CS446 Deliverable 5

Architecture and Design Document

Irvine Yao (b6yao)

Jialin Shan (j6shan)

Alex Lai (x7lai)

Johnny Gao (z53gao)

Architecture:

In order to fulfill our app's main functionality, the following general architecture logic of our application is the following. First, we display the questions according to the modes users are currently using. The user can further customize what questions that they want to have asked by using the filter page. Then user will try to sing the correct note that is asked of them. Our app will constantly listen to the user through listeners with the mic. Once we got the sound data we will send the data to our library to get the current frequency to check whether user is singing the correct note or not. During the sound detection, we will display a bouncing arrow, asking the user to sing higher or lower. Based on our how our algorithm processes the data sent back from the library, we will display the result accordingly and if the answer is correct, we will display the next question. Otherwise, user can switch mode freely between note practice, interval practice, and triad mode, and our question page will be updated and generate question accordingly.

In order to handle the sound data transmission between users, the result checking algorithm, and the library for sound frequency, we implemented the Model View Controller architecture (figure) as our main backend structure.

Once our application has started, our model will generate the question based on our default settings data, and send them to the viewer component of our architecture. At the same time, in our Controller components, we will create a runnable thread in our VoiceListener class by utilizing a method from the Tarsos library, the library we use to handle the sound signal data. The library creates a thread that will be used for constantly getting data from the microphone and generate the correct current frequency from the data. The user will see the question through the Viewer component and try to provide the answer to controller component, by singing to the microphone connecting to our controller component. Once the controller component get the sound data, it will generate the frequency of the current sound and send this data to the Model component in real-time. In our Model component, we have multiple internal logics and algorithms that will decide whether the current note our user is singing is too high, too low or is in the target frequency range, and will update the viewer component in real-time accordingly.

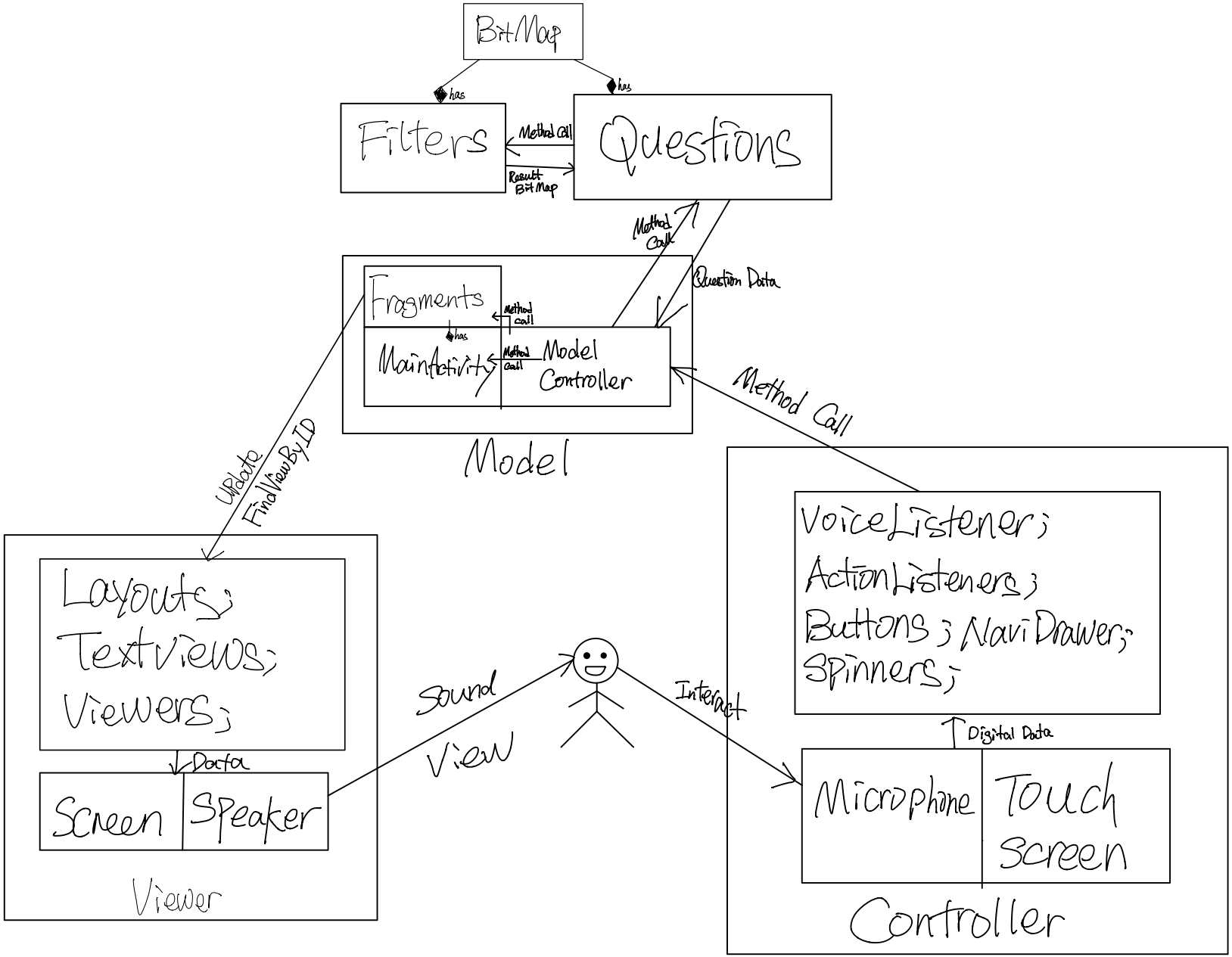
In order to let the user to navigate between different modes, we created a left drawer containing 3 buttons corresponding to the modes in the viewer. Each mode will be represented as a Fragment in the MainActivity, a part of Model component that mainly coordinate the main functionalities. Once the user

