Universidade Federal
do Rio de Janeiro
Escola Politécnica

DESENVOLVIMENTO DE APLICATIVO ANDROID PARA ESCREVER TABLATURAS DE INSTRUMENTOS MUSICAIS TRASTEADOS

Bruno Calou Alves

Projeto de Graduação apresentado ao Curso de Engenharia de Computação e Informação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Flávio Luis de Mello

Rio de Janeiro

Março de 2018

DESENVOLVIMENTO DE APLICATIVO ANDROID PARA ESCREVER TABLATURAS DE INSTRUMENTOS MUSICAIS TRASTEADOS

Bruno Calou Alves

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO COMPUTAÇÃO E INFORMAÇÃO

Autor:

Bruno Calou Alves

Orientador:

Prof. Flávio Luis de Mello

Examinador:

Prof. Fernando Gil Vianna Resende Junior

Examinador:

Prof. Guilherme Horta Travassos

Rio de Janeiro – RJ, Brasil

Março de 2018

Declaração de Autoria e de Direitos

Eu, Bruno Calou Alves CPF 144.045.297-08, autor da monografia Desenvolvimento de Aplicativo Android para Escrever Tablaturas de Instrumentos Musicais Trasteados, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.
2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e ideias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.
3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.
4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.
5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.
6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.
7. Por ser verdade, firmo a presente declaração.

Bruno Calou Alves

Bruno Calou Alves

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Escola Politécnica – Curso de Engenharia de Computação e Informação
Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária
Rio de Janeiro – RJ CEP 21949-900

AGRADECIMENTO

Agradeço à minha família por investir em minha educação. Agradeço também aos professores por todo seu conhecimento compartilhado e pelo apoio durante a minha jornada. Aos meus colegas de colégio, curso e faculdade, obrigado por terem marcado a minha trajetória. Agradeço também à minha namorada pelo apoio dado durante todos esses anos. Por todos os momentos que passamos, sou extremamente grato.

RESUMO

Este trabalho consiste no desenvolvimento de um aplicativo Android para escrever tablaturas de instrumentos trasteados como guitarra, baixo e violão. Uma comparação é feita com os aplicativos para Android capazes de escrever tablatura, com o intuito de ressaltar suas deficiências. A partir desta análise, foi desenvolvido um aplicativo que busca eliminar estes problemas e propor uma solução para a dificuldade de documentar ideias musicais. Assim, o app provê uma solução capaz de gravar áudio, escrever tablaturas e manter as ideias organizadas, mantendo uma boa experiência de usuário e utilizando uma interface moderna de acordo com os padrões atuais de design de aplicativos.

O aplicativo foi nomeado como Song Note e foi disponibilizado através do endereço https://play.google.com/store/apps/details?id=com.brunocalou.songnote. A recepção do aplicativo foi bastante positiva, alguns usuários relataram que estavam buscando um aplicativo do tipo a bastante tempo.

Apesar da boa recepção, o aplicativo possui suas próprias deficiências, como não ser capaz de reproduzir uma tablatura, não documentar o tempo de cada nota e não ser capaz de escrever uma tablatura a partir do áudio gravado. Ainda que ele não tenha todas essas funcionalidades, todas elas podem ser desenvolvidas em versões futuras do aplicativo.

Palavras-Chave: Android, aplicativo, guitarra, violão, baixo, tablatura, áudio, Kotlin, Java, celular, programa, experiência de usuário, interface de usuário

ABSTRACT

This work is the development of an Android app capable of writing tablatures of fretted musical instruments, like electric guitar, bass, and acoustic guitar. A comparison is made with Android apps capable of writing tablatures in order to highlight their deficiencies. From this analysis, an app was developed to eliminate these problems and to propose a solution for the trouble of documenting musical ideas. Thus, the app provides a solution that accomplishes audio recording, tablature writing, and idea organization through good user experience and modern interface, according to the current app design standards.

The app was named Song Note and it was made available on https://play.google.com/store/apps/details?id=com.brunocalou.songnote. The acceptance was fairly positive, some users have reported they were looking for an app of the kind for months.

Despite good acceptance, the app has its own flaws, like not being able to play tablatures, neither to document the tempo of the notes nor writing tablature from audio. Even if it does not contain all these features, they can be developed in future app versions.

Key-words: Android, app, acoustic guitar, electric guitar, bass, tablature, audio, Kotlin, Java, mobile, software, material design, user experience, user interface

SIGLAS

API - *Application Programming Interface*

APP – Aplicativo

CRUD – *Create, Read, Update, Delete*

FAB – *Floating Action Button*

IDE – *Integrated Development Environment*

MVVM – *Model, View, ViewModel*

UFRJ – Universidade Federal do Rio de Janeiro

UI – *User Interface*

UX – *User Experience*

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 – Tema

O trabalho tem como tema o desenvolvimento de um aplicativo para a plataforma Android a fim de auxiliar músicos em sua jornada de criação e documentação de novas músicas.

1.2 – Delimitação

O projeto é voltado para músicos que gostariam de salvar suas ideias musicais, na forma de áudio, tablatura e texto, de forma simples e prática em celulares Android.

A princípio, o aplicativo provê uma notação musical voltada para instrumentos de cordas trasteados (instrumentos de corda onde as notas são delimitadas por trastes, como em um violão, onde ao pressionar uma corda, a frequência da nota é fixada pelo traste ao qual ela está apoiada), conhecida como tablatura. A implementação desta notação no projeto é focada em guitarra e violão, oferecendo assim os símbolos mais utilizados para estes instrumentos.

Em versões futuras, uma gama maior de instrumentos poderá ser suportada, além de tornar possível a escrita de partituras. Há também a possibilidade de analisar o áudio e escrever a tablatura automaticamente a partir dele, mas esta funcionalidade está, por hora, fora do escopo.

1.3 – Justificativa

O sistema de partitura, apesar de ser bastante complexo, abrange uma gama enorme de instrumentos e é a notação padrão utilizada no mundo da música. Apesar disso, existe ambiguidade ao executá-la em instrumentos de cordas pois uma nota pode ser tocada em várias posições diferentes. Desta forma, o uso da partitura em conjunto com a tablatura provê um sistema completo e eficiente para registrar músicas compostas em instrumentos de cordas trasteados.

Contudo, aprender a notação de partitura é uma tarefa desafiadora. Escrever partituras também se torna um processo mais lento em comparação ao sistema de tablatura. Desta forma, quando se trata de documentar músicas em instrumentos de cordas, a tablatura se mostra uma ferramenta poderosa, de fácil aprendizado e compreendida tanto por músicos iniciantes quanto experientes.

O grande problema em se escrever uma tablatura é que, em sua forma mais simples, o tempo de cada nota não é documentado. Logo, para que uma tablatura seja executada corretamente, é necessário ter conhecimento prévio da música, ou seja, tê-la ouvido ao menos uma vez.

Sabendo deste contratempo, um músico que queira documentar sua ideia precisa escrever a tablatura, gravar o áudio e, de alguma forma, associar os dois. É de se esperar que, na *era da informação*, existam soluções que resolvam este problema. De fato, existem, porém, para o sistema operacional Android, estas não possuem uma interface de usuário (UI) e experiência de usuário (do inglês *User Experience*, UX) agradáveis. Além disso, muitas destas soluções exigem que o tempo de cada nota também seja documentado, o que torna o processo mais lento e tedioso. Documentar o tempo não deveria ser obrigatório, e sim opcional, levando em conta que o usuário só se interesse por escrever a tablatura.

1.4 – Objetivos

O objetivo principal é criar um aplicativo para Android que permita a gravação e escrita de tablaturas para guitarra e violão de forma fácil e rápida. Logo, o aplicativo deve também possuir uma boa interface e experiência de usuário. Neste sentido, os objetivos específicos são:

- Desenvolver um sistema capaz de editar, renderizar e armazenar tablaturas de guitarra e violão utilizando algoritmos e estruturas de dados apropriados para garantir baixo uso de memória e tempo computacional
- Permitir gravação e visualização de áudio em tempo real
- Possuir uma boa *performance*, sem travamentos
- Possuir boa UX e UI

1.5 – Metodologia

O projeto pode ser dividido em três partes: a primeira, se refere à pesquisa sobre UX e UI, e ao design das telas do aplicativo; a segunda, engloba todo o processo de desenvolvimento; a terceira, remete à validação do aplicativo.

O design das telas é feito com intuito de auxiliar as decisões de projeto, como que características são mais relevantes em uma tela, quais são os fluxos para realizar certa funcionalidade, como os elementos são posicionados, etc. Após realizar o projeto do sistema, foi utilizado um aplicativo de prototipagem para fazer a validação dos fluxos. Desta forma, era possível testar o fluxo do aplicativo e, caso não correspondesse com as expectativas, corrigi-lo rapidamente.

A etapa de desenvolvimento foi feita utilizando todo o conjunto de ferramentas providas pelo ecossistema Android.

A validação do aplicativo foi feita de forma online através do *feedback* dos usuários do aplicativo.

1.6 – Descrição

Os fundamentos sobre notação musical para instrumentos trasteados serão apresentados no capítulo 2, assim como os aplicativos atuais para o sistema Android que permitem escrever músicas utilizando este sistema. Além disso, serão apresentadas as tecnologias utilizadas para o desenvolvimento do projeto.

No capítulo 3 será apresentada a solução. Serão expostos os fluxos de interação, algoritmos e trechos de código. A arquitetura do projeto e os padrões de design de código também serão abordados de acordo com as diferentes partes da solução. Também há uma análise da interface e da experiência do usuário aplicados no projeto. Ao final do capítulo, são apresentados alguns dados estatísticos do uso do aplicativo e quais são foram os resultados obtidos.

O capítulo 4 apresenta brevemente a conclusão do projeto e os próximos passos do desenvolvimento de novas funcionalidades. Nele é apresentada uma lista com a funcionalidades desejadas para as versões futuras do aplicativo.

Capítulo 2

Fundamentação Teórica

2.1 – Instrumentos de Corda Trasteados

Instrumentos de corda trasteados como o violão possuem, em sua maioria, cabeça, corpo e braço, como pode ser visto na Figura 2.1. A cabeça é utilizada para fixar as cordas do instrumento e permitir sua afinação. O braço liga a cabeça ao corpo. É nele que são posicionados os trastes. O corpo, além de fixar uma extremidade de cada corda, funciona como uma caixa de ressonância, permitindo que o som produzido pelas cordas seja amplificado.

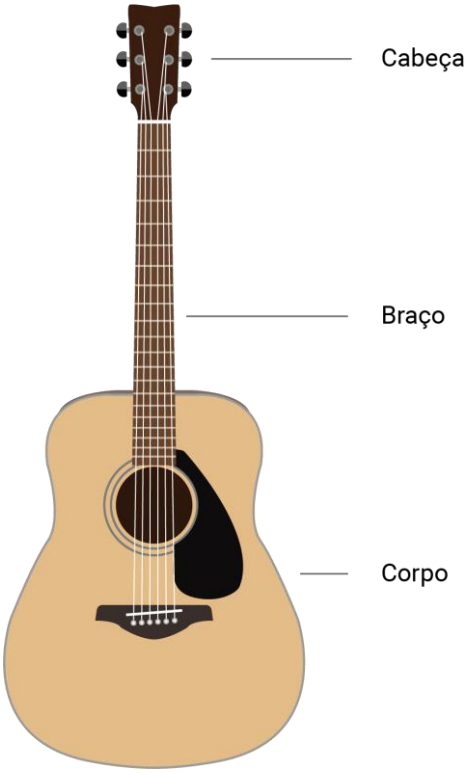


Figura 2.1 – Partes de um violão.

A espessura de cada corda varia de acordo com o tom desejado. Tons mais graves são executados em cordas mais espessas, enquanto que tons mais agudos, em

cordas mais finas. O posicionamento das cordas é feito de forma que, quando o instrumento é posicionado para ser tocado, as cordas mais graves se encontram acima das cordas mais agudas.

Ao vibrar uma corda, um som é produzido de acordo com sua afinação. Para mudar o tom, é necessário reduzir o espaço de vibração da corda; isto é feito pressionando-a contra o braço do instrumento. Como os trastes possuem posições fixas, ao pressionar a corda entre dois trastes subsequentes, a corda sempre vibrará em uma mesma frequência. Desta forma, o traste torna o tom resultante fixo e preciso.

2.2 – Sistema de Tablatura

Tablatura é um sistema de notação musical bastante utilizado em instrumentos de corda trasteados, como guitarra, baixo e violão. Em sua essência, é documentada a posição dos dedos no instrumento, diferente de uma partitura, onde as notas musicais são representadas, como pode ser visto na Figura 2.2.

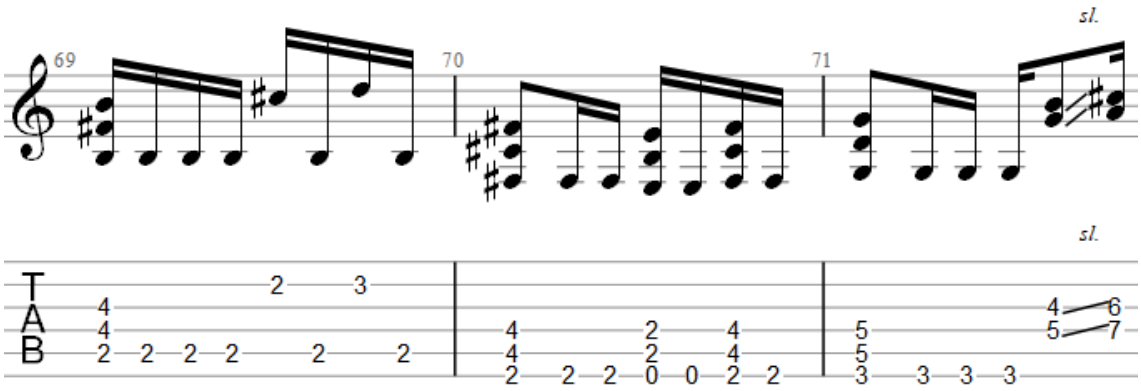


Figura 2.2 – Comparação entre partitura (superior) e tablatura (inferior)
Fonte: Programa Guitar Pro 7 [1].

Em uma tablatura, cada linha representa uma corda do instrumento, logo, tanto em violão quanto guitarra, utilizam-se, geralmente, seis linhas. A primeira linha, localizada na parte superior, representa a corda mais fina, enquanto que a última, a mais espessa. Pode-se acrescentar no início da tablatura, à esquerda de cada linha, o tom da corda correspondente quando tocada de forma aberta (sem pressionar o braço do instrumento). Desta forma, esta coluna de tons define a afinação do instrumento. Caso esteja ausente, é esperado que se utilize sua afinação padrão do instrumento.

Nas linhas, são posicionados números que representam os trastes a serem utilizados. Quando trastes são posicionados em uma mesma coluna, as cordas relativas devem ser tocadas ao mesmo tempo. Símbolos também são utilizados para representar técnicas aplicadas ao tocar uma nota, como pode ser visto na Figura 2.3.

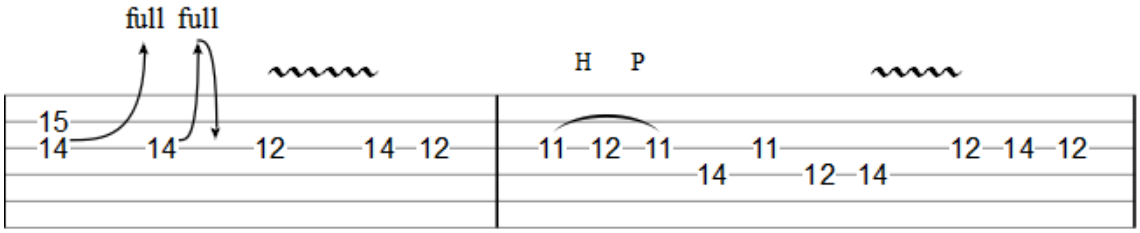


Figura 2.3 – Exemplo de tablatura
Fonte: Programa Guitar Pro 7 [1].

Não existe uma forma padrão de escrever uma tablatura, porém existem convenções. Ao utilizar um editor de texto, é comum escrever os símbolos com caracteres comuns. Já em um editor de tablatura, são usados desenhos e caracteres. A Figura 2.4 faz uma comparação entre os dois sistemas utilizando um mesmo trecho de música.

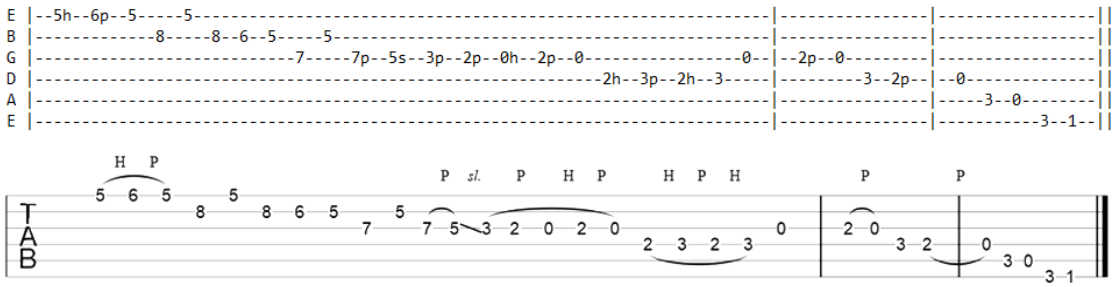


Figura 2.4 – Comparação entre uma mesma tablatura utilizando duas convenções diferentes. Em cima, utiliza-se uma notação em ASCII, em baixo, uma notação provida pelo programa Guitar Pro 7.
Fonte: Programa Guitar Pro 7 [1].

2.3 – Análise de Aplicações Existentes

A grande parte dos aplicativos disponíveis na Google Play [2] são utilizados para aprendizado de músicas já existentes. As partituras / tablaturas são exibidas para o usuário e podem ser reproduzidas com áudio dos instrumentos. Porém, poucos são os apps que permitem a escrita de tablatura. Nesta seção, estão documentados a maior parte dos aplicativos que possuem esta funcionalidade.

Os critérios utilizados para realizar a comparação de forma objetiva são a quantidade de passos necessários para inserir notas e efeitos. De forma subjetiva, existem outros aspectos como a organização da informação, a facilidade de uso e a consistência da interface.

2.3.1 – Guitar Pro

Guitar Pro [1] é um software profissional para edição de partitura e tablatura amplamente utilizado no mundo da música. Ele permite a criação de várias trilhas de partitura com instrumentos diferentes, além de ser capaz de reproduzir os áudios correspondentes. Assim, uma música pode ser escrita e reproduzida completamente pelo software. Ele também é capaz de exportar a partitura em diversos formatos, como PDF, MIDI MusicXML, ASCII, etc, além de seus formatos proprietários como .gp, .gp5, .gpx, etc, o que o torna excelente para compartilhamento de músicas.

