicks, or touch swipes. These objects are accessed via IDs that are defined in the xml file. The xml files are static.

The number of elements in an xml file does not change so to create buttons dynamically it is usually best to create the view within the activity. However, some native Android views are built to work dynamically, such as the `ListView` and `ExpandableListView`. To map data to these dynamic views, adapter classes need to be written. The adapters define how to find top level items and children of those items. There is no native tree view in Android, thus to create a three level `ListView` there needs to be an `ExpandableListView` inside an `ExpandableListView` and a second adapter class. This can be tricky to create and it is not obvious why, but a custom `ExpandableListView` needs to be created and used in the second level `ExpandableListView`. This custom class will have to extend `ExpandableListView` and have to override the `onMeasure()` method with its own measurements. The `ListView`s will not work without this custom class. The second level will not even appear without it. Android applications all have an `AndroidManifest` as an xml file. In these files, all the permissions associated with the application must be identified. This may include internet permissions or location services. Also, all activities must be logged in the manifest and the first screen must be identified so that the application knows where to start. Once the manifest is up-to-date the programmer can run their application on the emulator. It does have limitations. It cannot simulate GPS services or Bluetooth, for instance, but it does simulate internet services and API calls. The emulator can be used to manually test parts of the application.

One aspect of android, and mobile development in general, that is unique is automated UI testing. A lot of people are not familiar with this because most software that we have written for other classes and projects are not user-facing and therefore do not need the additional user interface testing that is required for mobile development. Android Apps do not have this luxury though and in order to ensure the interface works properly it must be tested.

Traditionally there are two ways to do the user testing. One way which is the arguably the easiest (but not very efficient) way to do it is to simply load the application onto a physical device and have someone use it and navigate through the options until something breaks. The user would then record what steps they took to get to the error and then submit this to the developers so then can inspect the issue and make changes. This way is not very efficient and does not allow for automated testing which will slow down the development process Android and Android Studio provide some tools to streamline this process and allow a developer to create and run automated user testing. These tools were extremely necessary to developing our UI tests and ensuring that the app worked as expected.

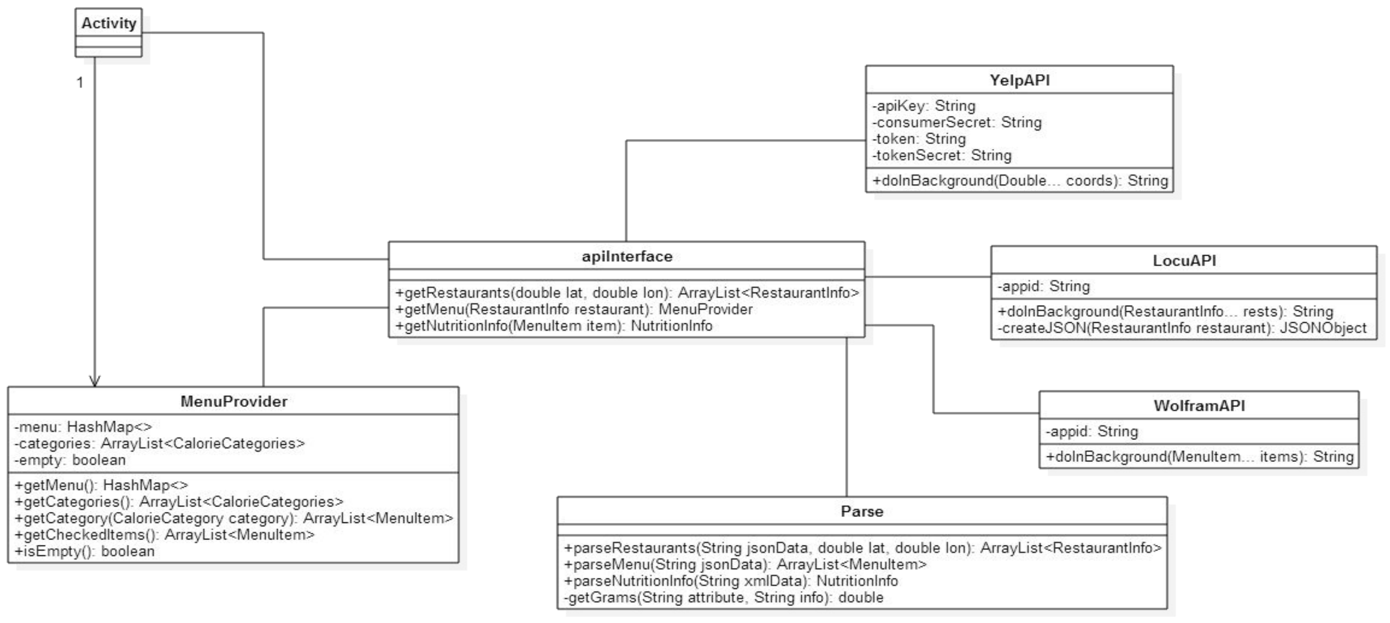
The first tool that Android Studio provides to assist with UI testing is the UI AUtomator Viewer. The viewer tool inspects a screen on that is running on a device and will show all of the visual elements of that screen. You can use this tool to inspect the layout hierarchy and view the properties of UI components that are visible on the foreground of the device. This will give the developer a way to call specific UI elements regardless of the way those elements are generated or how they interact with the underlying backend code. Using these UI elements and their names one can call specific functions in the UI testing framework to click and interact with the app.