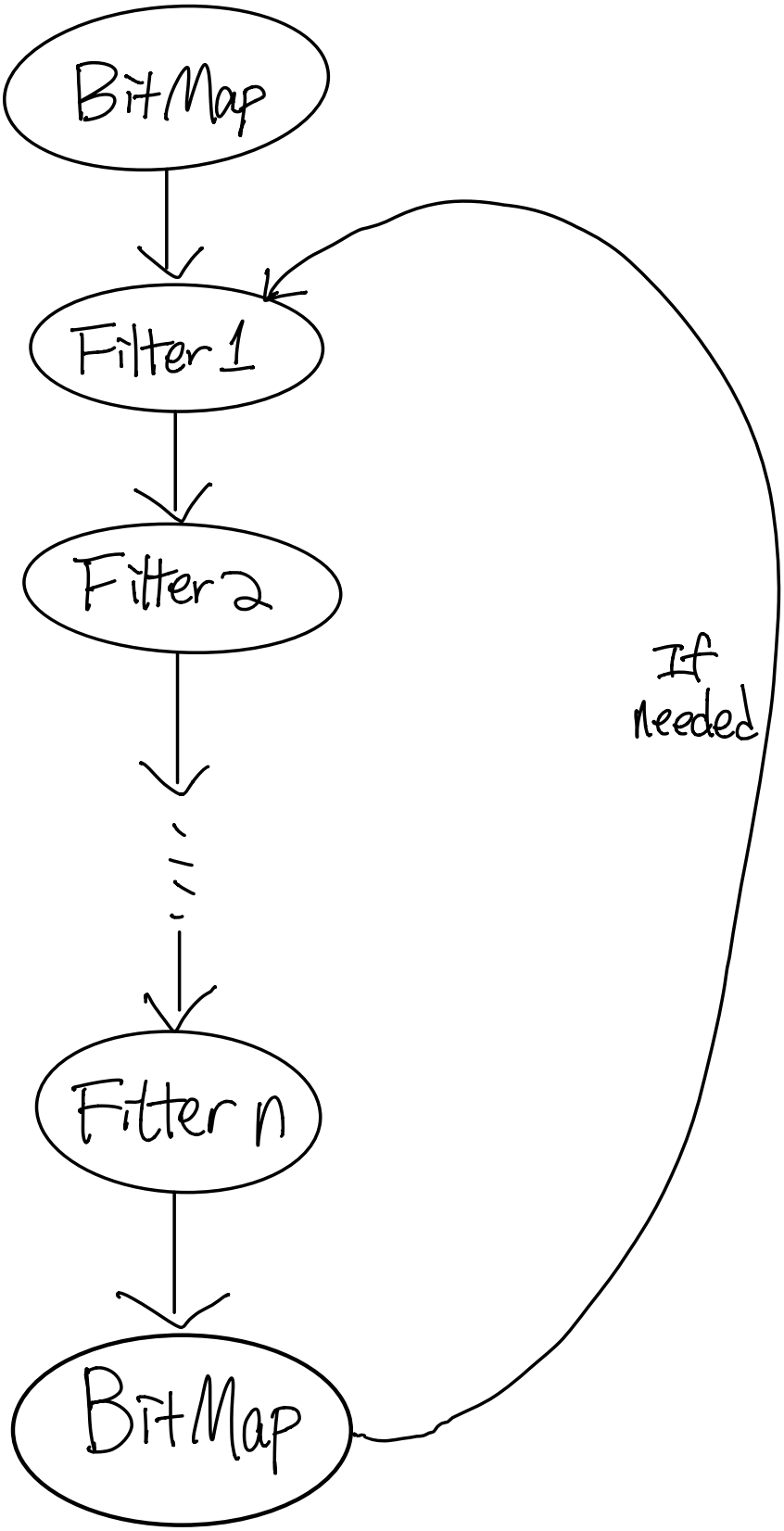
wants to switch mode, they will interact with our user interface through their touch screen, a Controller component. Afterwards, our controller components will notify the MainActivity in Model and tell MainActivity which mode is requested. Model's MainActivity then will update viewer automatically once the corresponding Fragment is been generated.