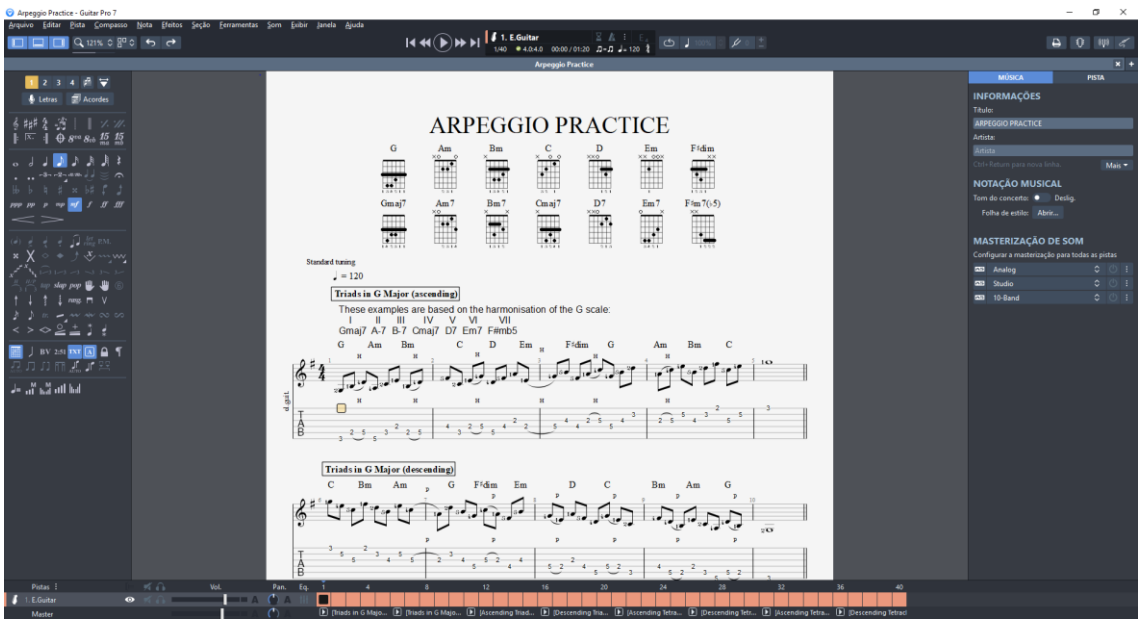


Figura 2.5 – Programa Guitar Pro 7 para desktop.

Fonte: Guitar Pro [1].

No ambiente desktop, o Guitar Pro é a melhor ferramenta de criação e aprendizado de tablaturas, porém, quando se trata de sua versão mobile, existem muitos problemas. Além de não possuir muitas das funcionalidades de sua versão desktop, sua UI é inconsistente e o aplicativo, até a data de finalização deste projeto, não é atualizado

desde 2014. O app possui somente uma versão paga, porém suas funcionalidades principais (busca, leitura e reprodução de partituras), estão disponíveis gratuitamente em outros aplicativos.

Com relação à escrita de tablatura, pode-se observar na Figura 2.6 um agrupamento de funcionalidades diferentes em um mesmo espaço. Os números (de 0 a 9) estão dispostos em uma forma não convencional, símbolos diferentes, que representam efeitos, são agrupados em um mesmo menu de forma que, para acessá-los, deve-se clicar em um botão com um ícone que tenta resumir todos estes símbolos.

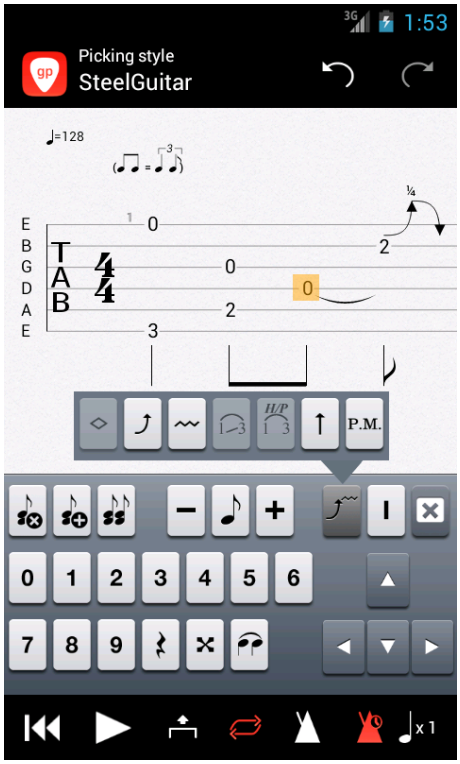


Figura 2.6 – Programa Guitar Pro para Android.

Fonte: Guitar Pro para Android [3].

2.3.2 – Guitar Tabs X

O Guitar Tabs X [4] é um aplicativo capaz de buscar por tablaturas em um servidor, baixa-las para acesso offline e reproduzi-las. Uma de suas principais funcionalidades é a de permitir a criação de novas tablaturas e salvá-las localmente.

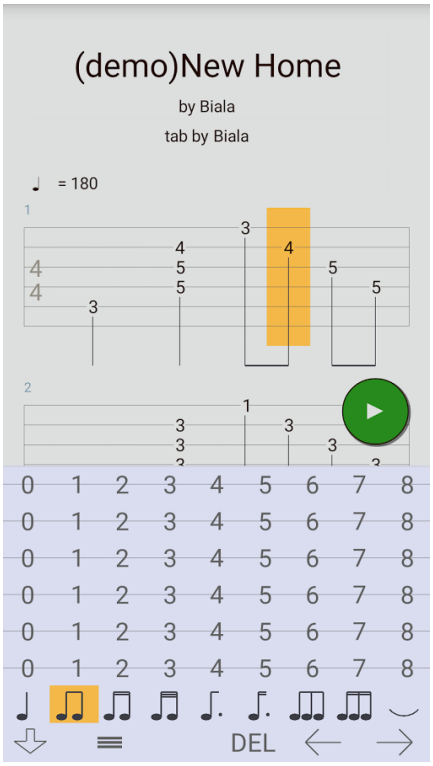


Figura 2.7 – Programa Guitar Tabs X.

Fonte: Guitar Tabs X [4].

Com relação à funcionalidade de edição de tablaturas, a forma como a entrada numérica é feita não é prática, pois é bastante propícia a erros e leva um tempo relativamente alto para ser feita. O usuário precisa localizar o número desejado na corda correta e depois selecioná-lo. Para inserir um número um pouco maior, é necessário fazer rolagem dos números na tela e repetir o processo. Por exemplo, para digitar o número 20 na terceira corda de cima para baixo, o usuário deve rolar a tela até encontrar o número 20, contar qual dos diversos números 20 está na terceira corda, selecioná-lo e verificar se ele foi inserido corretamente.

Existe um botão que substitui a caixa de seleção de números por outra com símbolos que podem ser inseridos, além de algumas outras funcionalidades, como pode ser visto na Figura 2.8. Porém, a quantidade de símbolos é bem limitada, tornando a escrita da tablatura menos fiel à música real.

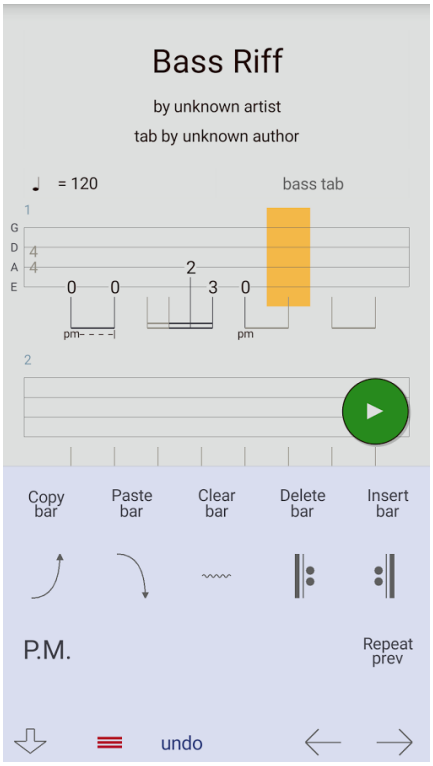


Figura 2.8 – Programa Guitar Tabs X.

Fonte: Guitar Tabs X [4].

2.3.3 – Tab Maker

O Tab Maker [5] é um aplicativo bem limitado capaz de escrever e reproduzir tablaturas. Para fazer uma inserção, o usuário deve pressionar a corda no local desejado, selecionar o número ou símbolo no menu que foi aberto e retirar o dedo da tela. Para inserir um número maior do que 9, deve repetir o processo duas vezes. Ele é capaz de enviar a tablatura por e-mail em formato de texto, porém, todo o processo de salvar, selecionar e enviar é feito de forma não intuitiva.

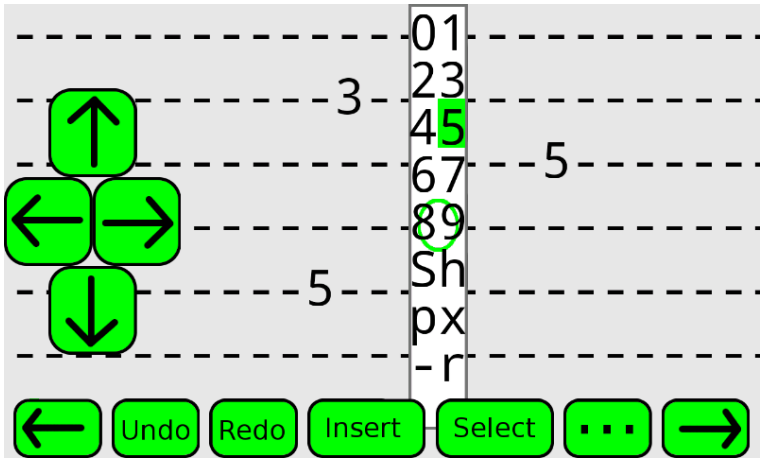


Figura 2.9 – Programa Tab Maker.

Fonte: Tab Maker[5].

2.3.4 – TuxGuitar

O TuxGuitar [6] é um app capaz de escrever e reproduzir tablaturas. Para inserir uma nota, basta digitá-la usando o teclado numérico. Para mudar o tempo da nota, basta utilizar as setas para cima e para baixo localizadas no centro do menu inferior. Para inserir um símbolo, deve-se selecionar a nota, abrir o menu superior, escolher a opção "*Effect*" e selecionar o símbolo desejado. Caso o usuário selecione incorretamente, é preciso repetir o processo clicando no mesmo símbolo para retirá-lo e depois repetir mais uma vez escolhendo o símbolo desejado. Existem símbolos que se sobrepõem, tornando desnecessário o processo de retirar um símbolo errado, porém, não tem como o usuário saber para quais opções isto se aplica. Note que em nenhum momento o programa indica qual efeito está selecionado.

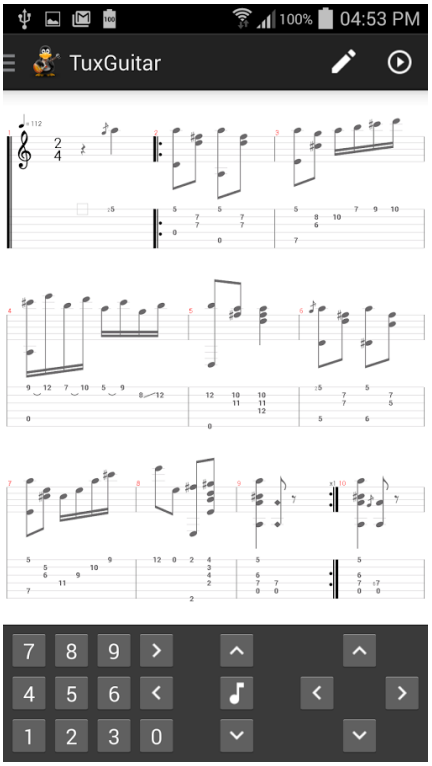


Figura 2.10 – Programa Tux Guitar.

Fonte: Tux Guitar[6].

2.3.5 – IgaraFu (Tablature Editor)

O IgaraFu (Tablature Editor) [7] é capaz de escrever e reproduzir tablaturas. Para inserir uma nota, deve-se selecionar no braço do instrumento a nota desejada e fazer um movimento rapidamente para cima, como se estivesse movendo-a para a tablatura. Vale ressaltar que em nenhum momento este gesto é mostrado para o usuário. Há também uma barra para selecionar os tempos de cada nota, porém, não existe a opção de inserir símbolos

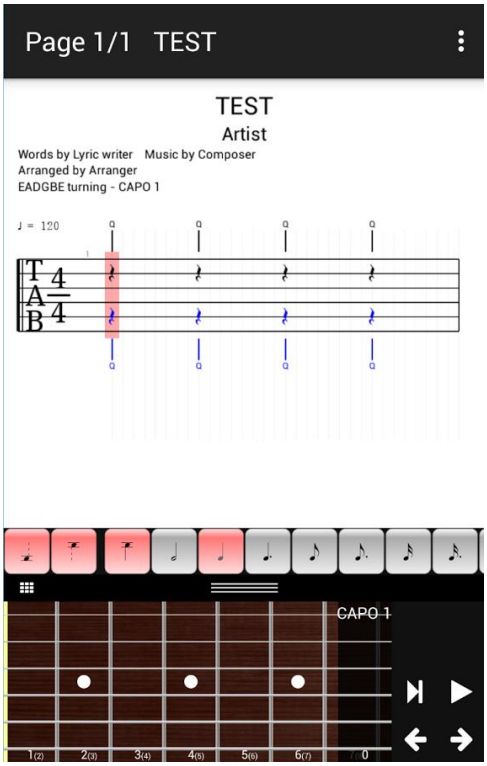


Figura 2.11 – Programa IgaraFu(Tablature Editor).

Fonte: IgaraFu(Tablature Editor)[7].

2.3.6 – Guitar Partner Lite

O Guitar Partner Lite [8] é um aplicativo capaz de escrever partituras e reproduzi-las. Para inserir um número, o usuário deve clicar no botão com o ícone de lápis e deslizar o dedo horizontalmente pela tela. Neste momento, o número aparecerá na corda selecionada e mudará de acordo com a posição horizontal do toque na tela. Para selecionar o tempo, deve-se clicar no ícone de nota musical para abrir uma janela de edição de tempo; logo em seguida, deve-se selecionar ao menos dois campos relativos ao tempo e clicar no botão de confirmação.

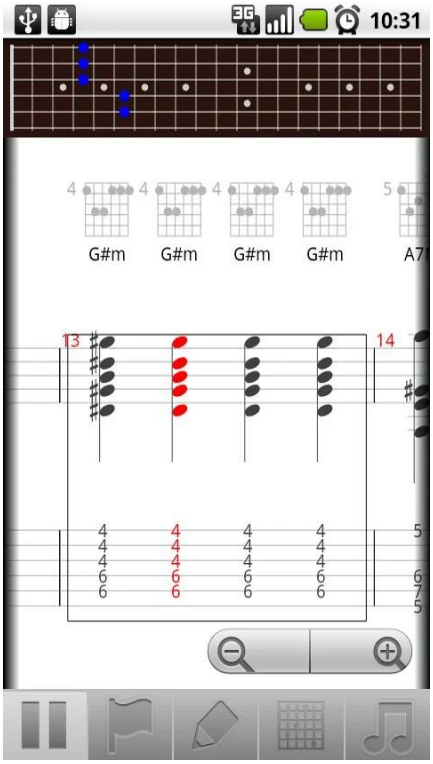


Figura 2.12 – Programa Guitar Partner Lite.

Fonte: Guitar Partner Lite[8].

2.3.7 – Guitar Notepad – Tab Editor

O Guitar Notepad – Tab Editor [9] é um aplicativo capaz de escrever tablaturas. Ele possui o mesmo funcionamento do Guitar Tabs X, porém, as notas de uma coluna são mostradas tanto na tablatura quanto na tabela semelhante ao braço do violão. A inserção de uma nota é feita clicando nesta tabela e, apesar de possuir os mesmos problemas do Guitar Tabs X a visualização é mais simples. Para inserir um símbolo, basta escolhê-lo na barra flutuante. Vale ressaltar que o app possui somente uma versão paga.

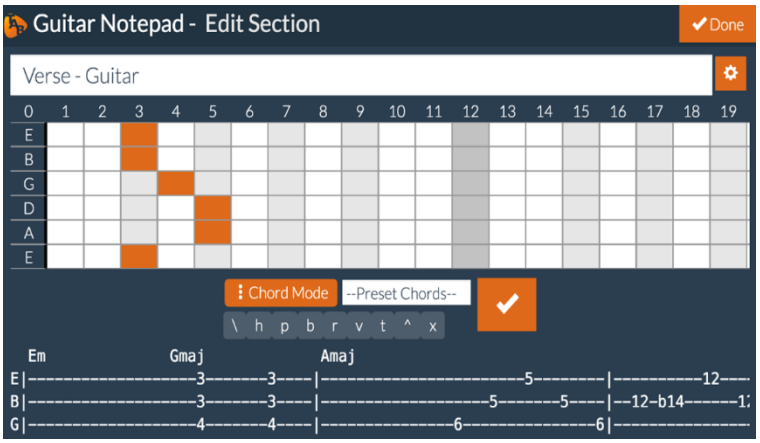


Figura 2.13 – Programa Guitar Notepad – Tab Editor.

Fonte: Guitar Notepad – Tab Editor[9].

2.3.8 – Tabify guitar composition

O Tabify guitar composition [10] é um aplicativo com um conceito interessante: ele grava o áudio e a partir dele tenta determinar a tablatura. Ao reproduzir, o áudio original é executado enquanto que a tablatura é deslizada para acompanhá-lo. Inserir ou editar uma nota é relativamente simples, basta clicar no local desejado que um menu aparecerá; depois, basta selecionar uma opção. Apesar do app ser de fato interessante, a tablatura gerada a partir do som possui muitos erros e, ao reproduzi-la, muitas vezes ela não acompanha a gravação corretamente.

Figura 2.14 – Programa Tabify guitar composition – Tab Editor.

Fonte: Tabify guitar composition [10].

2.4 – Android Studio

O Android Studio [11] é o Ambiente de Desenvolvimento Integrado (IDE) oficial para desenvolvimento de aplicativos Android. Por ser baseado na IDE IntelliJ IDEA [12], existem muitas funcionalidades e recursos que tornam o desenvolvimento mais produtivo.

A estrutura do projeto é dividida em 3 pastas principais (ver Figura 2.15):

- Manifests: Contém o arquivo AndroidManifest.xml
- Java: Contém todo o código fonte, incluindo códigos de teste usando JUnit
- Res: Contém arquivos que não são código, como layouts XML, strings e imagens

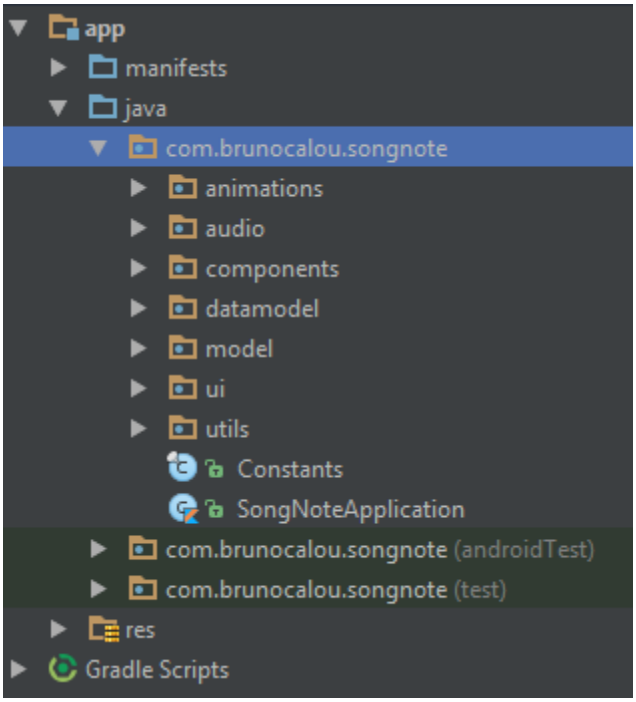


Figura 2.15 – Estrutura do projeto.