Once you have inspected the elements you wanted to test, you can begin writing the automated UI tests. If you are developing in Android Studio, when you created the project a folder where tests should be stored has been created. To locate this folder change the viewing mode from Android to Project and the folder can be found in the tree at “app -> src -> androidTest -> java -> [projectName]”. Once inside of this folder right click and go to new -> Java Class, name the class and import the proper UI packages. At this step the testing is much like JUnit testing with a different set of functions that are used. Please refer to the link for a tutorial from Android which explains in detail how to write these tests effectively and links to the UI Automator API [2].

Section 5: Architecture and Design

Section 5.1 Data Classes and API

Java classes were created to store the information that gets requested and parsed from the API calls. One was created to hold restaurant information with name and address fields. The other class was the Menuitem class which hold the name of a meal and its nutritional information.


Activity
1
YelpAPI
-apiKey: String
-consumerSecret: String
-token: String
-tokenSecret: String
+doInBackground(Double... coords): String
apiInterface
+getRestaurants(double lat, double lon): ArrayList<RestaurantInfo>
+getMenu(RestaurantInfo restaurant): MenuProvider
+getNutritionInfo(MenuItem item): NutritionInfo
LocuAPI
-appid: String
+doInBackground(RestaurantInfo... rests): String
-createJSON(RestaurantInfo restaurant): JSONObject
MenuProvider
-menu: HashMap<>
-categories: ArrayList<CalorieCategories>
-empty: boolean
+getMenu(): HashMap<>
+getCategories(): ArrayList<CalorieCategories>
+getCategory(CalorieCategory category): ArrayList<MenuItem>
+getCheckedItems(): ArrayList<MenuItem>
+isEmpty(): boolean
WolframAPI
-appid: String
+doInBackground(MenuItem... items): String
Parse
+parseRestaurants(String jsonData, double lat, double lon): ArrayList<RestaurantInfo>
+parseMenu(String jsonData): ArrayList<MenuItem>
+parseNutritionInfo(String xmlData): NutritionInfo
-getGrams(String attribute, String info): double

[3]

API requests can be made through the apiInterface class. Using this static class, you can request a list of restaurants from Yelp given a specified set of coordinates, a menu from a given restaurant, or the nutritional info about a certain menu item. Once a request has been made, a separate thread is created to query the desired API service and will wait until a

[3] Figure 3 - Data Classes and API

response is received. The json or xml string that is given back to us from the API service is then parsed into and returned through information classes to make handling the information easier.

The Yelp API requires a valid pair of latitude and longitude coordinates. A valid pair of coordinates is between the values -180 and 180. The query to the Yelp servers is a simple url call with query parameters consisting of the user's coordinates, the api key given to you by Yelp, and the category of restaurants you want to lookup, which in our case is fast food. The resulting string given to us from Yelp is a json file consisting of a list of restaurant names and other information about them sorted by distance to the user. This string is then sent to the parser. The parser then takes this json string from the api and parses it into the restaurant name, address, longitude and latitude. This a restaurant class object is created and the object gets added to a restaurant array list this list is then returned to to the front end for filling in button and the names are passed into locu to get menus.