To implement filter page, we utilize pipe and filter (figure) as architecture style. At first, we have a bitmap with 88 "1"s, representing all 88 notes, which are all the musical notes available. Once user finishes selecting their preference on different spinner on the filter page, we pass our initial bitmap to a series of filters one by one to get the final result, like a pipe line. For example, if you are in note practice mode's filter page, our backend will automatically have two filter, range and scale filter, ready. Then you will select the range of your preferred notes, scale and key signatures of the notes you want to practice. Every time you make a selection, the bitmap will go through those aforementioned filters to get the updated note pool, and display under the spinner. Finally, after user clicks the note that they don't want to sing, we will have the result bitmap with all the note that has passed the filter, and waiting to be chosen as question.

For our question generation, we use inheritance class structure and polymorphism to manage our question classes. We first have an abstract question class containing all of the reusable methods and fields for every modes' question. Then we implemented three question classes as the child class for each practice mode's question.

Our architecture design is also able to support most of the NFPs, such as scalability, maintainability and performance, aforementioned in the proposal documents. Thanks to the modularity of our overall architecture design, the scalability maintainability part of our NFP can be easily achieved. For example, If we would like to add more practice modes to our app later on, we could simply create corresponding fragments and their question class under their own higher abstract class, and that would be it. Also thanks to the advantage of MVC, we can maintain each component of MVC independently without disturbing the whole system. Last but not least, thanks to the clarity of our design and library utilize, our app is able to give real-time feedback while user singing without regard to your devices, achieving the performance and usability parts of NFP.


BitMap
has
has
Filters
Method Call
Result BitMap
Questions
Method Call
Question Data
Fragments
Method call
has
MainActivity
Method call
Model Controller
Model
Method Call
Update FindViewByID
Layouts;
TextViews;
Viewers;
Data
Screen
Speaker
Viewer
Sound
View
Interact
VoiceListener;
ActionListeners;
Buttons; NaviDrawer;
Spinners;
Digital Data
Microphone
Touch screen
Controller


BitMap
Filter 1
Filter 2
Filter n
BitMap
If
Needed

Design

First of all, we store our modes in their corresponding Fragment class. GeneralFragment is a parent class where NoteFragment, IntervalFragment, NoteGraphFragment, and TriadFragment class are inherited from. Each Fragment subclass represents a practice mode: NoteFragment represent Note Practice Mode, IntervalFragment represent Interval Practice Mode, etc. GeneralFragment class have a:

- series of private fields, such as Textviews for the message storage and delivery
- handle the creation setup in onCreate method when Fragment is being created,
- have onAttach and onDetach for activities' life cycle
- series of methods for outside class to call in order to update the view properly

For each Fragment subclass, since all of the aforementioned methods are inherited from GeneralFragment parent class, all they need to do is just create their customized view in their own onCreateView method.

Our Question classes have similar structure as Fragment class system. We have a parent abstract class Question, containing:

- String[] texts -- an array of texts that consists of the question asked
- Note[] notePool -- the note pool, which can be generated from the filter page, used to form a question
- Void generate_random_question() -- abstract function to generate random question given current fields
- String getTexts() -- getter of the text of the question
- Void print_question_texts() -- print texts separated by space in stdout
- Void setNotePool(Note[] notes) -- Setter for note pool

Subclasses of Question class include NoteQuestion, IntervalQuestion, and TriadQuestion have their own void generate_random_question() to generate their customized question and put them to the field.

A Note is a class that represents a note in frequency, and String, and internal index of in total 73 notes, where range from A1 to A7 (73 total notes), and 3 constructors correspond to those 3 ways of representation.

- Note(double frequency) -- Construct A Note from frequency
- Note(int i) -- Construct A Note with index
- Note(String text) -- Constructing Note via String text

With getter methods for those three type of note representation, outside classes can easily access the note information in very diverse fashion.

For Filter function, we have a parent abstract class called Filter, with a bitmap and public applyfilter method declaration, so that outside class and call and get the filtered bitmap. Each filter page will have their own subclass and call the bitmap's method to get the result filtered bitmap.

We also have a FilterHandler class that coordinate the whole filter structure. In each filter page, we create their own FilterHandler with array of filter that it is needed, and the call FilterHandler's public method: applyFilters() and getResultBitmap() to get the result bitmap. We use pipe and filter design style logic for this entire actions because it give us scalability if we want to add more filters later.