O arquivo AndroidManifest.xml é obrigatório em todo aplicativo Android. Sua função é prover informações essenciais sobre o aplicativo para o sistema Android, necessárias para o sistema antes que ele possa executar o código do aplicativo. Suas funcionalidades são:

- Nomear o pacote Java para o aplicativo. O nome do pacote serve como identificador exclusivo para o aplicativo.
- Descrever os componentes do aplicativo, que abrangem atividades (activity), serviços, receptores e provedores de conteúdo.
- Determinar os processos que hospedam os componentes de aplicativo.
- Declarar as permissões necessárias do aplicativo.
- Listar classes de instrumentação que fornecem geração de perfil e outras informações durante a execução do aplicativo.
- Declarar o nível mínimo da Android API que o aplicativo exige.
- Listar as bibliotecas às quais o aplicativo deve se vincular.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.brunocalou.songnote">

    <uses-permission android:name="android.permission.RECORD_AUDIO" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <meta-data android:name="firebase_analytics_collection_enabled" android:value="false" />

    <application
        android:name="com.brunocalou.songnote.SongNoteApplication"
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="Song Note"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name="com.brunocalou.songnote.ui.home.MainActivity"
            android:label="Song Note"
            android:windowSoftInputMode="adjustResize|stateHidden"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
```

Figura 2.16 – Parte do arquivo AndroidManifest.xml.

A pasta Java contém todo o código Java do aplicativo. Todas as classes e interfaces utilizadas pelo aplicativo definem seu comportamento, enquanto que os arquivos da pasta Res definem sua interface. Assim, todo código referente ao aplicativo, incluindo códigos de teste, devem estar presentes na pasta Java.

A pasta Res, como dito anteriormente, contém todos os arquivos relacionados à interface, além de conter arquivos de configuração do aplicativo. Sua estrutura é composta da seguinte forma:

- Animator: Arquivos XML que definem animações de propriedade.
- Anim: Arquivos XML que definem animações intermediárias.
- Color: Arquivos XML que definem uma lista de estado de cores. (por exemplo, um botão pode ter vários estados, como pressionado, com foco ou normal, logo cada cor pode ser associada a um estado).
- Drawable: Arquivos desenháveis, como imagens e XMLs
- Mipmap: Arquivos *drawable* do ícone do aplicativo referentes a diferentes densidades de tela
- Layout: Arquivos XML que definem os layouts de UI, como telas e componentes
- Menu: Arquivos XML que definem os menus do app

- Raw: Arquivos arbitrários salvos em forma bruta, ou seja, sem que haja nenhum processamento sobre eles
- Values: Arquivos XML que contém valores simples, como strings, números inteiros e cores. Usa-se um arquivo para cada tipo de recurso, de forma que propriedades semelhantes sejam agrupadas. Algumas convenções de nomes de arquivos presentes nessa pasta incluem:
    - colors.xml para valores de cor
    - dimens.xml para valores de dimensão
    - strings.xml para valores de string
    - styles.xml para estilos
- Xml: Arquivos arbitrários XML que podem ser lidos em tempo de execução

Capítulo 3

Solução

Após destacar os problemas com as soluções atuais, o entendimento sobre o problema torna possível a confecção de uma solução mais apropriada. Após definir os requisitos do sistema, é possível pensar em uma solução de interface que atenda as expectativas do usuário e ofereça uma boa experiência. Os fluxos de interação são definidos nesta etapa e, a partir deles, é possível fazer a arquitetura da solução.

3.1 – Requisitos Desejados do Sistema

O projeto deve atender os seguintes requisitos funcionais

- Permitir escrita de tablatura
    - Inserir e remover notas e efeitos rapidamente
    - Corrigir erros rapidamente, como errar ao digitar uma nota
    - Editar informações do cabeçalho (título, subtítulo, artista, afinação, tempo, comentário e capotraste)
    - Armazenar a tablatura em disco
- Permitir gravação de áudio
    - Renderizar o áudio
    - Descartar último áudio gravado
    - Continuar uma gravação
    - Se deslocar para qualquer momento da gravação com facilidade
    - Reproduzir o áudio
    - Armazenar o áudio em disco
- Adicionar informações para organização
    - Nome, descrição, instrumento e etiquetas
- Adicionar, remover e editar etiquetas

Note que neste momento, está fora do escopo do projeto prover uma solução para escrita de cifras e de partituras. Além disso, não será desenvolvida nenhuma forma de reprodução de tablatura nem de escrita automática a partir do áudio, ou seja, a tablatura será feita inteiramente pelo usuário de forma manual e não será gerada nenhuma forma de som a partir da mesma.

3.2 – Arquitetura da Solução

Esta seção tem como objetivo explicar sucintamente as diferentes partes da solução.

3.2.1 – MVVM

O padrão de arquitetura MVVM (Model, View, ViewModel) organiza a implementação em três camadas, provendo clara separação de responsabilidades.

- Model - Camada da aplicação responsável pelo armazenamento de dados e pela lógica de negócios
- View – Interface da aplicação. Informa ao *ViewModel* sobre as ações do usuário
- ViewModel – Acessa os dados vindos do *Model*, os modifica conforme o necessário e os dispõe para a *View* através de um *stream* de dados

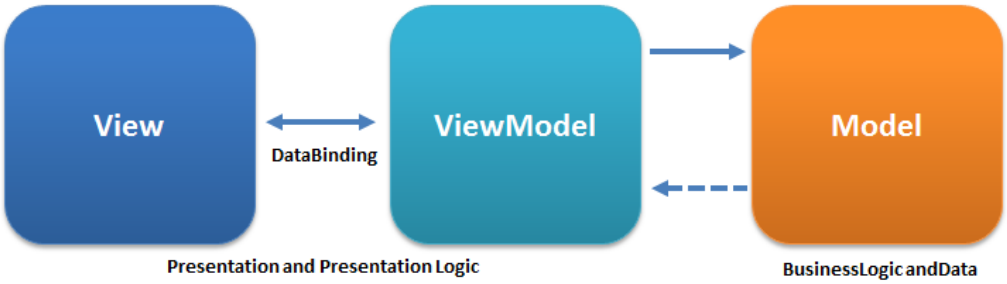


Figura 3.1 – Diagrama das relações entre Model, View, ViewModel.

Fonte: Model–view–viewmodel – Wikipedia [13].

A implementação deste modelo no projeto foi feita utilizando a biblioteca de programação reativa RxJava [14], pois ela é a biblioteca mais utilizada para este tipo de programação, possui uma documentação excelente e uma grande comunidade de desenvolvedores. Desta maneira, é possível criar *streams* de dados, além de poder utilizar o padrão de design *Observable* facilmente. Logo, as camadas do MVVM são implementadas da seguinte maneira:

- Model – Expõe os dados utilizando a classe *Flowable* do RxJava. Armazena e acessa os dados em um banco de dados local. No projeto, foi utilizado o termo *Model* para as classes que representam os dados (*Idea, Label, Tablature*, ...) e *DataModel* para as classes que lidam com o armazenamento dos dados.
- ViewModel – Utiliza o *Model* diretamente para recolher os dados necessários para a *View*. Trata os dados e os expõe como um *stream* para a *View* utilizando a classe *Flowable* do RxJava. O *ViewModel* não mantém uma referência direta para a *View*, porém ele mantém uma referência para o *Model*.
- View – Interface da aplicação. Repassa as ações do usuário diretamente ao *ViewModel* e o utiliza para recolher os dados necessários. No Android, a *View* pode ser qualquer classe responsável pela interface do usuário, como Activity e Fragment.

Uma implementação do MVVM no projeto pode ser vista nas figuras a seguir. Na Figura 3.2, é mostrado o *ViewModel* de ideias utilizado na página inicial. Note que o *DataModel* é injetado nele através de seu construtor. Na Figura 3.3, é apresentado a interface do *DataModel* utilizada por todos os *ViewModel* que precisam interagir com as ideias do aplicativo. O método de obtenção do *ViewModel* pela *View* é mostrado pela Figura 3.4. Note que todo *ViewModel* no projeto é um s*ingleton*, ou seja, só existe uma instância em todo programa. A Figura 3.5 mostra a criação de todos os *ViewModel* e *DataModel* no projeto através do *singleton SongNoteApplication.* Por fim, o *Model Idea*, responsável por conter as informações para organização da ideia e agrupar o áudio e a tablatura, é apresentado na Figura 3.6.

```
package com.brunocalou.songnote.ui.home

import com.brunocalou.songnote.datamodel.IIdeaDataModel
import com.brunocalou.songnote.model.Idea
import io.reactivex.Flowable
import io.realm.RealmResults

class IdeasFragmentViewModel (private val ideaDataModel: IIdeaDataModel) {
    fun getIdeas(): Flowable<RealmResults<Idea>> = ideaDataModel.getIdeas()
    fun getFavoriteIdeas(): Flowable<RealmResults<Idea>> = ideaDataModel.getFavoriteIdeas()
    fun getIdeasByLabelName(labelName: String): Flowable<RealmResults<Idea>> = ideaDataModel.getIdeasByLabelName(labelName)
    fun saveIdea(idea: Idea) = ideaDataModel.saveIdea(idea)
    fun setFavorite(idea: Idea, favorite: Boolean) = ideaDataModel.setFavorite(idea, favorite)
}
```

Figura 3.2 – ViewModel para o Fragment de Ideias utilizado na primeira página

```
package com.brunocalou.songnote.datamodel

import com.brunocalou.songnote.model.Idea
import io.reactivex.Flowable
import io.reactivex.subjects.PublishSubject
import io.realm.RealmResults

interface IIdeaDataModel {
    fun getIdea(id: String): Idea?
    fun getIdeas(): Flowable<RealmResults<Idea>>
    fun saveIdea(idea: Idea): Boolean
    fun getEventBus(): PublishSubject<StateChangeEvent<String>>
    fun getFavoriteIdeas(): Flowable<RealmResults<Idea>>
    fun getIdeasByLabelName(labelName: String): Flowable<RealmResults<Idea>>
    fun deleteIdea(idea: Idea): Boolean
    fun setFavorite(idea: Idea, favorite: Boolean): Boolean // Return if the method was executed successfully
}
```

Figura 3.3 – Interface do DataModel de Ideias

```
protected IdeasFragmentViewModel getViewModel() {
    if (viewModel == null)
        viewModel = ((SongNoteApplication) getActivity().getApplication()).getIdeasFragmentViewModel();
    return viewModel;
}
```

Figura 3.4 – Método utilizado no Fragment de Ideias para buscar o ViewModel correspondente

```
class SongNoteApplication : MultiDexApplication() {
    lateinit var labelDataModel: ILabelDataModel
    lateinit var labelFragmentViewModel: LabelFragmentViewModel
    lateinit var mainActivityViewModel: MainActivityViewModel
    lateinit var labelSelectActivityViewModel: LabelSelectActivityViewModel
    lateinit var infoEditFragmentViewModel: InfoEditFragmentViewModel

    lateinit var ideaDataModel: IIdeaDataModel
    lateinit var ideaEditActivityViewModel: IdeaEditActivityViewModel
    lateinit var ideasFragmentViewModel: IdeasFragmentViewModel
    lateinit var ideaViewActivityViewModel: IdeaViewActivityViewModel

    override fun onCreate() {
        super.onCreate()
        Realm.init(this)
        AudioRecorder.APP_FOLDER_NAME = FileUtils.getAppFolder(this).name
        labelDataModel = LabelDataModel()
        labelFragmentViewModel = LabelFragmentViewModel(labelDataModel)
        mainActivityViewModel = MainActivityViewModel(labelDataModel)
        labelSelectActivityViewModel = LabelSelectActivityViewModel(labelDataModel)
        infoEditFragmentViewModel = InfoEditFragmentViewModel(labelDataModel)

        ideaDataModel = IdeaDataModel()
        ideaEditActivityViewModel = IdeaEditActivityViewModel(ideaDataModel, labelDataModel)
        ideasFragmentViewModel = IdeasFragmentViewModel(ideaDataModel)
        ideaViewActivityViewModel = IdeaViewActivityViewModel(ideaDataModel)
```

Figura 3.5 – Injeção de dependências a partir do Singleton SongNoteApplication

```
open class Idea constructor(
        var title: String = "",
        var description: String = "",
        var duration: Int = 0, // [ms]
        @Ignore private var instrument: Instrument = Instrument.GUITAR,
        var audioPath: String = "",
        var labels: RealmList<Label> = RealmList(),
        var favorite: Boolean = false,
        var audioView: RealmList<Short> = RealmList(),
        var tablaturePath: String = "",
        var creationDate: Date = Date(),
        var modificationDate: Date = Date()
    ) : RealmObject(), ICopyable<Idea> {

    constructor(title: String, description: String): this(title, description, 0)

    constructor(title: String): this(title, "")

    constructor(): this("")

    @PrimaryKey
    var id: String = UUID.randomUUID().toString()
    private var mInstrument: String = instrument.toString()

    @Ignore
    var tablature: Tablature? = Tablature()
```

Figura 3.6 – Parte da implementação do Model da Ideia

Para remover o acoplamento entre o *DataModel* e o *ViewModel*, foi utilizado o padrão de injeção de dependência. Uma interface do *DataModel* é criada especificando as operações necessárias para o gerenciamento dos dados. No *ViewModel*, ao invés dele declarar como dependência a implementação do *DataModel*, ele se declara dependente da interface. Em uma outra parte da aplicação, injeta-se a implementação desejada no *ViewModel*. Desta forma, se for utilizada uma implementação que salva em um banco de dados local e haja uma mudança para outra que envia os dados pela rede, não é preciso fazer nenhuma alteração no *ViewModel*, basta injetar a dependência desejada. Note que este modelo é ótimo para a realização de testes automatizados, pois basta injetar uma implementação do *DataModel* que atenda aos casos de teste.

É interessante ressaltar que o *ViewModel* e o *DataModel* expõem um barramento de eventos. Este barramento utiliza o padrão *Publish*-Subscribe, onde é possível

enviar eventos (*pusblish*) em um canal de comunicação em que os inscritos (*subscribers*) são notificados. Ao realizar uma ação *CRUD (Create, Read, Update, Delete)*, por exemplo, o estado da ação é enviado para todos os inscritos no barramento de eventos. Isto é obtido utilizando a classe *PublishSubject* do RxJava.

No projeto, uma *View* se inscreve no barramento exposto pelo seu *ViewModel* correspondente. Quando a *View* chama um método de *CRUD* do *ViewModel*, ele repassa a ação para o *DataModel*, que por sua vez realiza a ação e publica no barramento de eventos. A *View* é então notificada sobre a ação e se atualiza para refletir o estado do sistema. Um exemplo pode ser visto na Figura 3.7.

```
disposable.add(labelFragmentViewModel.getEventBus()
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe({ t ->
            run {
                if (t.error == null) {
                    when(t.state) {
                        StateEnum.DELETE -> deleteLabel(t)
                        StateEnum.UPDATE -> updateLabel(t)
                        StateEnum.CREATE -> addLabel(t)
                        else -> {
                        }
                    }
                } else {
                    handleError(t)
                }
            }
        }))
```

Figura 3.7 – *View* de edição de etiquetas se inscrevendo no barramento de eventos do *ViewModel*

3.2.2 – Tablatura

A tablatura foi implementada utilizando três classes distintas:

- TablatureElement – Representa um elemento na tablatura. Carrega o valor do traste, corda, modificadores, símbolos e tempo. Apesar do *tempo* não ser usado atualmente, ele pode ser incluído no futuro

- TablatureNotation – Guarda todas as notas da tablatura e provê formas de busca, inserção, edição e remoção de notas, além de implementar funções auxiliares (como remoção de colunas vazias)
- Tablature – Guarda o cabeçalho da tablatura e todas as suas notas

```
public static class TablatureElement implements ICopyable<TablatureElement> {
    public final static int NO_FRET = -2;
    public final static int X_FRET = -1;
    public final static int MAX_FRET = 99;
    public final static int MAX_STRING = 9;
    public final static int MIN_STRING = 0;

    private int fret = NO_FRET;
    private Symbol symbol = null;
    private Modifier modifier = null;
    private Tempo tempo = null;

    private BendIntensity bendIntensity = null;

    private int string = 0;
    private int column = 0;
```

Figura 3.8 – Parte da implementação da classe TablatureElement

```
public class TablatureNotation implements Iterable, ICopyable<TablatureNotation> {

    /**
     * The TreeMap key is the string of the tablature element
     */
    LinkedList<TreeMap<Integer, TablatureElement>> notes = new LinkedList<>();

    public enum Symbol {
        LEGATO, PRE_BEND, BEND, PRE_BEND_RELEASE, BEND_RELEASE, SLIDE_UP, SLIDE_DOWN, PRE_SLIDE_UP, PRE_SLIDE_DOWN
    }

    public enum Modifier {
        PALM_MUTE, HARMONIC, TAPPING, VIBRATO
    }

    public enum Tempo {
        WHOLE, HALF, QUARTER, EIGHTH, SIXTEENTH, THIRTY_SECOND, SIXTY_FOURTH
    }

    public enum BendIntensity {
        HALF, ONE, ONE_AND_A_HALF, TWO
    }
```

Figura 3.9 – Parte da implementação da classe TablatureNotation

```
public class Tablature implements ICopyable<Tablature> {
    public static final String DEFAULT_TUNING = "E A D G B e";
    public static int MAX_BPM = 300;
    public static int MIN_BPM = 30;
    public static int DEFAULT_BPM = 120;
    public static int MAX_NUMBER_OF_STRINGS = TablatureNotation.TablatureElement.MAX_STRING;
    public static int MIN_NUMBER_OF_STRINGS = TablatureNotation.TablatureElement.MIN_STRING + 1;
    public static int DEFAULT_NUMBER_OF_STRINGS = 6;
    public static int MAX_CAPO_FRET = 15;
    public static int DEFAULT_CAPO_FRET = 0;

    public String title;
    public String subtitle;
    public String artist;
    public String comment;
    public String tuning;
    private int bpm;
    private int capo;
    private int numberOfStrings = DEFAULT_NUMBER_OF_STRINGS;

    @Ignore
    public TablatureNotation notes = new TablatureNotation();
```

Figura 3.10 – Parte da implementação da classe Tablature

Além disso, foi implementado uma classe *TablatureFactory* utilizando o padrão de design *Factory*. Desta forma, a criação de um objeto de tablatura é abstraída. Note que, na classe *Tablature*, as variáveis do cabeçalho (*title*, *subtitle*, *artist*, *...)* não são inicializadas. Através do *TablatureFactory*, é possível instanciar um objeto com estes valores corretos de acordo com a linguagem do usuário (utilizando a solução do próprio Android). É possível também utilizar valores iniciais configurados pelo próprio usuário (por exemplo, *artist* sempre será o nome do músico), porém esta funcionalidade foi deixada para uma versão futura do aplicativo.