The Wolfram API requires a valid MenuItem object, which includes having both an item name and the restaurant name the item belongs. The query to the Wolfram servers is a url call with query parameters consisting of the query string, which in this case is the item name concatenated with the restaurant name, as well as the app id and the pod id that will give us the desired information. Specifying the pod is important as it helps speed up the query time and ensures we get back the information we want. The resulting string given to us from Wolfram is an xml file consisting of a plaintext representation of the nutritional information Wolfram has on the given menu item. This string is then sent to the parser.

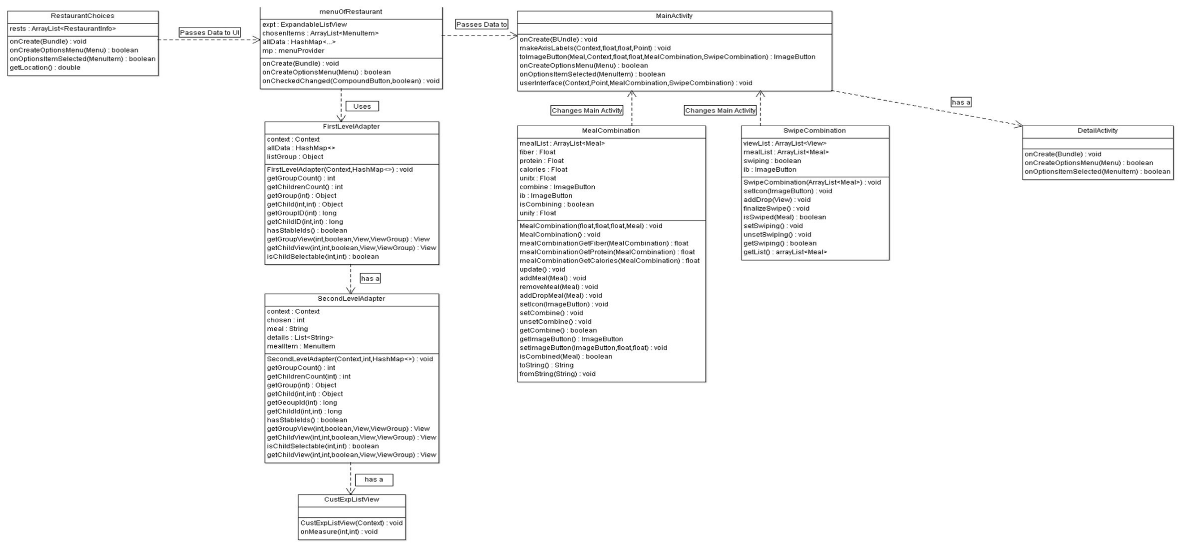
The parser first finds the plaintext of the nutritional info in the xml string. Once it has this, it looks through the plaintext string for the calories, the protein and the dietary fiber. Because protein and fiber can be listed as either grams or milligrams, a separate function checks to see which representation is used, and divides the amount by 1000 if it is represented in milligrams. A new NutritionInfo object is created to store this information and is sent back to the apiInterface which then gets returned by the apiInterface as well.

The Locu API requires a valid RestaurantInfo object, which includes having the restaurant name and coordinate information. Locu requires an html post containing a json object with a fields list, which specifies what information you want to receive, as well as a venue query object and a menu item query object specifying what search parameters you want Locu to use. The resulting string given to us from Locu is a json file consisting of a list of restaurants labeled by their locu ids, and a list of each restaurant's menu items if Locu has that restaurant's menu information. This string is then sent to the parser.

The parser then takes this json string from the api and parses it to get the names of the meals of a restaurant. The names are inputted into a MenuItems object and this object is added to a MenuItems array list. Which is returned to the MenuProvider.

The item list given to us by the parser is then passed into a new MenuProvider object, which will then be returned by the apiInterface. MenuProvider first calls wolfram on each item in this list using the apiInterface, then proceeds to sort the items into 4 categories based on the calories in the items. Each of these 4 categories is then placed into a hashmap needed by the menu screen activity. MenuProvider also provides several functionalities such as checking if it is empty, getting all the items in a single category and getting all the items that have been selected by the user.