Bitmap is a class represents all the notes in our app. Subclasses are NotesBitmap and IntervalBitmap. NotesBitmap instance in have 73 bits, represent 73 notes, and 1 means they are the candidates note and 0 means the opposite. It has the following public methods:

- getAllTrueNotesBitmap
- getNotesBitmapFromScale
- bitmapAnd -- bit wise 'and' operation on two Notesbitmap and return the result NotesBitmap
- toNotes -- convert bitmap to array of notes that are true (1) in bitmap, return the array, useful for implementing the buttons in NotesFilterPage

IntervalBitmap have 25 bits, represents 25 intervals, and have the following public methods:

- bitmapAnd(Bitmap new_bitmap) -- bit wise 'and' operation on two Intervalsbitmap and return the result IntervalsBitmap
- getAllTrueIntervalsBitmap() -- return a IntervalsBitmap of all 1
- getIntervalsBitmapFromRange(Interval from_interval, Interval to_interval) -- return a IntervalsBitmap given a low Interval and high Interval as parameters

VoiceListener, extends from Activity, is a Controller part of our MVC structure. In VoiceListener, we have a public method called startListening(), uses Tarsos's library to get the current sound frequency and pass the frequency to ModelController class by calling its processFrequency method. Main reason for this independent class is to get the modularity and clarity of our overall design.

NotePlayer is a viewer part of MVC, and have the following public methods:

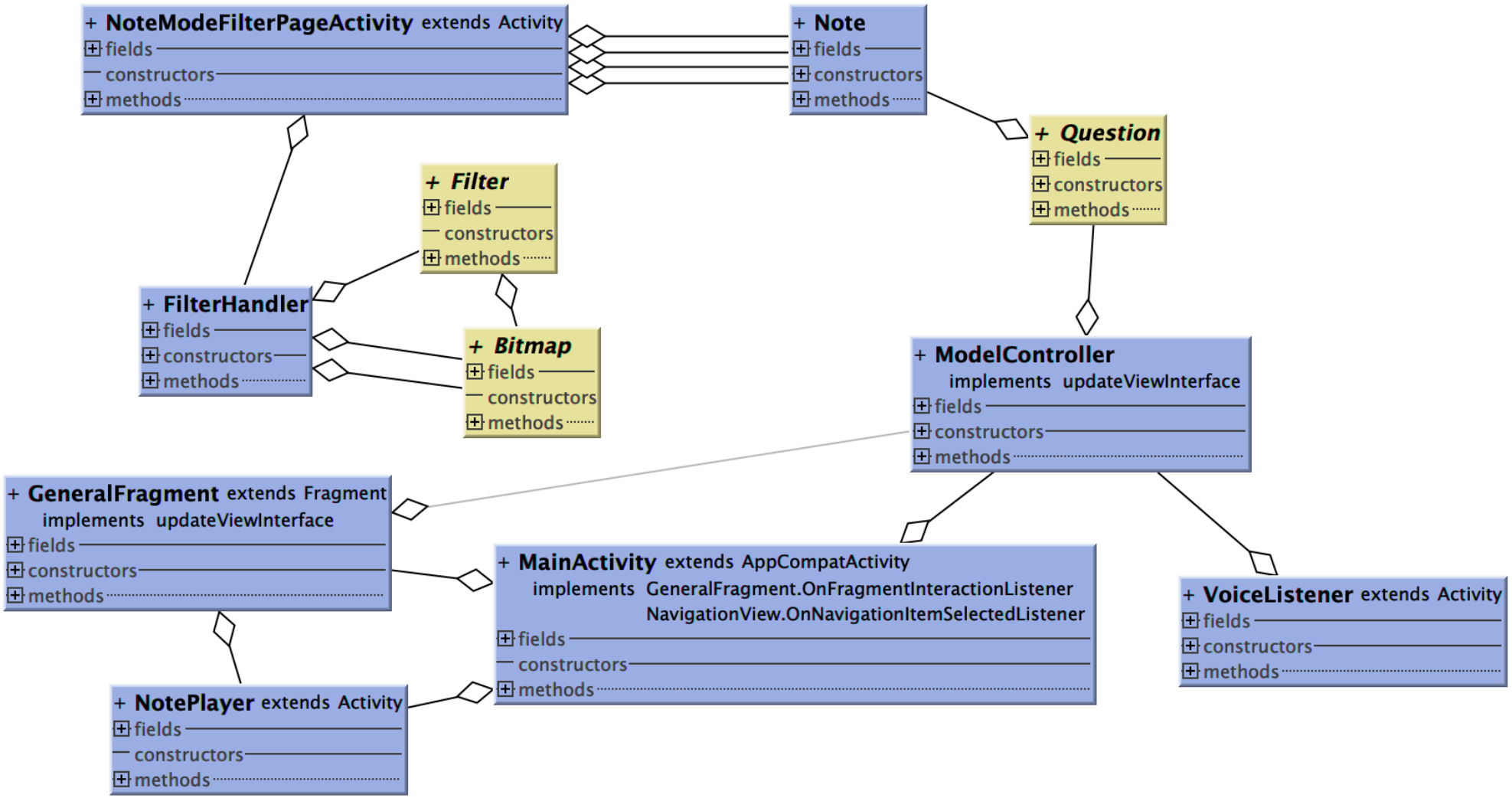
- genTone(int freqOfTone, int duration) -- for generating playable tone
- playSound() -- to play sound