```
public class TablatureFactory {
    private TablatureFactory(){}

    public static Tablature getTablature(Context context) {
        Tablature tablature = new Tablature();

        tablature.setBpm(Integer.valueOf(context.getString(R.string.tablature_default_tempo_value)));
        tablature.title = context.getString(R.string.tablature_title);
        tablature.subtitle = context.getString(R.string.tablature_subtitle);
        tablature.artist = context.getString(R.string.tablature_artist);
        tablature.comment = context.getString(R.string.tablature_comment);
        tablature.tuning = context.getString(R.string.tablature_standard_tuning);
        tablature.setBpm(Tablature.DEFAULT_BPM);
        tablature.setCapo(Tablature.DEFAULT_CAPO_FRET);
        tablature.setNumberOfStrings(Integer.valueOf(context.getString(R.string.tablature_default_number_of_strings_value)));
        return tablature;
    }
}
```

Figura 3.11 – Classe TablatureFactory

A renderização da tablatura é feita por outra classe chamada *TablatureView*. Ela é um componente customizado que herda da classe *LinearLayout* do Android. Ela, em si, desenha o cabeçalho da tablatura e delega a renderização das notas da tablatura para a classe *TablatureNotesView* (ver Figura 3.12), que herda da classe *View* do Android. Note que a classe *View* é a classe base para todos os componentes de interface do Android.

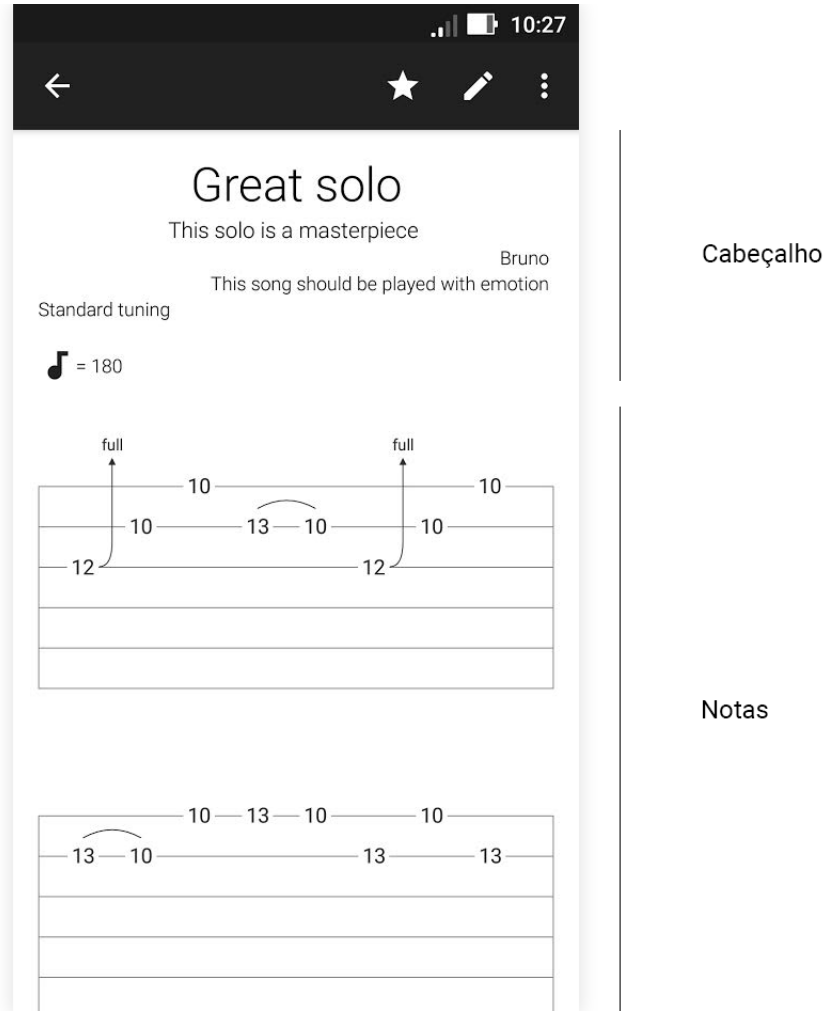


Figura 3.12 – Renderização da tablatura

É interessante ressaltar que a partir do *Model* da tablatura, é possível desenvolver classes que a desenhem de uma outra maneira qualquer. Por exemplo, é possível desenhar a tablatura inteiramente em forma de texto utilizando os caracteres da tabela ASCII, ou até desenhá-la continuamente, sem quebra de linha.

3.2.3 – Interação Com a Tablatura

A interação com a tablatura pode ser dividida em algumas partes:

- Seleção de um elemento na tablatura
- Inserção e remoção de notas
- Inserção e remoção de efeitos

A **seleção de um elemento na tablatura** é realizada através de um cursor, que indica o elemento atual a ser editado na tablatura. Para mover o cursor, foi criado um

componente chamado *JoystickButtons*, que chama uma *callback* sempre que o usuário realiza um clique válido (por exemplo, um clique na região central do componente não é considerado um clique válido).

```
public class Cursor {
    private int string = 0;
    private int column = 0;
    private Paint backgroundPaint;
    private boolean hidden = false;
```

Figura 3.13 – Parte da implementação da classe Cursor

```
public interface OnJoystickClickListener {
    enum Arrow {LEFT, UP, RIGHT, DOWN};

    void onJoystickClick(Arrow arrow);
}
```

Figura 3.14 – Interface de clique válido da classe JoystickButtons

É possível também interagir diretamente com um elemento ao clicar nele (ou próximo a ele). Para isso, existe um método que verifica que elemento o usuário deseja selecionar a partir da posição do evento de clique na tablatura.

A **inserção e remoção de notas** foi feita criando o componente *Fretboard*. Nele, são dispostas todas as notas (0 a 9), além da nota especial *x*, um botão para apagar (semelhante ao *backspace* do teclado) e um botão de inserir (se estiver selecionado, ele insere uma nova coluna na posição do cursor; se o usuário sair daquela coluna e ela estiver vazia, a coluna é apagada). Ao clicar em algum desses botões, o componente dispara uma *callback* e passa como parâmetro que botão foi clicado. Desta forma, uma outra classe responsável por interagir com a tablatura pode lidar com o evento e tomar uma ação.

```
public class Fretboard extends LinearLayout {

    OnFretboardActionListener onFretboardActionListener;
    private ToggleButton btnInsert;

    public enum FretboardKeys {
        FRET_0, FRET_1, FRET_2, FRET_3, FRET_4, FRET_5, FRET_6, FRET_7, FRET_8, FRET_9,
        FRET_X, BACKSPACE, INSERT;
    }
    public interface OnFretboardActionListener {

        void onKey(FretboardKeys key, boolean fromUser);
    }
```

Figura 3.15 – Parte da implementação da Fretboard

Para realizar a inserção de um elemento, uma coluna de inserção é injetada ao final da tablatura. Após o usuário digitar uma nota e deslocar o cursor para a direita, uma outra coluna de inserção é injetada. Caso o usuário adicione um efeito mas não adicione uma nota correspondente, ao deslocar o cursor, os elementos inválidos são removidos. Logo, não é possível ter um efeito desassociado de uma nota. O fluxo de inserção pode ser observado na Figura 3.16

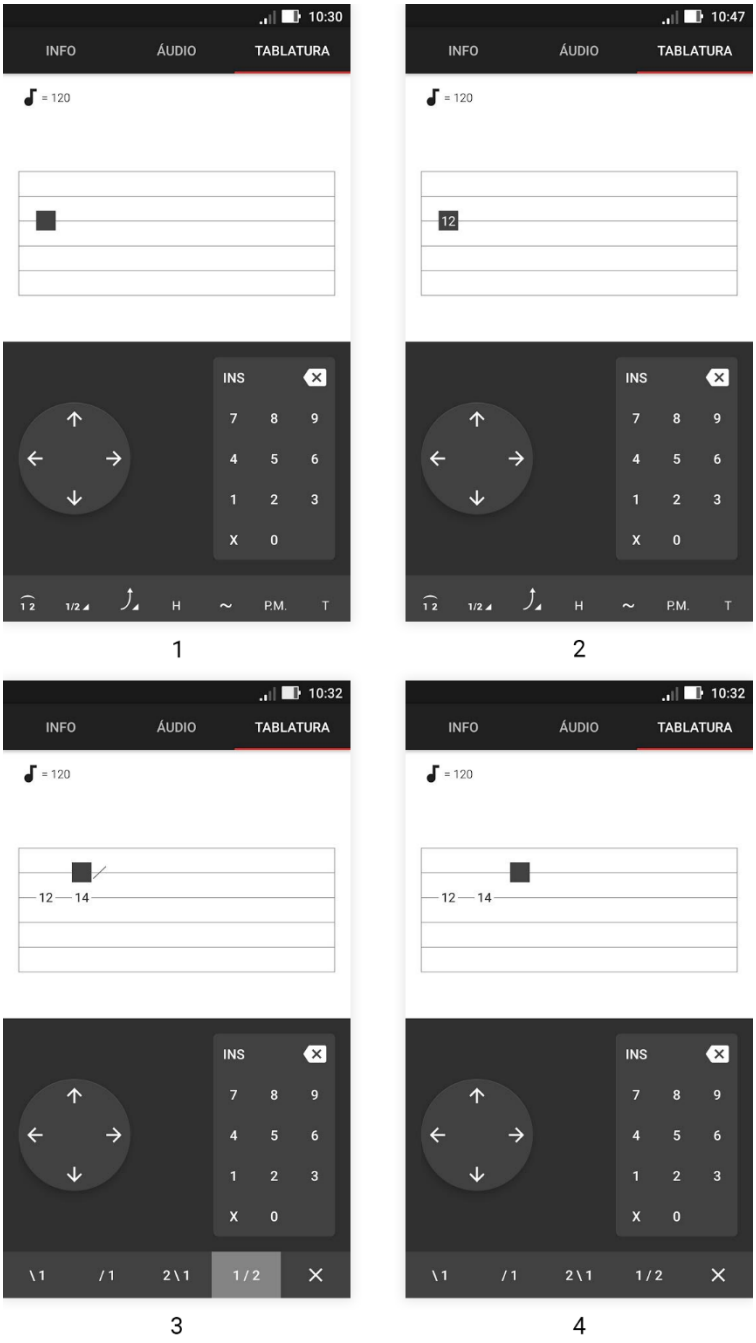


Figura 3.16 – Um possível fluxo de inserção. 1) A coluna de inserção está na posição 0. 2) O usuário digita o número 1 e depois o 2 na Fretboard. 3) O usuário move o cursor para a direita, digita os números 1 e 4, move o cursor para cima e insere um efeito. No momento que o cursor move para a direita, uma coluna de inserção é adicionada na posição 1. 4) O usuário move o cursor para a direita. Neste momento, uma coluna de inserção é adicionada na posição 3 e o efeito inválido é removido da coluna anterior

No caso da funcionalidade de inserção *INS*, a coluna de inserção é adicionada sempre na posição do cursor. Quando o usuário se desloca, uma nova coluna é adicionada e, se a coluna antiga estiver vazia, ela é removida, como mostra a Figura 3.17

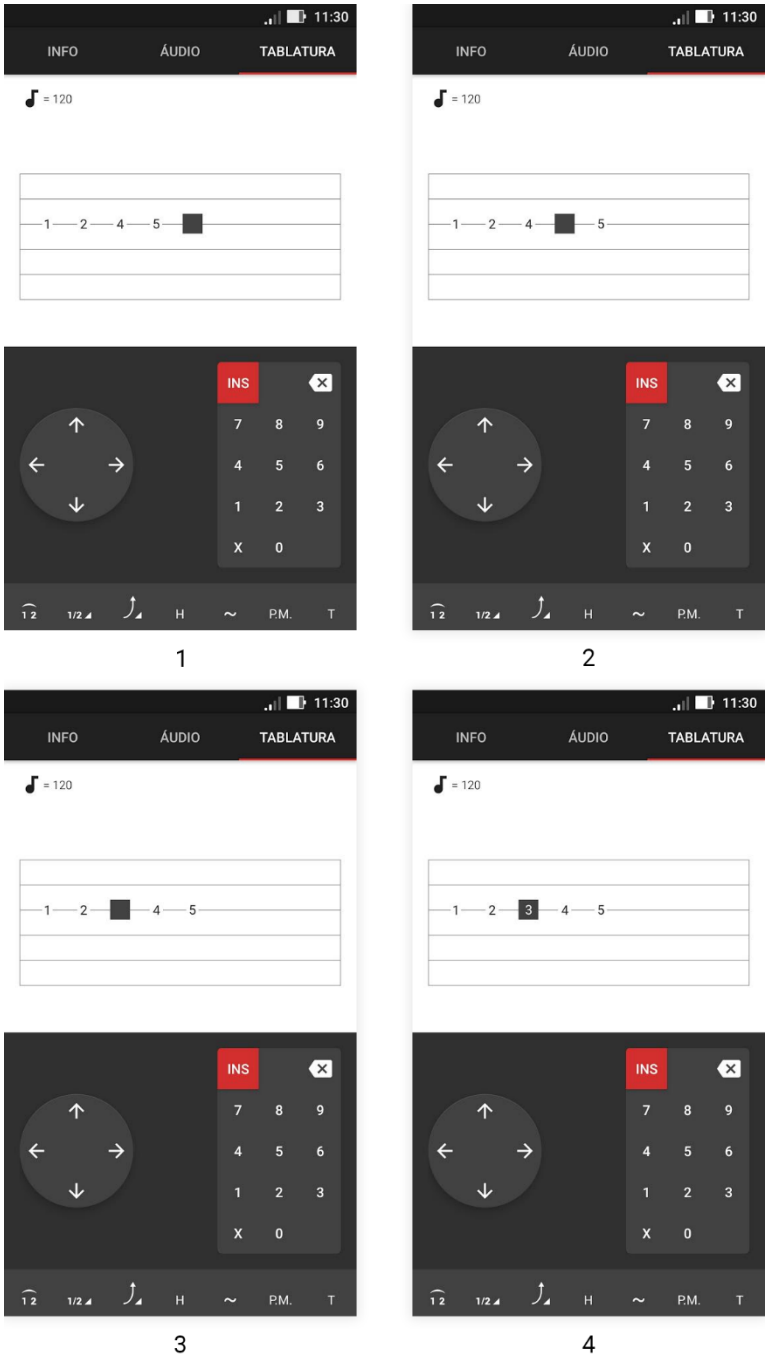


Figura 3.17 – Um possível fluxo de inserção com o botão INS selecionado. 1) O usuário seleciona o botão INS. 2) O usuário move o cursor para a esquerda. Neste momento, a coluna de inserção anterior é apagada e uma nova é adicionada na posição 3. 3) O usuário move o cursor para a esquerda. Novamente, a coluna de inserção anterior é apagada e uma nova é adicionada na posição 2. 4) O usuário digita o número 3.

O processo de remoção de notas é similar ao processo de apagar um caractere utilizando um teclado convencional. Se o usuário estiver digitando um elemento e pressionar o botão de apagar, ele apagará a última nota do elemento. Caso o usuário

apague uma nota e a coluna esteja vazia, no momento em que o cursor for movido horizontalmente, a coluna vazia será apagada. Este comportamento foi implementado para que não existam colunas vazias na tablatura. Um fluxo do processo pode ser visto na Figura 3.18

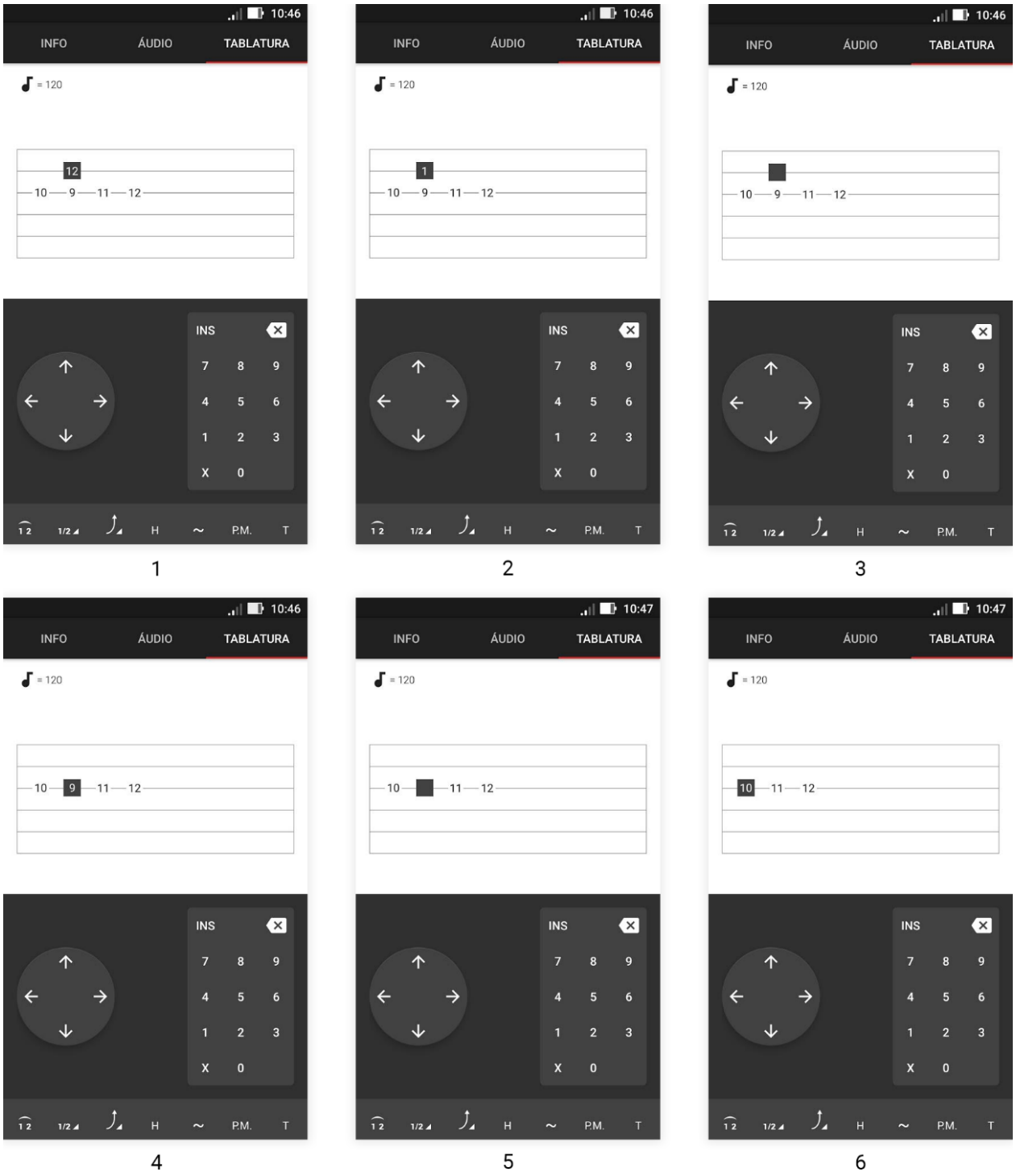


Figura 3.18 – Um possível fluxo de apagar. 1) O usuário digita os números 1 e 2 na coluna 1. 2) O usuário pressiona o botão de apagar. O número 2 é apagado. 3) O usuário pressiona o botão de apagar. O número 1é apagado. 4) O usuário move o cursor para

baixo. 5) O usuário pressiona o botão de apagar. Toda a nota é apagada. 6) O usuário move o cursor para a esquerda. Neste momento, a coluna vazia é apagada

A **inserção e remoção de efeitos** foi implementada criando os componentes *BottomBar* e *BendIntensityBar*. O *BottomBar* é uma barra de efeitos localizada de forma fixa na parte inferior da tela de edição de tablatura. Ela insere o efeito desejado no elemento de tablatura selecionado pelo cursor. Como nem todos os efeitos cabem nesse espaço, foi utilizado uma técnica de agrupar os elementos semelhantes (como todos os efeitos de *slider* por exemplo) e mostrá-los no clique. Desta forma, o usuário realiza um clique para ver os efeitos do grupo e mais um clique para selecionar. No caso do *bend*, além desta técnica, foi necessária a inserção de uma barra vertical para a escolha da intensidade do *bend*. Esta barra só é mostrada no momento em que o usuário clica na variante do *bend* desejada. Ao clicar na intensidade, o efeito é aplicado à nota atual, como mostra a Figura 3.19.

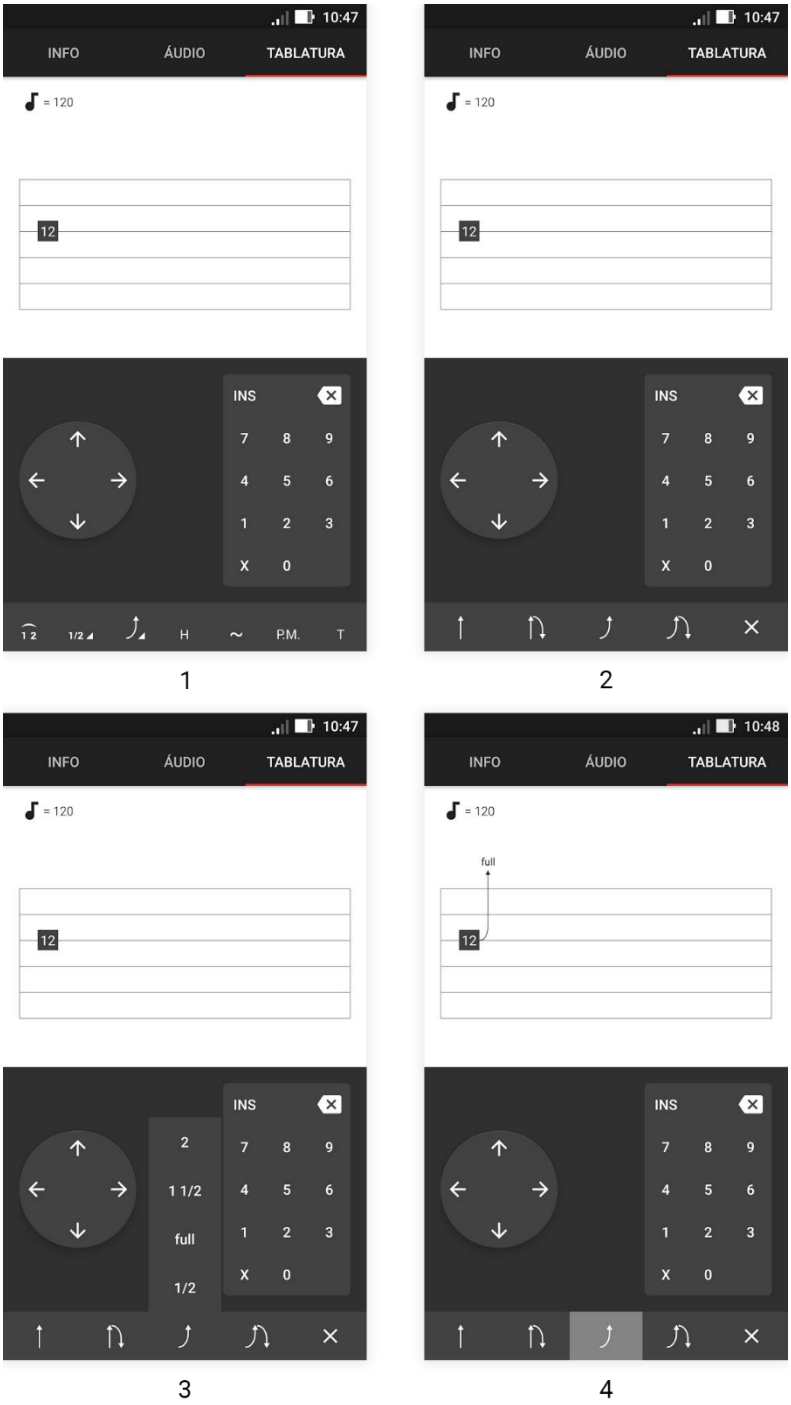


Figura 3.19 – Fluxo de inserção de um bend com intensidade de 1 tom. 1) Tela inicial. O usuário já digitou o número 12. 2) O usuário seleciona a opção de bend na barra inferior. 3) O usuário seleciona o tipo de bend. Neste momento, uma barra vertical com a intensidade do bend é mostrada. 4) O usuário seleciona a intensidade. A barra vertical é escondida e o efeito é aplicado. Note que bend aplicado à nota é marcado para facilitar sua localização