Section 5.2 User Interface


RestaurantChoices
rests : ArrayList<RestaurantInfo>
onCreate(Bundle) : void
onCreateOptionsMenu(Menu) : boolean
onOptionsItemSelected(MenuItem) : boolean
getLocation() : double
Passes Data to UI
menuOfRestaurant
expt : ExpandableListView
chosenItems : ArrayList<MenuItem>
allData : HashMap<...>
mp : menuProvider
onCreate(Bundle) : void
onCreateOptionsMenu(Menu) : boolean
onCheckedChanged(CompoundButton,boolean) : void
Passes Data to
MainActivity
onCreate(BUndle) : void
makeAxisLabels(Context,float,float,Point) : void
toImageButton(Meal,Context,float,float,MealCombination,SwipeCombination) : ImageButton
onCreateOptionsMenu(Menu) : boolean
onOptionsItemSelected(MenuItem) : boolean
userInterface(Context,Point,MealCombination,SwipeCombination) : void
Uses
FirstLevelAdapter
context : Context
allData : HashMap<>
listGroup : Object
FirstLevelAdapter(Context,HashMap<>) : void
getGroupCount() : int
getChildrenCount() : int
getGroup(int) : Object
getChild(int,int) : Object
getGroupID(int) : long
getChildID(int,int) : long
hasStableIds() : boolean
getGroupView(int,boolean,View,ViewGroup) : View
getChildView(int,int,boolean,View,ViewGroup) : View
isChildSelectable(int,int) : boolean
has a
SecondLevelAdapter
context : Context
chosen : int
meal : String
details : List<String>
mealItem : MenuItem
SecondLevelAdapter(Context,int,HashMap<>) : void
getGroupCount() : int
getChildrenCount(int) : int
getGroup(int) : Object
getChild(int,int) : Object
getGeoupId(int) : long
getChildId(int,int) : long
hasStableIds() : boolean
getGroupView(int,boolean,View,ViewGroup) : View
getChildView(int,int,boolean,View,ViewGroup) : View
isChildSelectable(int,int) : boolean
getChildView(int,int,boolean,View,ViewGroup) : View
has a
CustExpListView
CustExpListView(Context) : void
onMeasure(int,int) : void
Changes Main Activity
Changes Main Activity
MealCombination
mealList : ArrayList<Meal>
fiber : Float
protein : Float
calories : Float
unitx : Float
combine : ImageButton
ib : ImageButton
isCombining : boolean
unity : Float
MealCombination(float,float,float,Meal) : void
MealCombination() : void
mealCombinationGetFiber(MealCombination) : float
mealCombinationGetProtein(MealCombination) : float
mealCombinationGetCalories(MealCombination) : float
update() : void
addMeal(Meal) : void
removeMeal(Meal) : void
addDropMeal(Meal) : void
setIcon(ImageButton) : void
setCombine() : void
unsetCombine() : void
getCombine() : boolean
getImageButton() : ImageButton
setImageButton(ImageButton,float,float) : void
isCombined(Meal) : boolean
toString() : String
fromString(String) : void
SwipeCombination
viewList : ArrayList<View>
mealList : ArrayList<Meal>
swiping : boolean
ib : ImageButton
SwipeCombination(ArrayList<Meal>) : void
setIcon(ImageButton) : void
addDrop(View) : void
finalizeSwipe() : void
isSwiped(Meal) : boolean
setSwiping() : void
unsetSwiping() : void
getSwiping() : boolean
getList() : arrayList<Meal>
has a
DetailActivity
onCreate(Bundle) : void
onCreateOptionsMenu(Menu) : boolean
onOptionsItemSelected(MenuItem) : boolean

[4]

As described previously, user interfaces are created as Activity classes in Android, which describe how the objects in the view will react to different user actions. The end user interacts with three main activities in our application. One activity allows the end user to choose a fast food restaurant in their vicinity. Another activity provides the menu from the chosen restaurant, along with the nutrition information associated with that meal. And the last activity allows the user to view the meals plotted on a visualization that is meant to simplify nutritional comparison between meals. This last activity will be discussed in the next section.