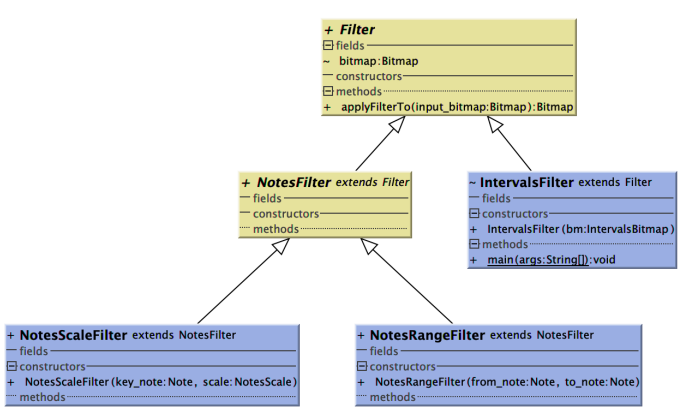
ModelController(MC) is the main part of the Model component of our MVC structure. It has the answer checking logics and field needed to do that. When MC is being created, it automatically set up all the field. The way MC know whether it is time to show the next question is that, when VoiceListener call MC's public method process processFrequency() and pass the frequency to ModelController, MC will compare the current frequency with the correct frequency, which can access from Question class's public method getQuestionNote(). If the current note is in the previously set correct frequency range, it will start the timer using T value fields in MC. If the time length of user being correct reaches t_correct, then it will call its next_question() to get the next question. ModelController can pass the data to Fragment's update methods to update the viewer properly

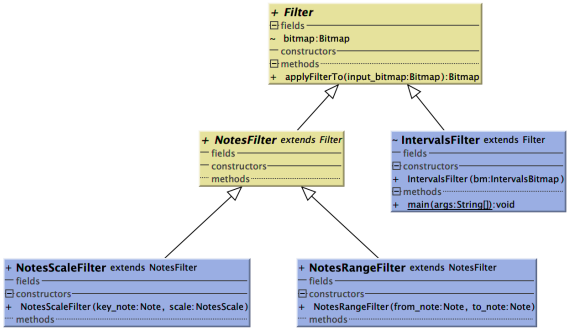
In our backend, the central coordinator of the whole app is MainActivity. Once the app is launched, the MainActivity class is being created, and within the onCreate method of MainActivity, we do all the pre-run setup such as checkMicrophonePermission, creating ModelController instance, setup front end such as setContentView, setSupportActionBar. Then create an initial Fragment and FragmentManager. The initial Fragment we create in MainActivity is noteFragment, meaning the default mode of our app is Note Practice Mode. It has a bunch of public methods that android Activity can call, such as: onBackPressed(), onNavigationItemSelected(), etc, where it can update the viewer component.

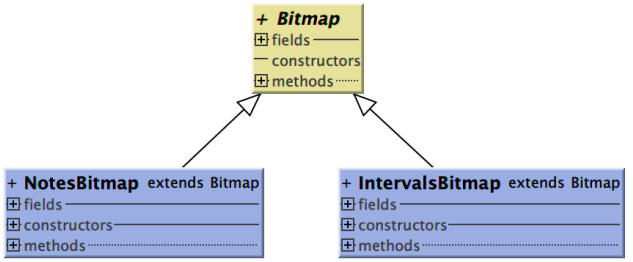
For our program we have some data coupling as a consequence of the MVC architecture, the model always passes data to the main activity, and the main activity to the active fragment. There is no data in which is processed later being passed around. Our app has sequential cohesion, meaning that it has data passing through the parts but no data is operated by two things at once. Again, the model controller is the only one doing calculations or the questions, but never both at once.

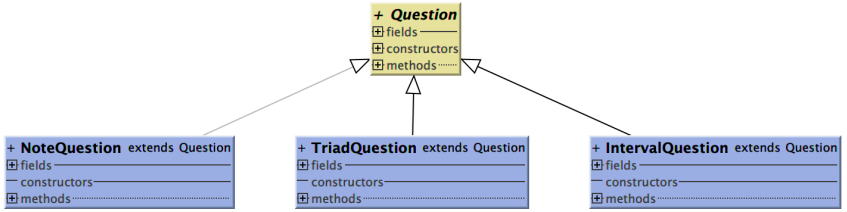
In the future we would definitely need to add more modes, specifically the song mode. As explained above we have made the system easy to generalize - to add modes we can simply add a new type of fragment that extends the GeneralFragment and add it to the navigation. We can then add the logic to handle the backend for it easily and pass data through to the fragment with a callback.