```
OnBottomBarActionListener onBottomBarActionListener;
OnBendBarCloseListener onBendBarCloseListener;

ArrayList<View> activeButtons = new ArrayList<>();

public enum BottomBarKeys {
    LEGATO, PRE_SLIDE_UP, PRE_SLIDE_DOWN, SLIDE_UP, SLIDE_DOWN, PRE_BEND, PRE_BEND_RELEASE, BEND,
    BEND_RELEASE, HARMONIC, VIBRATO, PALM_MUTE, TAPPING;
}

public interface OnBottomBarActionListener {

    void onKey(BottomBarKeys key, View button);
}

public interface OnBendBarCloseListener {

    void onClose();
}
```

Figura 3.20 – Parte da implementação da BottomBar

```
public class BendIntensityBar extends LinearLayout {

    OnBendIntensityChangeListener onBendIntensityChangeListener = null;
    Button activeButton;
    Button halfButton;
    Button oneButton;
    Button oneAndAHalfButton;
    Button twoButton;

    public interface OnBendIntensityChangeListener {

        void onChange(TablatureNotation.BendIntensity intensity);
    }
```

Figura 3.21 – Parte da implementação da BendIntensityBar

3.2.4 – Áudio

O áudio pode ser dividido nas seguintes partes:

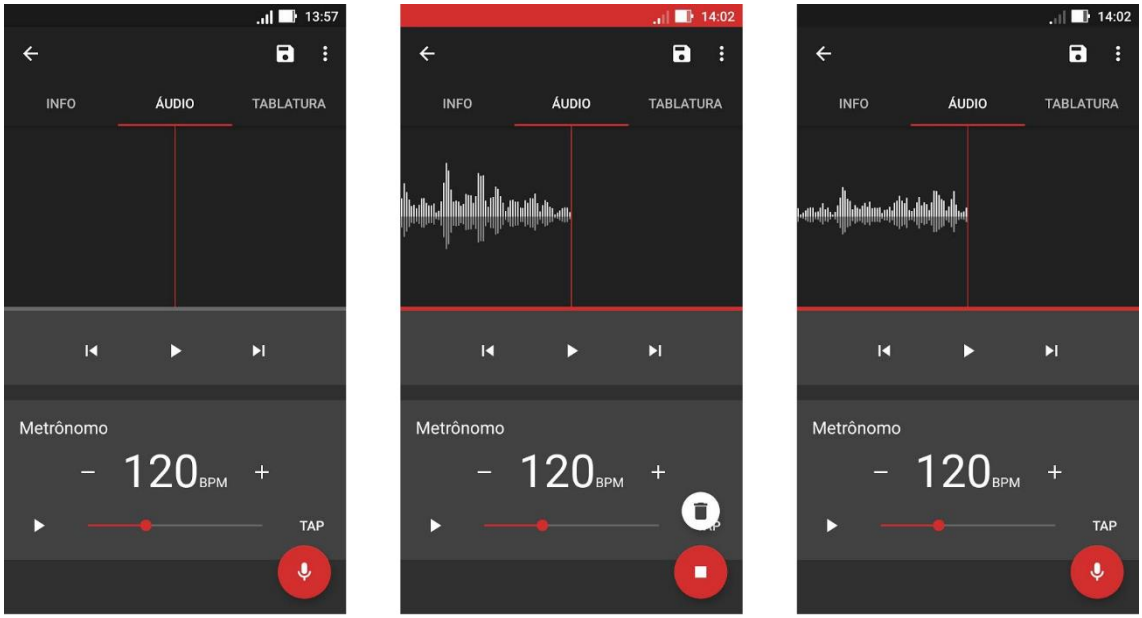
- Gravação
- Reprodução
- Renderização

A **gravação** é feita em uma *thread* própria, pois se fosse feita na mesma *thread* da UI, a mesma seria bloqueada e pararia de processar os eventos de entrada do usuário.

Para este projeto, é necessário que o áudio gravado possa ser parado e retornado a qualquer momento. Logo, deve ser possível fazer uma nova gravação e concatená-la com uma gravação já existente. A classe *AudioRecord* [15] do Android não possui esta funcionalidade, logo, foi desenvolvida uma solução própria.

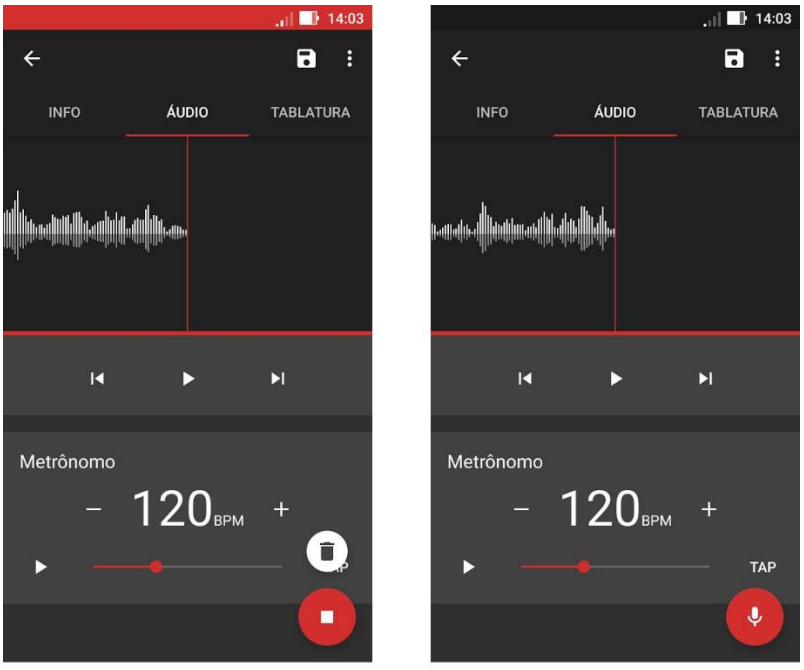
Para realizar a gravação, foi criada a classe *RecordingThread*, responsável por configurar uma instância da classe *AudioRecord*, capturar o áudio para um *buffer* e chamar uma *callback* informando que o áudio foi capturado. Uma outra classe chamada *AudioRecorder* instancia a *RecordingThread* e realiza a escrita do *buffer* de áudio para um arquivo utilizando a classe *FileOutputStream*. Desta maneira, só é mantido em memória o *buffer* da gravação, enquanto que o áudio completo é mantido em disco. A classe *AudioRecorder* implementa a funcionalidade de "parar" uma gravação e retornar de onde parou da seguinte forma: Ao realizar uma gravação, o áudio é armazenado em disco como em sua forma pura (*raw)*, sem nenhuma codificação nem cabeçalho. A gravação seguinte é feita da mesma forma, porém, ao invés de criar um arquivo novo, o mesmo é utilizado como parâmetro para o *FileOutputStream*.

No app, também é possível descartar a última gravação. Ou seja, se o usuário realizou *n* gravações, e durante a gravação *n + 1* ele decide que ela deve ser descartada, a gravação final será composta de *n* gravações. Isto é implementado guardando a posição da gravação *n + 1* no arquivo de áudio e. Ao ser descartada, o arquivo de áudio é copiado da posição 0 até o início da gravação *n + 1* e o arquivo anterior é apagado. O processo de gravar é representado pela Figura 3.22 e o de descartar, pela Figura 3.23. Caso o usuário realize uma gravação grande e deseje descartar, uma janela de confirmação é apresentada para confirmar a ação (ver Figura 3.24)



1 2 3

Figura 3.22 – Fluxo de gravação. 1) Tela inicial. 2) O usuário pressiona o botão de gravação. A gravação começa a ser feita 3) O usuário pressiona o botão de parar. A gravação é parada.



1 2

Figura 3.23 – Processo de descartar o último áudio. 1) O usuário está gravando um áudio. 2) O usuário pressiona o botão de descartar. A última gravação é descartada.

Figura 3.24 – Janela mostrada ao usuário quando ele tenta descartar uma gravação grande.

Todas as músicas do usuário ficam armazenadas em uma pasta criada pelo aplicativo; ao editar uma música já existente, o arquivo de áudio original é copiado para uma pasta temporária, onde o usuário possa fazer as modificações que desejar. Desta forma, não há risco de comprometer o arquivo de áudio original. Quando o usuário salva a música, o arquivo da pasta temporária é copiado para o local correto. Vale ressaltar que só é possível utilizar um arquivo de áudio temporário por vez, logo quando o usuário edita ou cria uma outra música, o arquivo de áudio na pasta temporária é apagado e um novo arquivo é copiado ou criado.

A **reprodução**, assim como a gravação, deve ser feita em uma *thread* separada para não comprometer a UI. A classe *PlaybackThread* foi criada para reproduzir um arquivo gerado pela *RecordingThread*. Ela faz a leitura do arquivo de áudio utilizando a classe *FileInputStream* e, sempre que o *buffer* de leitura está pronto, ela o escreve em uma instância da classe *AudioTrack* [16].

Para lidar com a interação do usuário com respeito à reprodução de áudio, foi criado um componente chamado *Player*. Ele cria uma instância da *PlaybackThread*, provê funções de *play*, *pause*, *stop*, ir para o início e final da música, habilitar e

desabilitar a reprodução, além de *callbacks* que informam o estado da reprodução e outras funcionalidades.

```
public interface OnPlayerStateChangeListener {
    void onPlay();
    void onPause();
    void onStop();
    /**
     * @param progress The track position
     * @param fromUser True if it was originated from the user
     */
    void onProgress(int progress, boolean fromUser);
    void onFinish();
    void onPrevious();
    void onNext();
}
```

Figura 3.25 – Interface que expõe o estado do Player

O processo de **renderização** é responsável por mostrar para o usuário um "gráfico" de barras correspondente ao áudio gravado. Para isso, deve-se utilizar uma amostra bem menor do arquivo de áudio, pois não é possível, nem desejado, plotar todas as amostras de áudio gravado. Como a visualização é renderizada em tempo real, deve-se coletar amostras conforme o áudio é gravado. A classe *AudioRecorder* implementa esta funcionalidade a partir de uma *callback*. Em um determinado intervalo de tempo, ela chama a *callback* passando como parâmetro uma amostra aleatória do áudio gravado naquele intervalo de tempo. Assim, temos uma amostra pequena, porém significativa do áudio gravado. Vale ressaltar que esta amostra é armazenada para que a renderização seja sempre a mesma.

```
private Runnable notifyRunnable = new Runnable() {
    @Override
    public void run() {
        if (isRecording()) // if we are already recording
        {
            // get the current amplitude
            recordNotificationListener.onRecordNotification(randomSample);

            // update in 40 milliseconds
            handler.postDelayed(this, NOTIFY_INTERVAL);
        }
    }
};
```

Figura 3.26 – Notificação de amostras de áudio

A renderização em si é feita de duas maneiras. Como o usuário é capaz de mover a renderização horizontalmente para ajustar a posição do áudio, ele deve estar completamente renderizado; quando o áudio está sendo gravado, qualquer interação com o *Player* e com o gráfico é desabilitada e somente as últimas amostras gravadas são mostradas na tela (ver Figura 3.27).

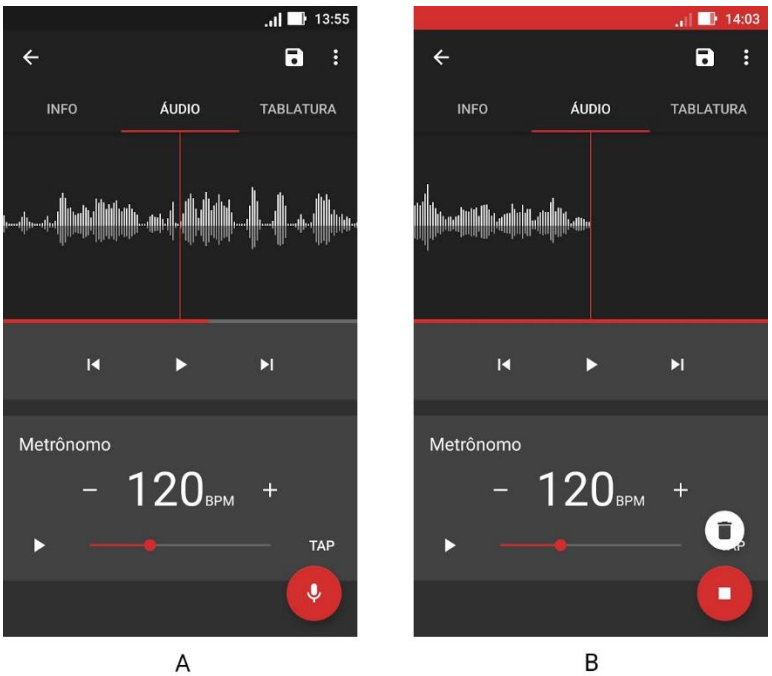


Figura 3.27 – Comparação entre um áudio já gravado e uma gravação. A) Um áudio gravado. O usuário pode interagir com ele e movimentá-lo horizontalmente. B) Áudio durante uma gravação. Toda a interação do usuário com o mesmo é desabilitada

Assim, temos que, quando uma gravação não está sendo feita, todo o áudio é renderizado de uma só vez. Para prover o deslizamento fluido e manter o início ou final

do áudio no centro da tela, o componente *AudioView* adiciona dois espaçamentos, um à esquerda do *AudioViewContent* e outro à direita. Além disso, ele encapsula estes três componentes em uma *View* de *scroll* (ver Figura 3.28 e Figura 3.29)

```
<com.brunocalou.songnote.utils.ObservableHorizontalScrollView
    android:id="@+id/observable_scroll_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:scrollbars="none">

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:orientation="horizontal">

        <Space
            android:id="@+id/left_space"
            android:layout_width="1dp"
            android:layout_height="match_parent" />

        <com.brunocalou.songnote.components.audioeditor.AudioViewContent
            android:id="@+id/audio_view_content"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:background="@color/colorPrimary"/>

        <Space
            android:id="@+id/right_space"
            android:layout_width="1dp"
            android:layout_height="match_parent" />
    </LinearLayout>
</com.brunocalou.songnote.utils.ObservableHorizontalScrollView>
```

Figura 3.28 – Parte do arquivo audio_view.xml

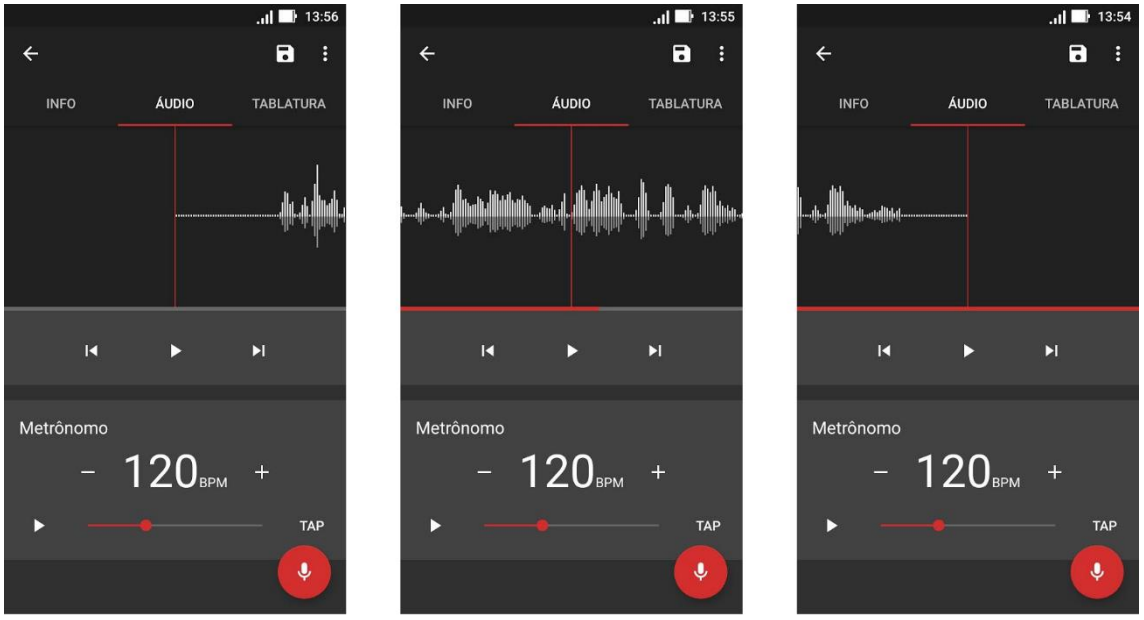


Figura 3.29 – Visualização do áudio. A) Início do áudio. O usuário só pode deslocá-lo para a esquerda. B) Posição intermediária. O usuário pode deslocá-lo para a direita e para a esquerda. C) Final do áudio. O usuário só pode deslocá-lo para a direita

Quando uma gravação está sendo feita, seria altamente custoso redesenhar todas as amostras de áudio a cada vez que uma amostra fosse recolhida. Se isso fosse feito, a UI se tornaria extremamente lenta após alguns segundos de gravação. Para resolver este problema, o componente *AudioViewContent* possui dois modos de operação. Um, já visto, é capaz de renderizar todas as amostras de áudio. Neste processo, ele se redimensiona para que todas as amostras possam ser desenhadas; o segundo modo renderiza somente um conjunto de amostras; neste caso, o componente possui um tamanho fixo (determinado pelo *AudioView*) e desenha somente as últimas amostras. O algoritmo desenvolvido para desenhar o áudio pode ser visto na Figura 3.30

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    if (drawOnAudioWindowOnly) {
        int firstIndex = (audio.size() - 1) - getIndex(canvas.getWidth());
        if (firstIndex < 0) firstIndex = 0;
        drawAudioBars(canvas, firstIndex, audio.size() - 1, audioPaint, audioReflectionPaint);
    } else {
        drawAudioBars(canvas, 0, audio.size() - 1, audioPaint, audioReflectionPaint);
    }
}
private void drawAudioBars(Canvas canvas, int firstIndex, int lastIndex, Paint audioPaint, Paint reflectionPaint) {
    if (audio.size() > 0) {
        average.clear();
        int averageStartSamples = average.getSize();

        if (firstIndex - average.getSize() < 0) {
            averageStartSamples = firstIndex;
        }
        int startIndex = firstIndex - averageStartSamples;
        if (startIndex < 0) startIndex = 0;

        for (int i = startIndex; i < firstIndex; i += 1) {
            average.add(Math.abs(audio.get(i)));
        }

        int bottom = canvas.getHeight() / 2 + baseLineAdjust;

        for (int index = firstIndex; index < lastIndex; index += 1) {
            average.add(Math.abs(audio.get(index)));
            int sample = (int) average.getAverage();
            int audioBarHeight = (int) Math.max(sample * yScale, minAudioBarHeight);
            int left = canvas.getWidth() - (int) getLeftRect(lastIndex - index);
            int top = bottom - audioBarHeight;
            int right = (int) (left + audioBarWidth);

            if (left < 0) left = 0;
            canvas.drawRect(left, top, right, bottom, audioPaint);
```