The activity that controls locating and selecting restaurants is RestaurantChoices. When choosing a restaurant, the user is given a dynamically generated list of buttons, each labeled with a restaurant name. There were several options in terms of designing a selection interface. We considered checkboxes or dropdown menus, but a scrollable list of buttons proved to be a superior choice. The buttons are laid out vertically and span the width of the screen. This reduces the effort and likelihood of error when selecting a restaurant. Currently, we have code that utilizes location services to get the user's gps coordinates and call Yelp with an API instruction to retrieve data about the restaurants nearby. There is a significant tradeoff between usability and performance with using APIs.

In order to filter all no-menu restaurants out of the button list, Locu - an API for menus - would have to be called for each restaurant, whether or not the user would want to look up menu items from that restaurant. Thus, we chose to leave all the restaurants on the list regardless of if a menu is available and create buttons for them.

[4] Figure 4: User Interface

Also, to save on performance, we make updating gps coordinates a decision for the end user. GPS is only called when first opening the app. After that, if the user is still using the app in a single session but wishes to update the gps location, they can choose the Locate Restaurants button that appears at the end of the restaurant list, out of the way of the user's normal use of the app since most people do not change locations often and thus we predict this will be a less-used function.

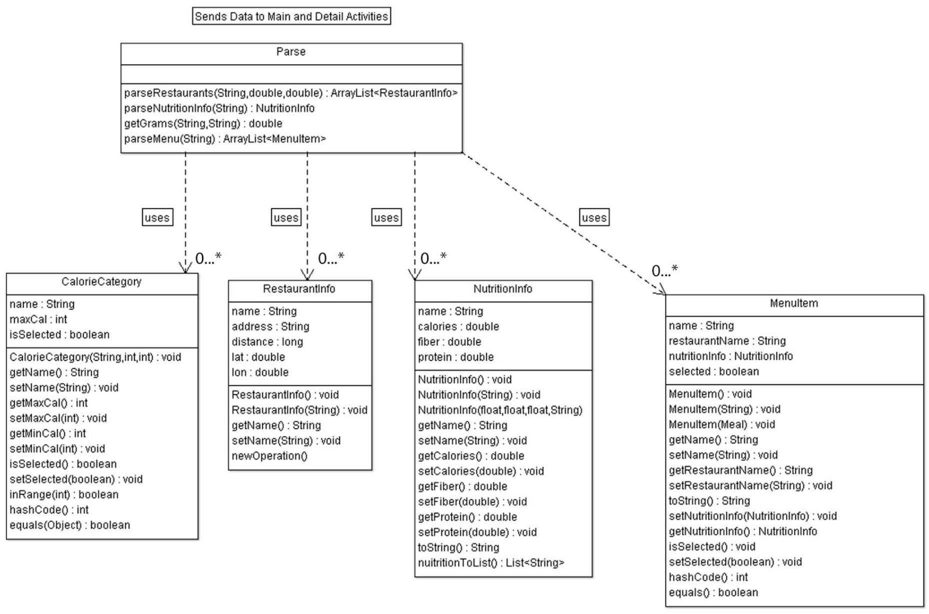
The second activity is a three level ExpandableListView with checkboxes for selection. The motivation for using three-level ExpandableListView, even though it was difficult to implement correctly and no native Android views provide a tree structure, was due to usability concerns. People feel overwhelmed when presented with too much information at once, but we needed to be able to show them the calorie category of each item - whether it was high calorie, low calorie, or somewhere in between - as well as a summary of the nutrition info for each menu item so that they do not waste time placing items they definitely do not want onto the visualization. Instead of listing the meal items in an ExpandableListView, some design alternatives could have been buttons that opened either summary screens or overlays, but removing the user from the flow of the application can be frustrating if a user wants to look at many menu items at a time. Anything that opens new screens or views has the potential to take time, particularly on older phones, and is not conducive to allowing a user to compare two meals. Although the visualization screen in our app functions as a means for comparison, it is meant to help users visualize tradeoffs - for instance if a meal is low in fat but high in calories - as well as trying combinations of menu items to see what their net calories, fat, and protein counts will be. The visualization is not necessarily for comparisons that a user could easily make on their own, which is why we display menu item nutrition info in the menu selection screen and thus, we use expandable lists to allow easier initial comparison without interrupting the workflow of the selecting user.