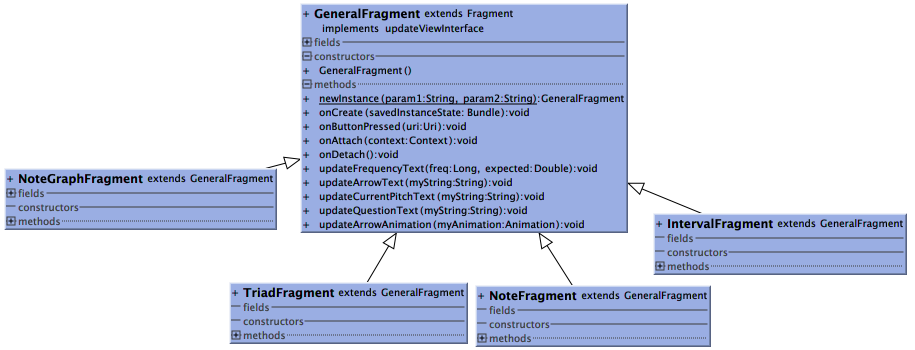

+ NoteModeFilterPageActivity extends Activity
fields
constructors
methods
+ Note
fields
constructors
methods
+ Question
fields
constructors
methods
+ Filter
fields
constructors
methods
+ FilterHandler
fields
constructors
methods
+ Bitmap
fields
constructors
methods
+ ModelController
implements updateViewInterface
fields
constructors
methods
+ GeneralFragment extends Fragment
implements updateViewInterface
fields
constructors
methods
+ MainActivity extends AppCompatActivity
implements GeneralFragment.OnFragmentInteractionListener
NavigationView.OnNavigationItemSelectedListener
fields
constructors
methods
+ VoiceListener extends Activity
fields
constructors
methods
+ NotePlayer extends Activity
fields
constructors
methods

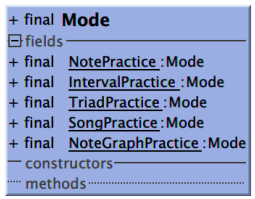

+ Filter
fields
~ bitmap:Bitmap
constructors
methods
+ applyFilterTo(input_bitmap:Bitmap):Bitmap
+ NotesFilter extends Filter
fields
constructors
methods
~ IntervalsFilter extends Filter
fields
constructors
+ IntervalsFilter(bm:IntervalsBitmap)
methods
+ main(args:String[]):void
+ NotesScaleFilter extends NotesFilter
fields
constructors
+ NotesScaleFilter(key_note:Note, scale:NotesScale)
methods
+ NotesRangeFilter extends NotesFilter
fields
constructors
+ NotesRangeFilter(from_note:Note, to_note:Note)
methods

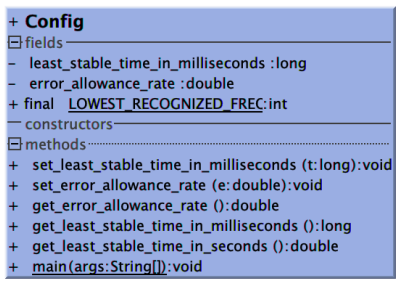

+ Filter
fields
~ bitmap:Bitmap
constructors
methods
+ applyFilterTo(input_bitmap:Bitmap):Bitmap
+ NotesFilter extends Filter
fields
constructors
methods
~ IntervalsFilter extends Filter
fields
constructors
+ IntervalsFilter(bm:IntervalsBitmap)
methods
+ main(args:String[]):void
+ NotesScaleFilter extends NotesFilter
fields
constructors
+ NotesScaleFilter(key_note:Note, scale:NotesScale)
methods
+ NotesRangeFilter extends NotesFilter
fields
constructors
+ NotesRangeFilter(from_note:Note, to_note:Note)
methods

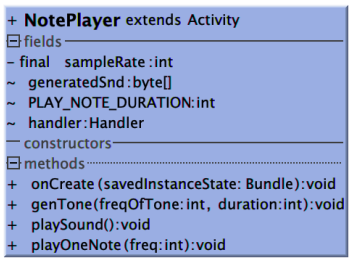

+ Bitmap
fields
constructors
methods
+ NotesBitmap extends Bitmap
fields
constructors
methods
+ IntervalsBitmap extends Bitmap
fields
constructors
methods

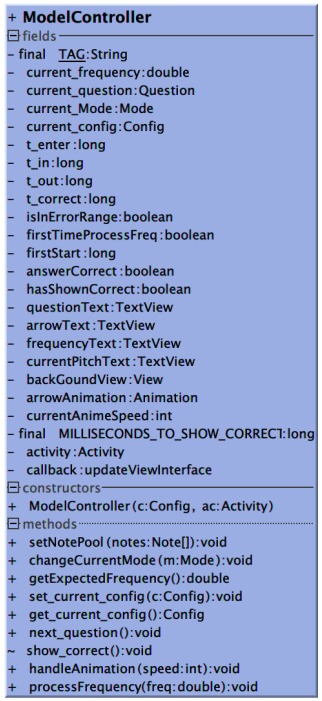

+ Question
fields
constructors
methods
+ NoteQuestion extends Question
fields
constructors
methods
+ TriadQuestion extends Question
fields
constructors
methods
+ IntervalQuestion extends Question
fields
constructors
methods