Figura 3.30 – Parte principal do algoritmo para realizar o desenho do áudio

Note que no algoritmo é utilizado um componente para fazer a média móvel do módulo das amostras antes de renderizar. Isto é feito para garantir que o desenho feito não contenha muitas mudanças bruscas entre uma barra de áudio e outra. Como as amostras são aleatoriamente escolhidas, comumente são escolhidos valores muito altos e muito baixos, o que tornam o desenho desproporcional. Uma comparação entre um áudio renderizado com e sem média móvel pode ser visto na Figura 3.31

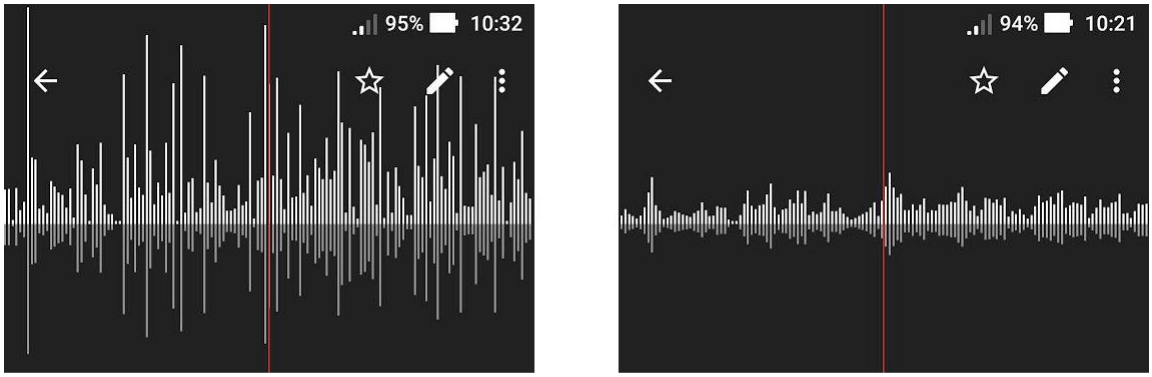


Figura 3.31 – Comparação entre renderização de áudio. A) Áudio renderizado sem média móvel. B) O mesmo áudio renderizado com média móvel

A média móvel utilizada neste projeto foi feita utilizando um *buffer* circular e um vetor de pesos. Os pesos foram atribuídos utilizando a fórmula

$$peso_i = \frac{N - i}{\sum_i^N i}; i \geq 0; i < N$$

onde N é o tamanho da janela alocada (tamanho do *buffer*). O denominador da fórmula é o somatório de i a N, logo, na implementação, foi utilizada a fórmula da soma da progressão aritmética para este caso específico

$$Sn = \frac{(1 + n) * n}{2}$$

A implementação completa pode ser vista no Apêndice 1

3.2.5 – Metrônomo

O metrônomo é um instrumento de suporte ao músico, cuja funcionalidade é indicar o andamento musical através de pulsos sonoros. Um metrônomo simples foi incorporado nesta versão do projeto para auxiliar o usuário durante uma gravação ou até durante o treinamento de uma música.

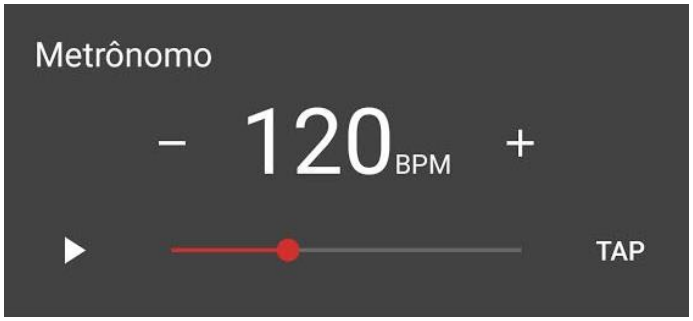


Figura 3.32 – Metrônomo implementado no projeto

Sua implementação foi feita a partir de duas classes, uma chamada *MetronomeSound*, capaz de reproduzir o som do metrônomo em uma *thread* separada, e outra chamada *Metronome*, responsável por interagir com o usuário e controlar os parâmetros do metrônomo.

Existem dois áudios para o metrônomo, nomeados como *tick* e *tock*. Ao instanciar a classe *MetronomeSound*, esses sons são carregados em uma *SoundPool* [17] que os mantém em memória. Quando o usuário inicia o metrônomo, uma *thread* é iniciada e executa os áudios de acordo com as configurações.

A classe *Metronome* recebe a quantidade de batidas por minuto (*BPM*) através de um *slider* horizontal e de dois botões para ajuste fino. Porém, nem sempre é fácil o usuário saber exatamente quantos *BPM* ele deseja, logo foi adicionado um botão chamado *TAP* configura o *BPM* a partir de toques consecutivos do usuário. Ou seja, se o usuário clicar várias vezes no botão, o intervalo entre os cliques irá determinar o *BPM*. Isto é feito dividindo-se um minuto pelo tempo entre duas batidas consecutivas, como pode ser visto na Figura 3.32. Note que pressionar um botão em intervalos de tempo regulares não é tão fácil, duas medidas consecutivas podem ser muito discrepantes. Para resolver este problema, foi utilizada uma média com pesos fixos, onde as medidas mais recentes têm peso maior do que as mais antigas. Assim, a medição do *BPM* consegue se estabilizar depois de poucos cliques.

```
tapButton.setOnClickListener(new OnClickListener() {
    long startTime = System.currentTimeMillis();
    long oneMinute = 60000L;
    int bpmHistory[] = {0,0,0};
    int weights[] = {1, 3, 6};
    int weightSum = 10;
    int bpmIdx = 0;

    @Override
    public void onClick(View view) {
        long now = System.currentTimeMillis();
        long delta = now - startTime;
        int bpm = (int) (oneMinute / delta);
        int average = 0;

        bpmHistory[bpmIdx] = bpm;
        bpmIdx += 1;
        if (bpmIdx == bpmHistory.length) bpmIdx = 0;

        for (int i = 0, length = bpmHistory.length; i < length; i += 1) {
            average += bpmHistory[i] * weights[i];
        }

        average /= weightSum;

        if (average > Tablature.MIN_BPM) {
            seekBar.setProgress(average - Tablature.MIN_BPM);
        }
        startTime = now;
    }
});
```

Figura 3.32 – Implementação do algoritmo para medir o *BPM*

3.2.6 – Informações e Etiquetas

Cada ideia musical é acompanhada de dados que servem para sua identificação e organização. Toda ideia pode possuir título, descrição, instrumento e etiquetas. Os campos título, descrição e instrumento são armazenados diretamente na classe *Idea* e são editados diretamente pelo usuário (ver Figura 3.33).

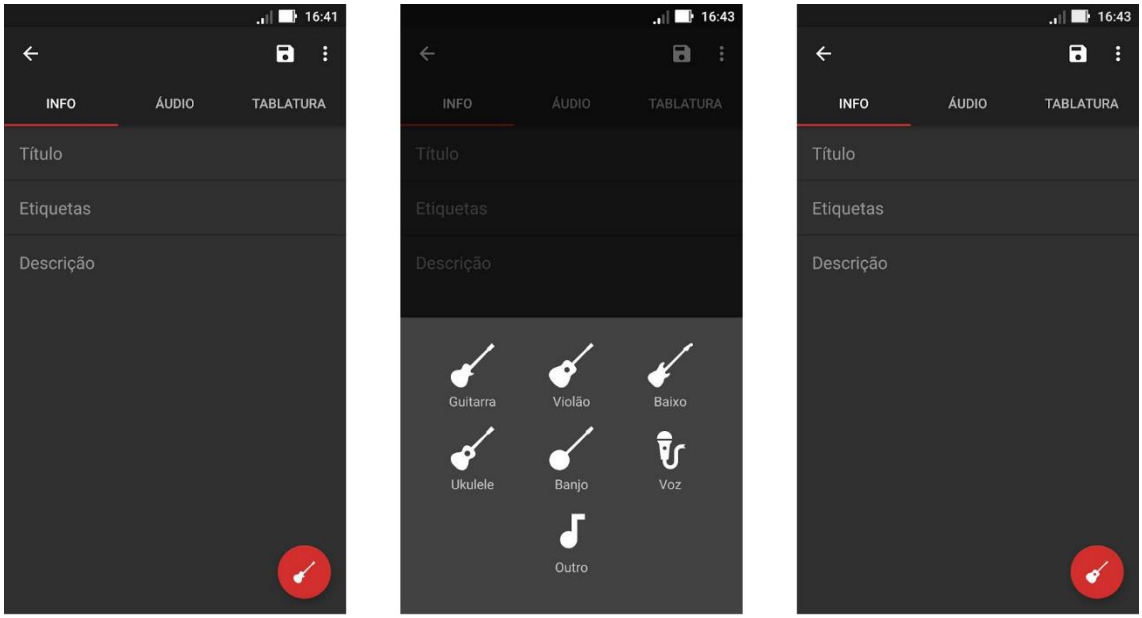


Figura 3.33 – Seleção de instrumento. 1) Usuário aperta o botão para mudar o instrumento. 2) Usuário seleciona o instrumento desejado. 3) Instrumento é alterado

As etiquetas são representadas pela classe *Label* e se relacionam de forma n:n com a classe *Idea*. Existem dois *DataModels* no projeto atualmente, um que lida com a classe *Idea* e outro, com a classe *Label*. A interface *ILabelDataModel* expõe métodos para lidar com as etiquetas e pode ser observado na Figura 3.34

```
interface ILabelDataModel {
    fun getLabels(): Flowable<RealmResults<Label>>
    fun updateLabel(label: Label, newName: String)
    fun deleteLabel(label: Label)
    fun getEventBus(): PublishSubject<StateChangeEvent<String>>
    fun addLabel(name: String)
    fun getLabel(name: String): Label?
    fun getLabels(names: List<String>): List<Label>
    fun getLabelById(id: String): Label?
    fun getErrorMessageResId(error: StateChangeError): Int
}
```

Figura 3.34 – Interface ILabelDataModel

Existem duas formas do usuário adicionar uma etiqueta. Ao editar uma ideia, existe um campo na aba *INFO* que representa as etiquetas associadas à ideia. Ao clicar neste campo, o usuário é levado para uma página de seleção contendo todas as etiquetas do projeto. Se ele desejar, pode adicionar uma etiqueta nova, conforme é mostrado na Figura 3.35. Nesta tela, não é possível editar nem apagar etiquetas. A segunda maneira é

feita a partir de uma tela própria para edição de etiquetas. Na tela inicial, no menu lateral, existe uma opção para editar etiquetas. Ao clicar, o usuário é direcionado para uma tela de edição de etiquetas, onde as operações de adicionar, renomear e apagar são fornecidas. Um fluxo de criação de etiqueta pode ser visto na Figura 3.36 e um apagando uma etiqueta está presente na Figura 3.37.

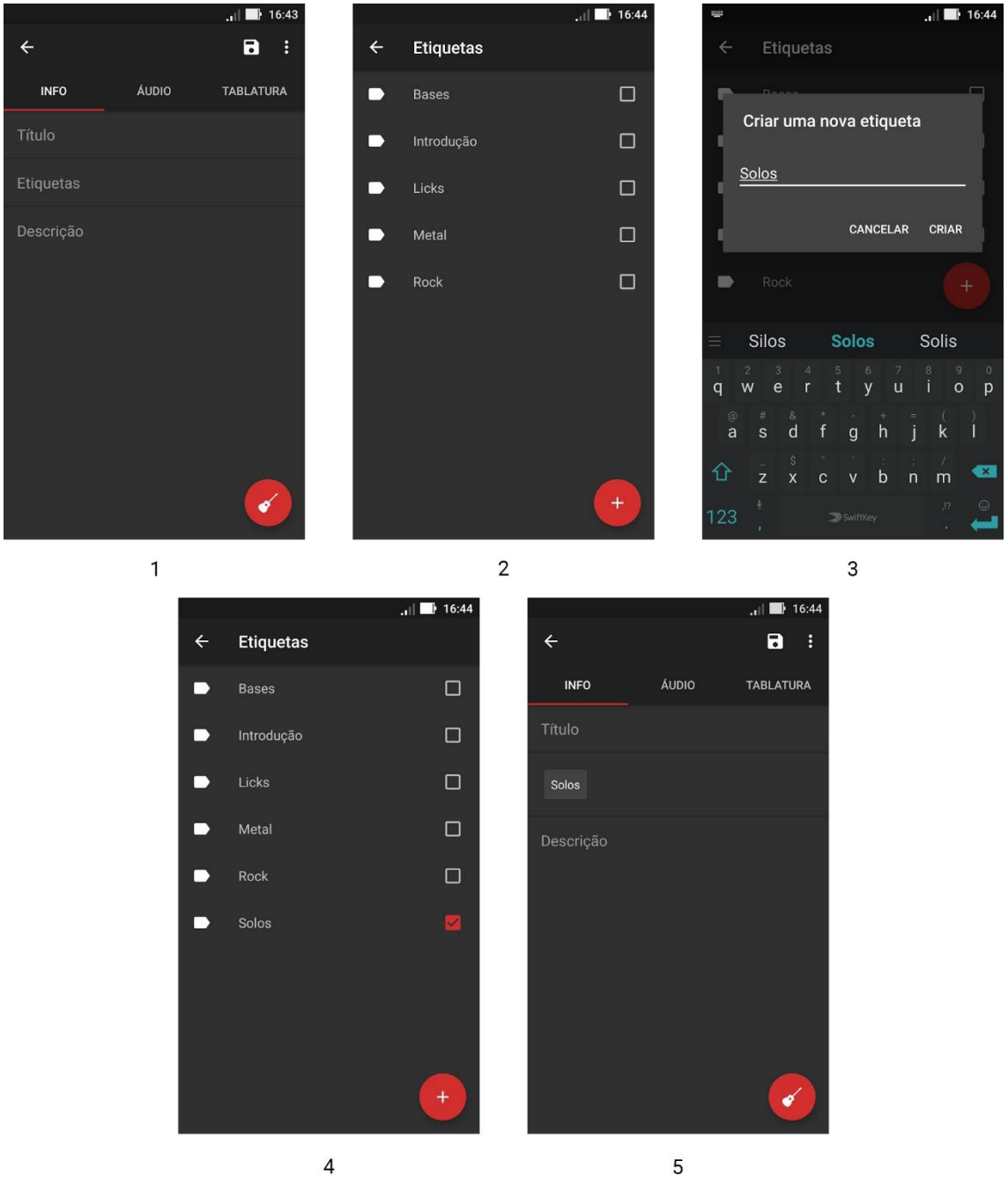


Figura 3.35 – Fluxo de inserção de etiqueta a partir da edição de ideia. 1) Tela inicial. 2) Ao clicar em Etiquetas, uma janela de seleção é mostrada para o usuário. 3) O usuário clica no botão de adicionar etiqueta. Uma janela é mostrada perguntando no nome da nova etiqueta. 4) O usuário cria a etiqueta e ela é selecionada automaticamente. 5) O usuário volta para a tela anterior. A etiqueta está aplicada na ideia

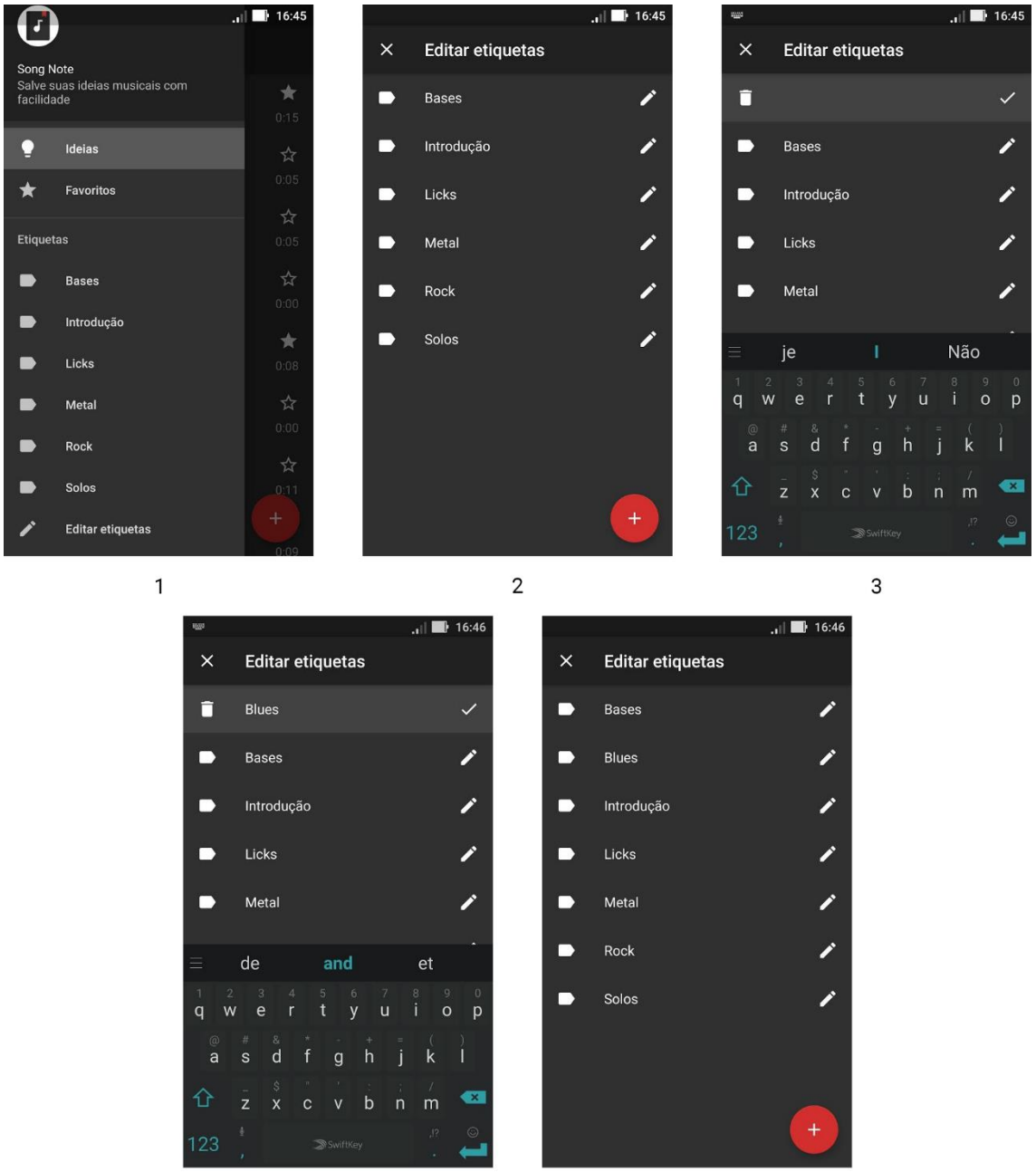


Figura 3.36 – Fluxo de criação de etiqueta a partir do editor. 1) A partir da tela inicial, o usuário abre o menu lateral. 2) Uma janela de edição de etiquetas é mostrada para o usuário. 3) O usuário pressiona o botão de adicionar etiqueta. Um campo para inserir o nome da etiqueta é mostrado na parte superior da lista. 4) O usuário escreve o nome desejado. 5) O usuário confirma o nome. A nova etiqueta é adicionada à lista