Each calorie category (the top level expandable list) has a checkbox that propagates selection to all its children. The children have checkboxes of their own if individual selection is desired. Choosing a top level checkbox does not mean the end user has to select all the children individually. They can deselect children and the list and checkboxes will update.

In the code, selection is controlled in the onCheckedChange() method in the activity MenuOfRestaurant. Because this method only receives a buttonView, to differentiate a checkbox of the child from the checkbox of the parent, the Java operator isInstanceOf confirms whether the buttonView is of type CalorieCategory or MenuItem. If it is a CalorieCategory, the activity finds all the menu items listed beneath this category and marks them all as selected. If the buttonView is a MenuItem, then it toggles the item, so that if it was previously selected, it is now deselected or if it was not selected, it becomes marked as selected. All chosen items are placed into an ArrayList and shuttled to the visualization via an Android Intent.

Intents allow buttons to pass data from one screen to the next. It is a more Android style of accessing data and it generally means that the data is being generated on the screen. This Intent method is, in particular, the only way we managed to capture buttonView changes correctly. When we had the menu items being updated through communication with the MenuProvider class which generates the data structure the ExpandableListView displays, the checkboxes stopped updating themselves correctly. There are two adapters associated with the three-level ExpandableListView. The first adapter maps CalorieCategory instances onto the top level list and then creates an adapter for each of its children. The second-level adapter is the one each child of a CalorieCategory created and it maps the child as the parent-level and the nutrition info as the child-level. The adapters map the data, inflate and populate the ExpandableListView cells with text and checkboxes and provide the means for identifying which item was selected. To accommodate these views, the data is structured to include a HashMap of CalorieCategory instances, which each contain a HashMap as the value. These value-HashMaps have MenuItem keys that map to List<String> structures that contain all nutrition info for the menu item. MenuProvider is responsible for gathering the data and structuring it correctly. Locu is queried for the menu, and then Wolfram for the nutrition info. Our main issue with how this screen works is that the queries for a whole menu take a long time. Each menu item has to call Wolfram to ask for nutrition data. We did not know that the calls would be so temporally expensive, so as it stands, the code still uses these expensive API calls but an alternative solution would be preferred and shall be discussed in the section on future work. Besides the ExpandableListView, there are also two buttons on this menu item selection screen. One button allows the end user to decide that nothing on the menu is to their liking and return to restaurant selection. The other button passes the selected items to the visualization. The end user is constrained, through available buttons to always have a menu selection preceding the visualization. Data from the visualization is passed through all screens so if menu items were pinned on the visualization, the user will have to remove them in the visualization. However, the data is not kept between sessions of opening and closing the app. We assume a pattern of behavior wherein the end user opens the app whenever they wish to eat out and that they explore different meals and different restaurants. They are saved from the work of clearing the visualization of all points before they start selecting restaurants in a new session. This is safer than having stale points pinned to the visualizations across sessions. Stale points may be outdated because of menus change, sometimes seasonally or geographically. To ensure our data is always fresh, we do not save previous sessions.

Section 5.3 Visualization



[5]

The visualization is the main feature of our application. The main functionality of the visualization occurs in three classes: `Meal`, `MealConbination` and `SwipeCombination`. Meals are stored in the omonimous class, used to store the meal's information as well as the `ImageButton` and `ImageView`, which are needed to erase the points while swiping. The combination of meals into a "menu" of sorts is done in `MealCombination`, where we store an array of `Meals`, as well as the global data of the combination, that we update the data structure to show how healthy the selected meal would be within the visualization chart. `SwipeCombination` is in charge of swiping controls and contains a method for allowing users to select a group of meals and delete them simultaneously. This class has the updateable list of `Meals` as well as a list of the `ImageButtons` that were selected to be removed from the screen. The final swipe is done when the swipe icon is clicked, which executes the method `finalizeSwipe` where the majority of the logic of the swiping takes place.

The second activity used in the visualization is `DetailActivity`, used to show the nutritional information of a selected `Meal` or `MealCombination`. This is done by passing the information to and from the `DetailActivity` with `Intents`. The list of `Meals` is passed to all the other activities to ensure continuity through all the app.