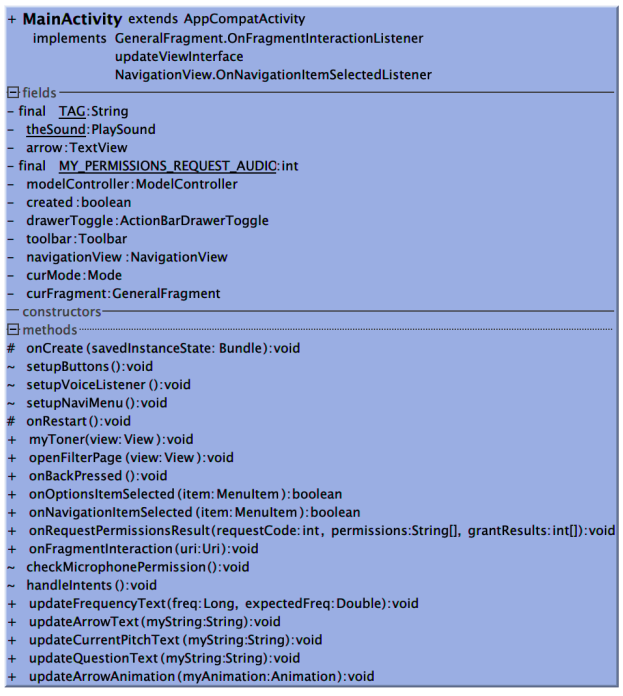

+ GeneralFragment extends Fragment
implements updateViewInterface
fields
constructors
+ GeneralFragment()
methods
+ newInstance(param1:String, param2:String):GeneralFragment
+ onCreate(savedInstanceState: Bundle):void
+ onButtonPressed(uri:Uri):void
+ onAttach(context:Context):void
+ onDetach():void
+ updateFrequencyText(freq:Long, expected:Double):void
+ updateArrowText(myString:String):void
+ updateCurrentPitchText(myString:String):void
+ updateQuestionText(myString:String):void
+ updateArrowAnimation(myAnimation:Animation):void
+ NoteGraphFragment extends GeneralFragment
fields
constructors
methods
+ IntervalFragment extends GeneralFragment
fields
constructors
methods
+ TriadFragment extends GeneralFragment
fields
constructors
methods
+ NoteFragment extends GeneralFragment
fields
constructors
methods

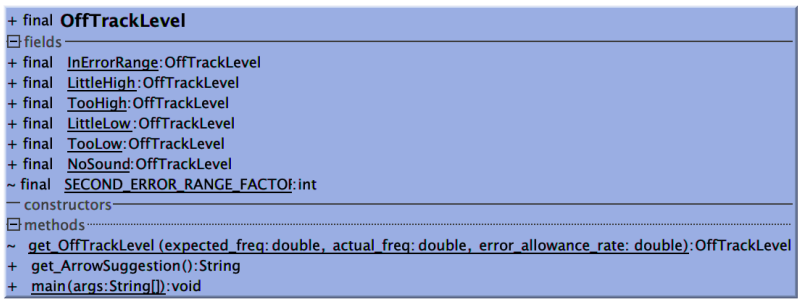

+ final Mode
fields
+ final NotePractice:Mode
+ final IntervalPractice:Mode
+ final TriadPractice:Mode
+ final SongPractice:Mode
+ final NoteGraphPractice:Mode
constructors
methods


+ Config
fields
- least_stable_time_in_milliseconds:long
- error_allowance_rate:double
+ final LOWEST_RECOGNIZED_FREC:int
constructors
methods
+ set_least_stable_time_in_milliseconds(t:long):void
+ set_error_allowance_rate(e:double):void
+ get_error_allowance_rate():double
+ get_least_stable_time_in_milliseconds():long
+ get_least_stable_time_in_seconds():double
+ main(args:String[]):void


+ NotePlayer extends Activity
fields
- final sampleRate:int
~ generatedSnd:byte[]
~ PLAY_NOTE_DURATION:int
~ handler:Handler
constructors
methods
+ onCreate(savedInstanceState: Bundle):void
+ genTone(freqOfTone:int, duration:int):void
+ playSound():void
+ playOneNote(freq:int):void