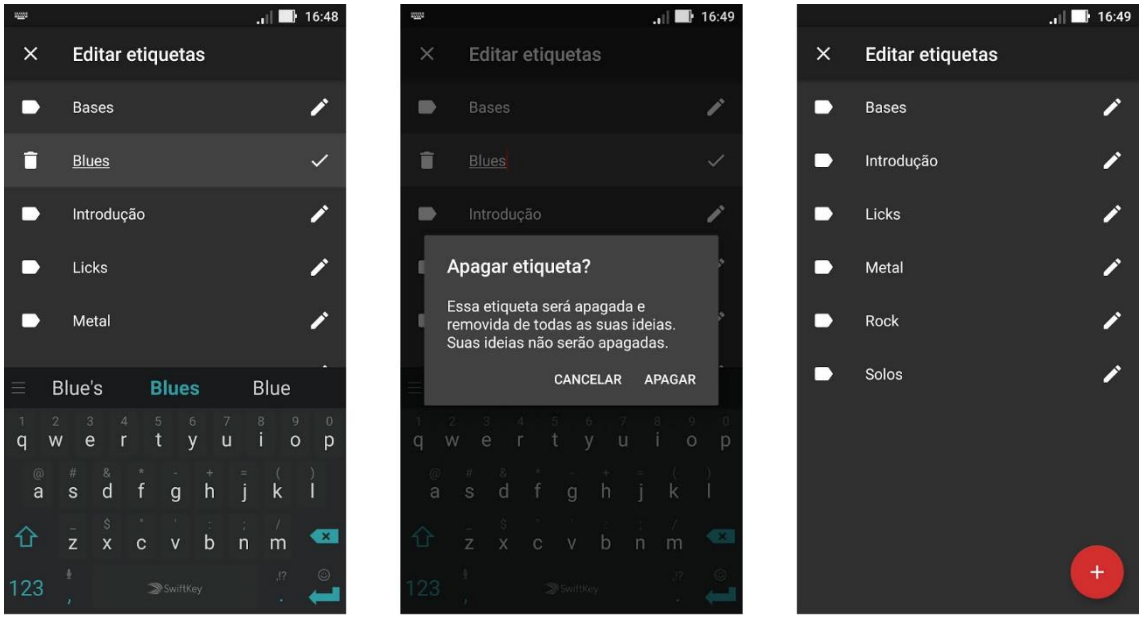
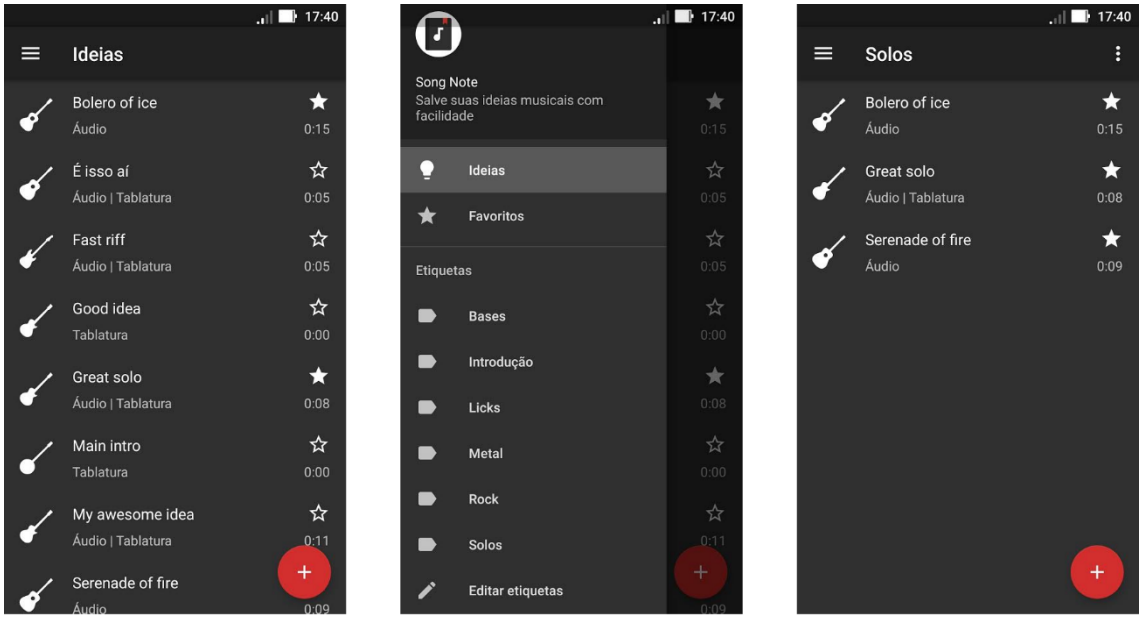


Figura 3.37 – Fluxo de apagar etiqueta a partir do editor. 1) Usuário seleciona a etiqueta que deseja apagar. 2) Usuário pressiona o botão de apagar. Um aviso é mostrado para o usuário para confirmar a ação. 3) A etiqueta é apagada

Uma etiqueta agrupa várias músicas com características semelhantes. Para ver músicas com uma mesma etiqueta, basta o usuário acessar a página da etiqueta pelo menu lateral na página inicial (ver Figura 3.38). Caso ele queira editar rapidamente o nome da etiqueta, basta ele acessar o menu superior. Uma janela será aberta para realizar a edição (ver Figura 3.39).



1 2 3

Figura 3.38 – Fluxo de ver ideias agrupadas por uma etiqueta. 1) Tela inicial. 2) Usuário abre o menu lateral. 3) Ao clicar na etiqueta desejada, a lista de ideias é atualizada

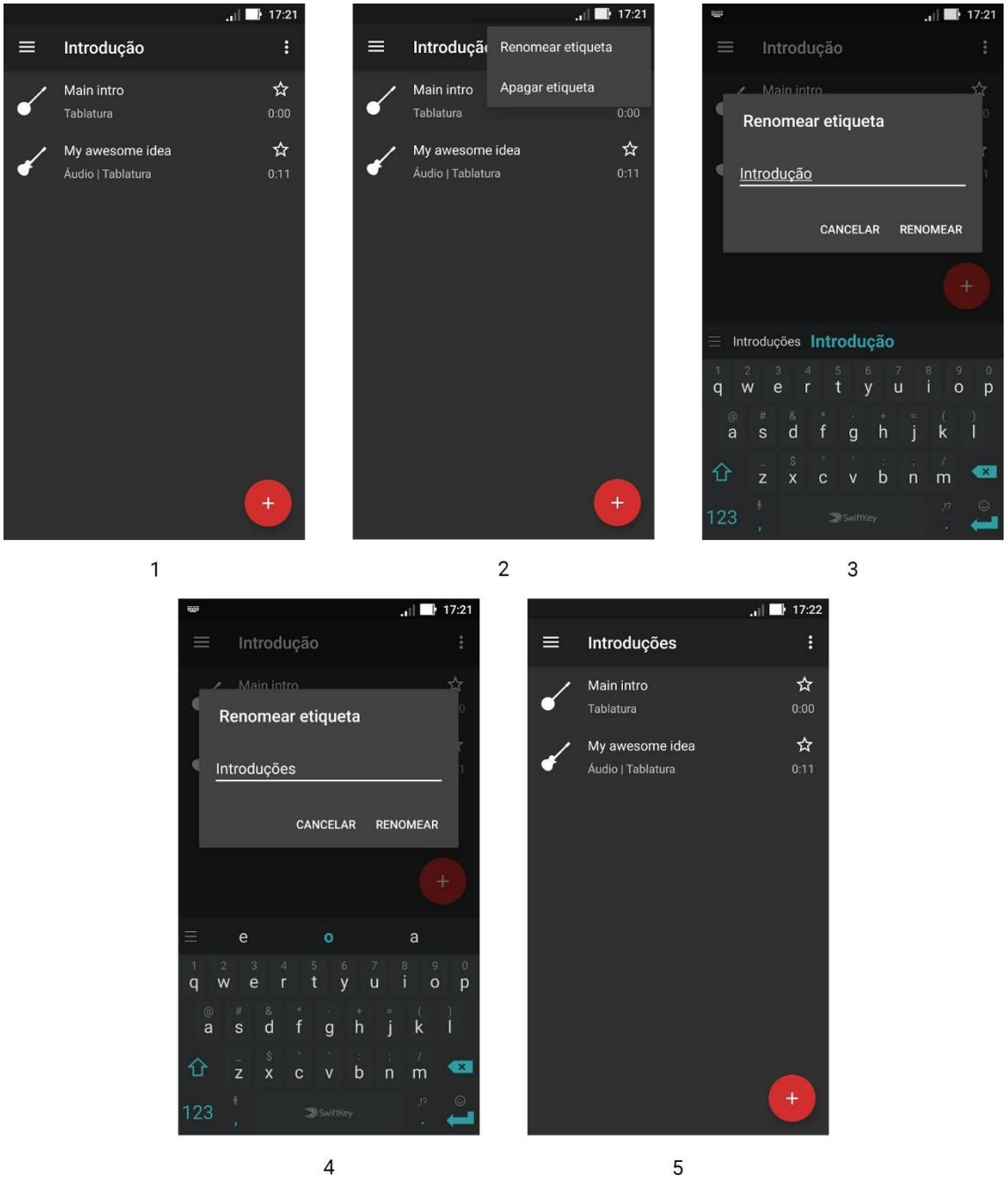


Figura 3.39 – Fluxo de renomear etiqueta a partir da tela da etiqueta. 1) Tela inicial. 2) Usuário clica no botão superior à direita e um menu é mostrado. 3) Usuário seleciona a opção de renomear etiqueta. 4) Usuário digita o novo nome da etiqueta. 5) Usuário confirma a ação. A etiqueta é renomeada

Vale ressaltar que existem restrições para criação de etiquetas. Caso o usuário adicione uma etiqueta inválida, uma mensagem é mostrada e a ação não é feita. (ver Figura 3.40)

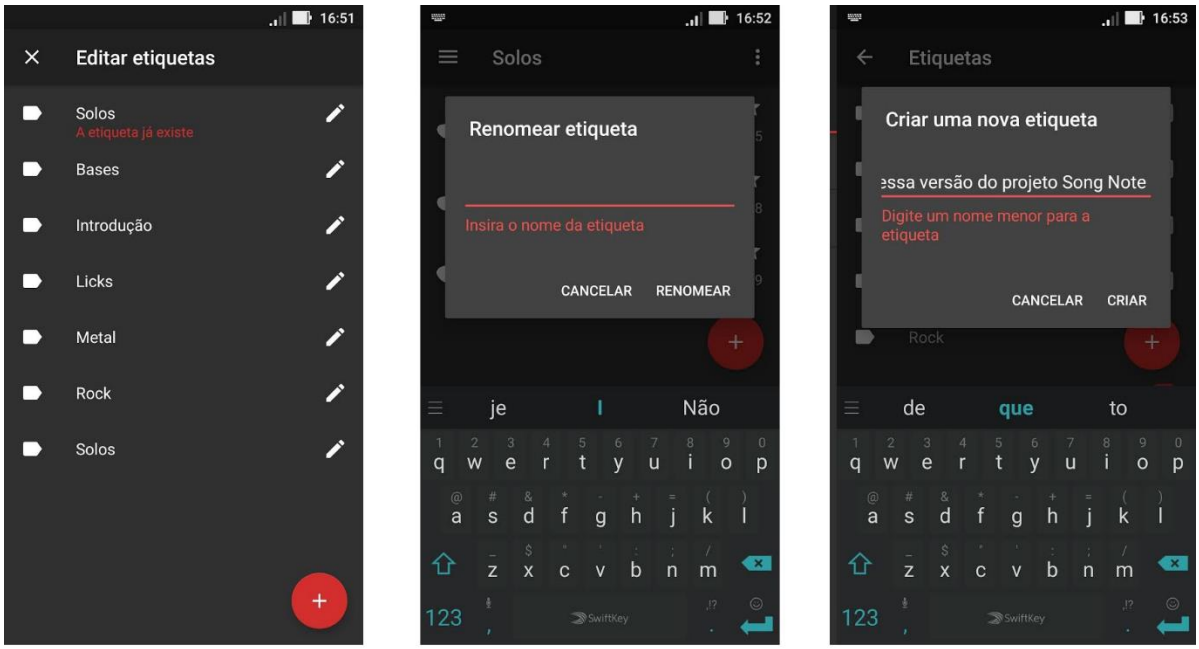


Figura 3.40 – Telas de etiquetas inválidas em lugares distintos do aplicativo

3.2.7 – Armazenamento

Na solução apresentada, existem três formas de armazenamento. Um banco de dados local armazena as ideias e as etiquetas. O áudio gravado é armazenado diretamente na memória interna do dispositivo. A tablatura é armazenada com o formato JSON na memória interna do aparelho. A tablatura é carregada inteiramente em memória somente quando é necessário (quando necessita ser visualizada ou editada); o áudio é carregado aos poucos em forma de *stream* durante uma reprodução, logo ele nunca é carregado inteiramente na memória.

Para armazenar os dados, foi utilizado o Realm Database [18]. Diferente de outros bancos de dado, o Realm abstrai toda a manipulação de dados através de uma API. Todo dado que pode ser armazenado deve ser uma instância de classe que herde da classe *RealmObject*. Desta forma, o *Realm* é capaz de gerenciar e realizar as operações esperadas de um banco de dados de uma maneira fácil e rápida. Não é necessário escrever nenhuma query, pois o *Realm* provê uma API que abstrai toda e qualquer *query* (ver Figura 3.41).

```
public class Dog extends RealmObject {
  public String name;
  public int age;
}

Dog dog = new Dog();
dog.name = "Rex";
dog.age = 1;

Realm realm = Realm.getDefaultInstance();
realm.beginTransaction();
realm.copyToRealm(dog)
realm.commitTransaction();

RealmResults<Dog> pups = realm.where(Dog.class)
                              .lessThan("age", 2)
                              .findAll();
```

Figura 3.41 – Demonstração de operações usando Realm
Fonte: Realm: Create reactive mobile apps in a fraction of the time [18].

Utilizando o modelo MVVM, o uso do *Realm* fica muito bem definido. As classes principais que representam o *Model* da aplicação são classes que herdam de *RealmObject* (salvo algumas exceções). A camada *DataModel* é a única que gerencia os dados, logo ela é a única que realiza operações utilizando a API do *Realm*. Na Figura 3.42, é possível observar um método da classe *IdeaDataModel* que implementa a interface *IIdeaDataModel*. Neste método, é feito uma busca utilizando os métodos providos pelo *Realm* e é retornado um *stream* de dados de forma assíncrona a partir da classe *Flowable* do RxJava. Note que o *Realm* possui uma boa integração com o RxJava.

```
override fun getIdeasByLabelName(labelName: String): Flowable<RealmResults<Idea>> =
        realm.where(Idea::class.java)
                .equalTo("labels.name", labelName)
                .findAllSorted("title")
                .asFlowable()
```

Figura 3.42 – Método que busca todas as ideias a partir do nome de uma etiqueta

Existem duas exceções no projeto que não utilizam o *Realm* diretamente. Uma delas, já mencionado, é o áudio gravado. Ele é gravado diretamente em disco e somente seu caminho é gravado no *Realm*. A segunda exceção é a tablatura. Todos os dados do

cabeçalho da tablatura poderiam ser salvos tranquilamente com o *Realm*, porém, não é possível salvar as notas da tablatura de maneira prática.

O jeito mais simples e eficaz de lidar com as notas da tablatura foi utilizando a biblioteca *GSon* [19], capaz de converter uma classe Java em uma representação JSON [20] e vice-versa. Além de ser capaz de converter classes que carregam tipos primitivos, ela dá suporte a diversas estruturas de dados padrões do Java. Logo, as notas presentes na tablatura são convertidas perfeitamente pela biblioteca.

O *GSon* foi utilizado no projeto para converter a classe *Tablature* inteira para JSON. Assim, na classe *Idea*, existe uma instância da classe *Tablature* que é anotada com a anotação *@Ignore* para que o *Realm* não tente gravá-la. Ao invés disso, o JSON é salvo em disco e o caminho do arquivo é salvo pelo *Realm* em forma de *string*. O próprio JSON poderia ser convertido em *string* e armazenado no *Realm*, porém sempre que uma ideia fosse obtida, uma *string* enorme representando a tablatura também seria obtida, convertida para JSON e depois convertida para uma instância da classe *Tablature*. E isso ocorreria para todas as ideias sempre que o usuário entrasse no app, por exemplo. Na solução implementada, apenas o caminho do arquivo é carregado. Somente quando o usuário necessita ver a tablatura (na página de visualização de ideia e na de edição) é que ela é carregada e convertida. Na Figura 3.43, é possível ver as funções que gravam e carregam a tablatura.

```
fun loadTablature(file: File = File(tablaturePath)) {
    if (file.exists() && file.isFile) {
        tablaturePath = file.path
        val gson: Gson = Gson()
        val json: String = file.readText()
        Log.d(LOG_TAG, "Load tablature")
        Log.d(LOG_TAG, "json = " + json)
        tablature = gson.fromJson(json, Tablature::class.java)
        tablatureIsEmpty = tablature?.notes?.isEmpty != false
    }
}

fun saveTablature(outputFile: File = File(tablaturePath)) {
    if (outputFile.exists()) {
        outputFile.delete()
    }
    outputFile.createNewFile()
    tablaturePath = outputFile.path
    Log.d(LOG_TAG, "Save tablature")
    val gson: Gson = Gson()
    val tablatureAsJson = gson.toJson(tablature)
    Log.d(LOG_TAG, tablatureAsJson.toString())
    outputFile.writeText(tablatureAsJson.toString())
}
```

Figura 3.43 – Métodos que salvam e carregam a tablatura

3.2.8 – Interface e Experiência de Usuário

Em qualquer sistema, existem regras e padrões que definem sua identidade visual e no Android não seria diferente. O Material Design [21] é "Um sistema unificado que combina teoria, recursos e ferramentas para construir experiências digitais" [21] e ele é a base para toda a interface do sistema Android a partir da versão 5.0. Ao desenvolver um app para Android, é recomendado seguir suas *guidelines*. No projeto, o Material Design foi utilizado extensivamente, desde a paleta de cores até as animações. Uma comparação entre os exemplos do Material Design e a implementação no projeto pode ser observado pela Figura 3.44 e Figura 3.45