As requested to us, both the background and the points of the chart are color-coded to easily show their caloric content. The individual points and the combinations change size depending on the calorie count, a change that is easily seen in real time by adding or removing `Meals` to a `Combination`.

[5] Figure 5: Visualization

Section 6: Installation

To install the application EatRight onto an Android, first download the .apk file onto your phone or tablet. Then set your phone or tablet so that it allows apps from unknown sources. This can be done in Settings -> Security. All that is left is to install the .apk. It will then appear under the name EatRight_UIUC inside the folder with all the applications.

Section 7 : JavaDocs

Our code has been documented for ease of use for future users using the JavaDoc standard for documentation. Android Studio provides a good generator to create html pages of this documentation. With the project open go to Tools ->Generate JavaDoc and then follow the onscreen instructions to create the HTML documentation.

Section 8: Future Work

The API calls are slower than we expected and not very reliable. A preferred solution, knowing what we do now about how difficult it is to find restaurant menus and get nutrition info via Wolfram's API, would be to create a database using both API calls and manually add menus, then use this collection of data to pull the necessary information for the app. Creating such a database would likely be another semester's work for a database course so it was not attempted in our project.

The professor who oversaw this project mentioned a desire for internationalization and having the app be available on the Google PlayStore. This would require interpreting the English into different languages and allowing changes in language to appear in the settings. There is a stackOverFlow article detailing how to export the .apk file from our project so we can potentially upload to PlayStore though we do not do so, since it would require getting a developer's account [3]. For now, the .apk can just be installed onto a phone by dragging and dropping a file from the computer's file system to a connected android phone.

In the future a system that allowed for more than one combination and dual selection of points for both combinations and swipes would give the user a greater amount of freedom to showcase his/her tastes fully, and maybe by using a database the data passage between screens of the application could be reduced and simplified.

Reflection (Siddharth Bhaduri):

Through this project and the XP process that we followed I was able to learn a lot about Android mobile development. For this project I learnt a lot about the testing environment within Android Studio (which was the IDE that we used for this project) and about how to do mobile application development testing. This project also helped me understand the steps that are needed to build a mobile application that includes the development of use cases and user stories, testing, and finally the packaging of the application to be deployed and used by users.

Reflection (Clarence):

I have learned a few things while embarking on this project, namely getting familiar with android and android studio, and learning how to write and execute useful UI tests. A major part of my role in the project was to create useful tests for various parts of the app. I ended up having to do a good amount of research into the area of automated UI testing and I have learned a lot about how it works and in what cases it is extremely useful.

Reflection (Alhaji):
The project helped me a lot with coordinating with a team in executing deliverables. With working with Matt I learned a few things about api's and working with java in a mobile dev environment. Working on the parser I learned that I do not have to implement everything from scratch. Using the internet as tool to help me debug and make functions simpler and cleaner was a useful skill to learn. Learning about the different tools like use case diagrams and actor goals list are useful skills as well.

Reflection (Matt):
This project taught me a lot about how to call various different APIs. Many APIs use different methods to receive and respond to queries, and the 3 different ones used in this app all showed me how different each one can be. Yelp required a authentication system and Locu required an included json object string while wolfram simply needed the api key given to us. Working on this app also taught me how to use android studio for both developing and testing purposes.

Reflection (Lucas):
Being the first time I worked on a Android application I have learned how to conect, draw and modify the different components of an application. From being able to draw images on the screen with and without the use of .xml files, to the update and modification of those images to change based on the user input. The XP process was very useful to us on this type of project, allowing us to showcase the app every couple of weeks and test it throughout the whole semester.

Reflection (Kathleen):
I have discovered that in design and software engineering, it takes planning and iteration to get a project to meet specifications and there is always room for improvement. Android programming, iterating to meet client specifications, testing, and working in an iterative process with a team was an educational experience that led to a better understanding of what it takes to produce a working, well-designed product.

Resource Links

1. http://developer.android.com/tools/studio/index.html
2. http://developer.android.com/tools/testing-support-library/index.html#UIAutomator
3. http://stackoverflow.com/questions/16622843/how-do-i-export-a-project-in-the-android-studio