+ ModelController
fields
- final TAG:String
- current_frequency:double
- current_question:Question
- current_Mode:Mode
- current_config:Config
- t_enter:long
- t_in:long
- t_out:long
- t_correct:long
- isInErrorRange:boolean
- firstTimeProcessFreq:boolean
- firstStart:long
- answerCorrect:boolean
- hasShownCorrect:boolean
- questionText:TextView
- arrowText:TextView
- frequencyText:TextView
- currentPitchText:TextView
- backGoundView:View
- arrowAnimation:Animation
- currentAnimeSpeed:int
- final MILLISECONDS_TO_SHOW_CORRECT:long
- activity:Activity
- callback:updateViewInterface
constructors
+ ModelController(c:Config, ac:Activity)
methods
+ setNotePool(notes:Note[]):void
+ changeCurrentMode(m:Mode):void
+ getExpectedFrequency():double
+ set_current_config(c:Config):void
+ get_current_config():Config
+ next_question():void
~ show_correct():void
+ handleAnimation(speed:int):void
+ processFrequency(freq:double):void


+ MainActivity extends AppCompatActivity
implements GeneralFragment.OnFragmentInteractionListener
updateViewInterface
NavigationView.OnNavigationItemSelectedListener
fields
- final TAG:String
- theSound:PlaySound
- arrow:TextView
- final MY_PERMISSIONS_REQUEST_AUDIC:int
- modelController:ModelController
- created:boolean
- drawerToggle:ActionBarDrawerToggle
- toolbar:Toolbar
- navigationView:NavigationView
- curMode:Mode
- curFragment:GeneralFragment
constructors
methods
onCreate(savedInstanceState: Bundle):void
~ setupButtons():void
~ setupVoiceListener():void
~ setupNaviMenu():void
# onRestart():void
+ myToner(view:View):void
+ openFilterPage(view:View):void
+ onBackPressed():void
+ onOptionsItemSelected(item:MenuItem):boolean
+ onNavigationItemSelected(item:MenuItem):boolean
+ onRequestPermissionsResult(requestCode:int, permissions:String[], grantResults:int[]):void
+ onFragmentInteraction(uri:Uri):void
~ checkMicrophonePermission():void
~ handleIntents():void
+ updateFrequencyText(freq:Long, expectedFreq:Double):void
+ updateArrowText(myString:String):void
+ updateCurrentPitchText(myString:String):void
+ updateQuestionText(myString:String):void
+ updateArrowAnimation(myAnimation:Animation):void


+ final OffTrackLevel
fields
+ final InErrorRange:OffTrackLevel
+ final LittleHigh:OffTrackLevel
+ final TooHigh:OffTrackLevel
+ final LittleLow:OffTrackLevel
+ final TooLow:OffTrackLevel
+ final NoSound:OffTrackLevel
~ final SECOND_ERROR_RANGE_FACTOR:int
constructors
methods
~ get_OffTrackLevel(expected_freq:double, actual_freq:double, error_allowance_rate:double):OffTrackLevel
+ get_ArrowSuggestion():String
+ main(args:String[]):void

Work Breakdown

Classes:

Alex: Bitmap, Config, Filter, FilterHandler, GeneralFragment, Interval, IntervalFragment, IntervalQuestion, IntervalsBitmap, IntervalsFilter, MainActivity, Mode, ModelController, Note, NoteFragment, NoteGraphFragment, NoteModeFilterPageActivity, NotePlayer, NoteQuestion NotesBitmap, NotesFilter, NotesRangeFilter, NotesScale, NotesScaleFilter, OffTrackLevel, Question, Triad, TriadFragment, TriadQuestion, VoiceListener, Layouts

Jia-lin:GeneralFragment, IntervalFragment, IntervalsBitmap, IntervalsFilter, MainActivity, ModelController, Note, NoteFragment, NoteGraphFragment, NoteModeFilterPageActivity, NotePlayer, NoteQuestion, NotesFilter, NotesRangeFilter, NotesScale, NotesScaleFilter, OffTrackLevel, Question, Triad, TriadFragment, TriadQuestion, VoiceListener, Layouts

Johnny: MainActivity, ModelController, Note, NoteFragment, NoteGraphFragment, NoteModeFilterPageActivity, NotePlayer, NoteQuestion, NotesFilter, NotesRangeFilter, NotesScale, NotesScaleFilter, OffTrackLevel, Question, TriadFragment, NoteModeFilterPageActivity, TriadQuestion, Layouts, GeneralFragment, IntervalsBitmap

Irvin: GeneralFragment, IntervalFragment, MainActivity, ModelController, Note, NoteFragment, NoteModeFilterPageActivity, NotePlayer, NoteQuestion, NotesFilter, NotesRangeFilter, NotesScale, NotesScaleFilter, OffTrackLevel, Question, TriadFragment, TriadQuestion, VoiceListener, Layouts, GeneralFragment, IntervalFragment,

Architecture:

Model: Alex, Jialin, Johnny, Irvin

View: Alex, Jialin, Johnny, Irvin

Controller: Alex, Jialin, Johnny, Irvin