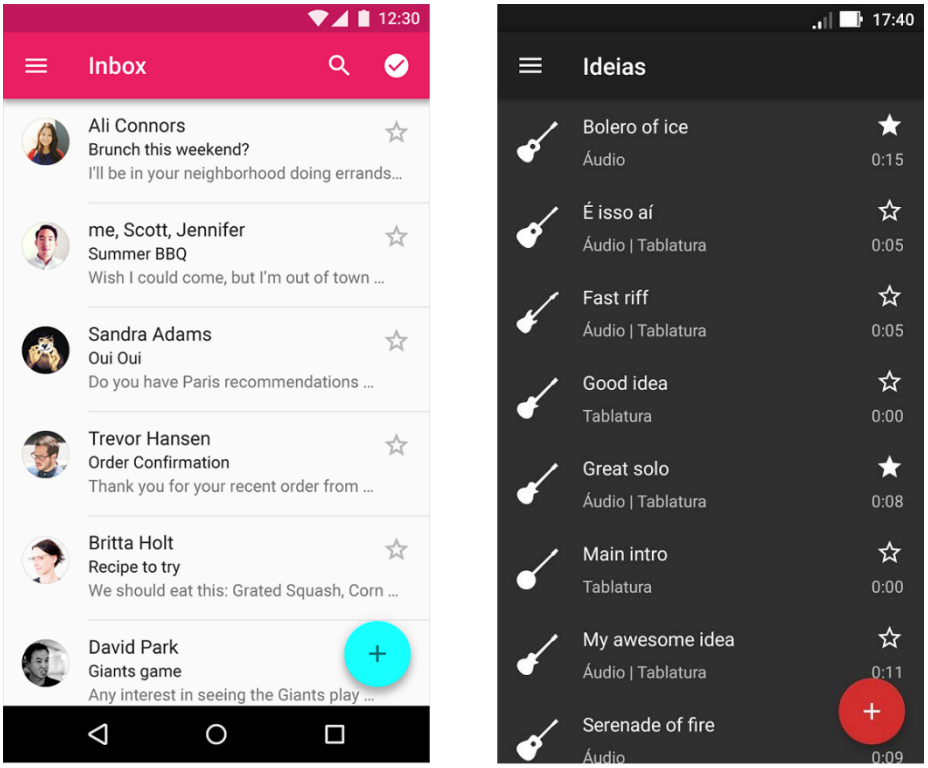


Figura 3.44 – Comparação entre um exemplo de lista definida pelo Material Design (A) e uma lista do projeto (B)
Fonte: Adaptado de [22]

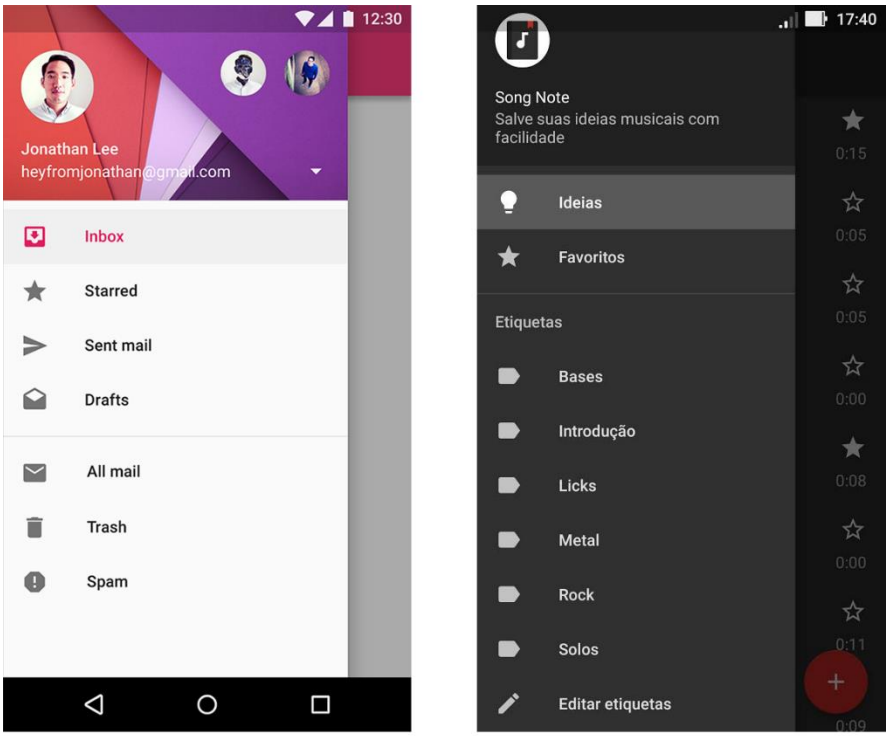


Figura 3.45 – Comparação entre um exemplo de menu lateral definido pelo Material Design (A) e o utilizado no projeto (B)
Fonte: Adaptado de [23]

Quando se trata de design, é muito importante manter a consistência da interface. No projeto, o botão de ação flutuante (FAB – *Floating Action Button*) [24] é bastante utilizado, principalmente para suas ações principais (ver Figura 3.46).

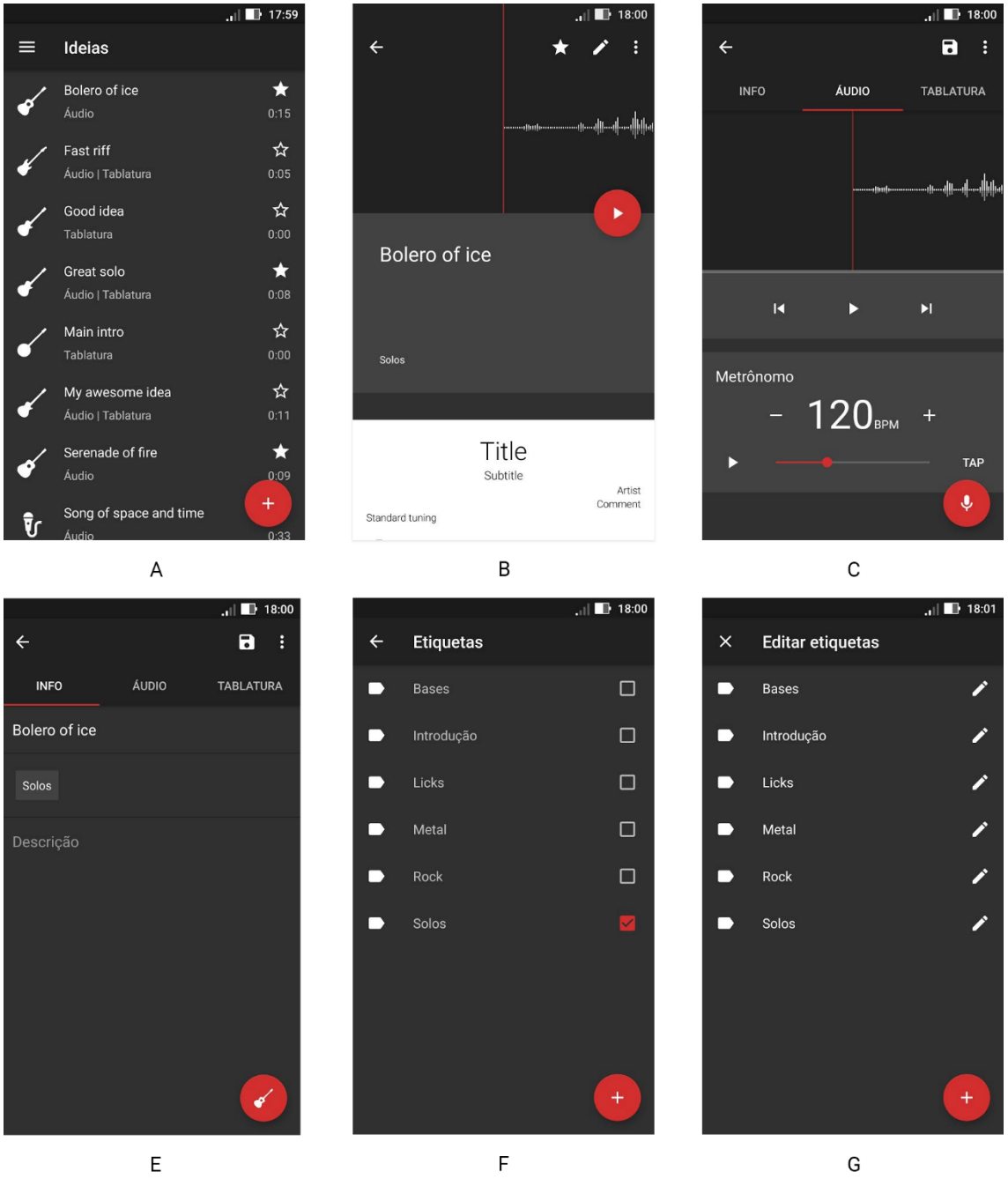


Figura 3.46 – Todos os FABs utilizados no projeto

Também é importante manter a consistência entre os textos do aplicativo. Neste caso, a mesma nomenclatura foi utilizada para ações semelhantes. Por exemplo, a palavra “descartar” foi utilizada quando um usuário está editando uma ideia e resolve apagar a tablatura ou áudio atual, ou quando ele sai do editor sem salvar; neste caso, o aplicativo pergunta se o usuário pretende descartar a ideia (ver Figura 3.47). A palavra “apagar” foi utilizada para realizar ações de apagar em dados já salvos; por exemplo, o usuário pode apagar uma ideia ou uma etiqueta (ver Figura 3.48).

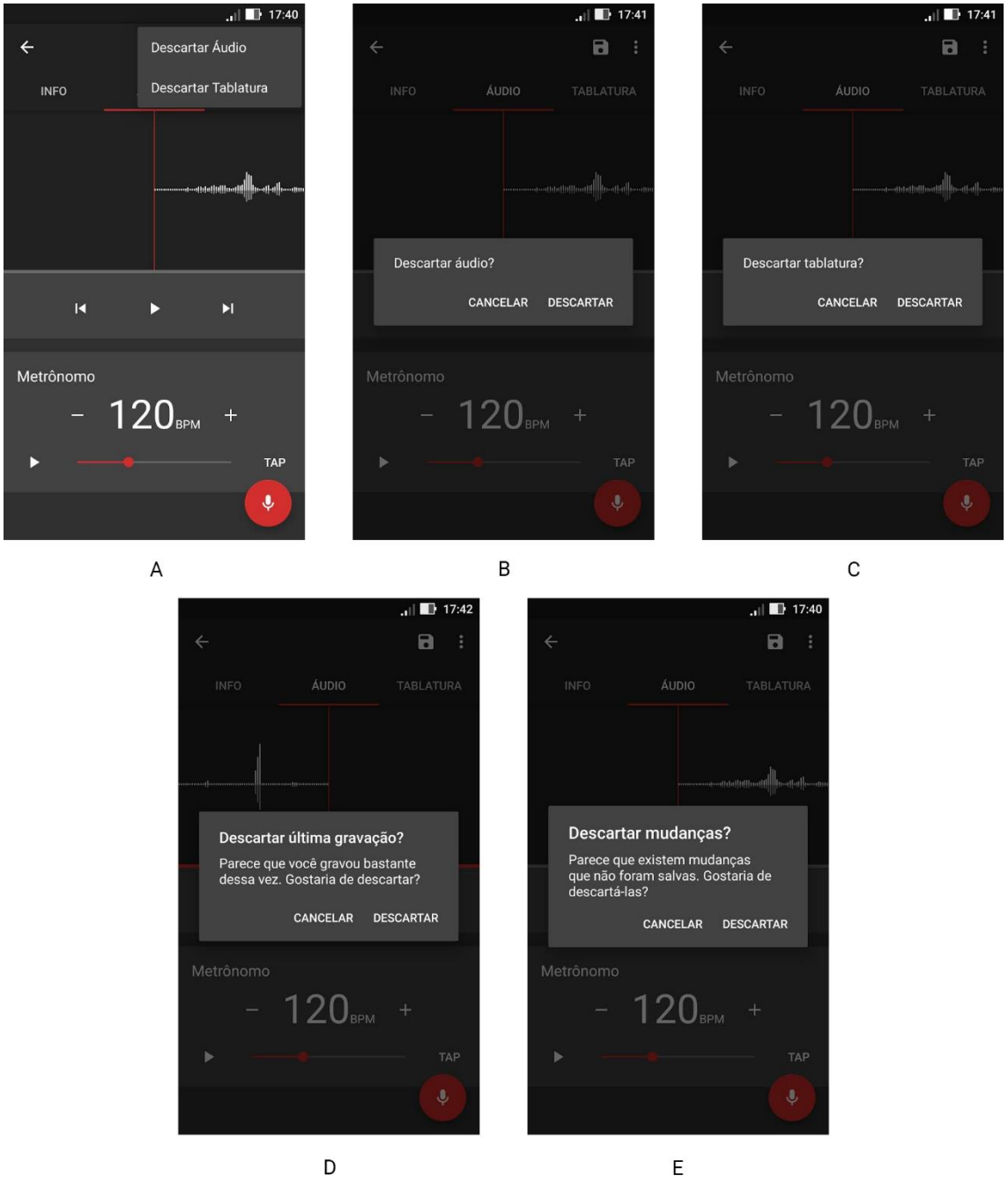


Figura 3.47 – Palavra “descartar” utilizada no projeto

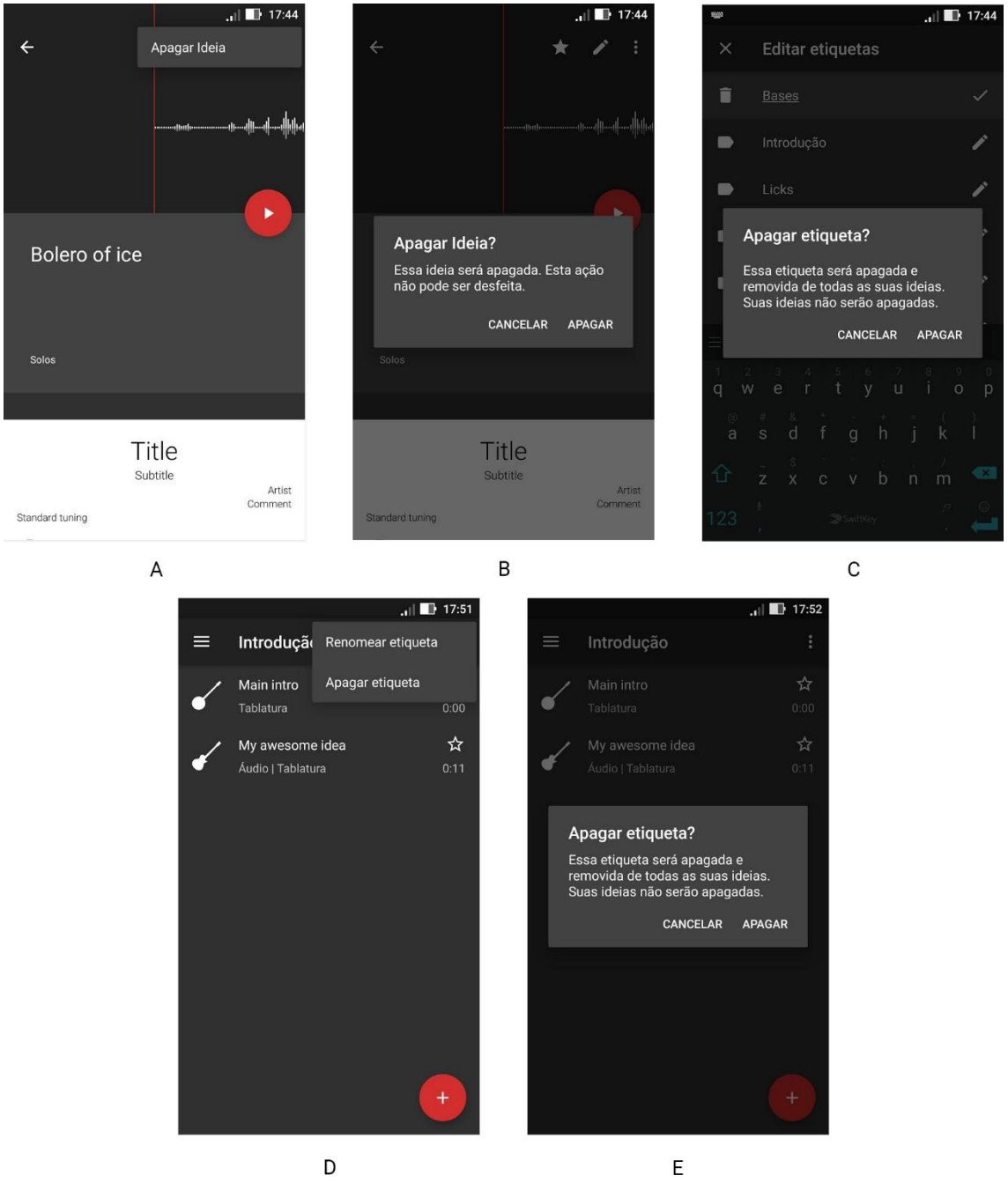


Figura 3.48 – Palavra “apagar” utilizada no projeto

As telas que utilizam os mesmos componentes também são consistentes. Observe na Figura 3.49 uma comparação entre a visualização e edição de ideia. Os componentes utilizados são os mesmos, com leves modificações para aprimorar a experiência do usuário de acordo com o contexto. O mesmo ocorre na edição e na seleção de etiquetas, onde os elementos são em forma de lista e os elementos visuais de cada item da lista depende do contexto.

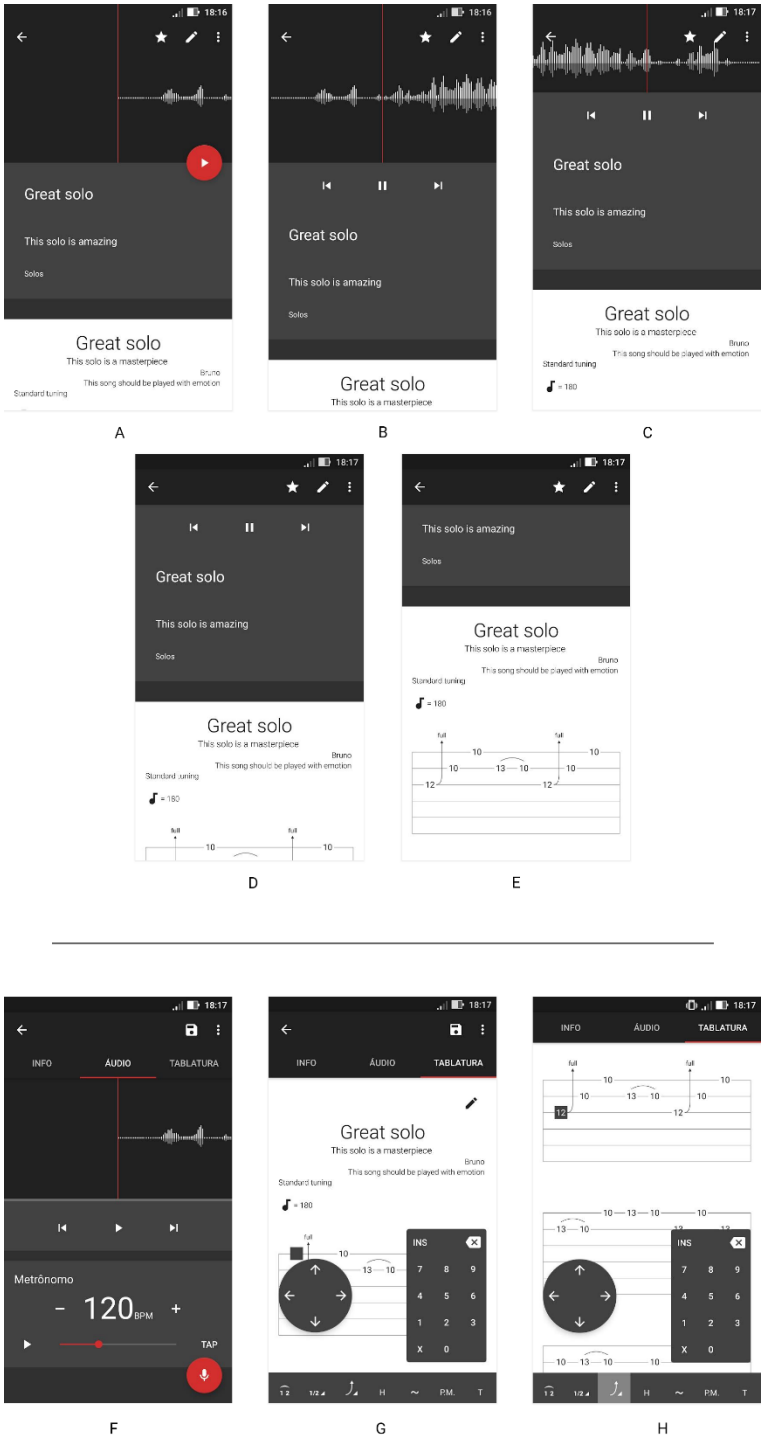


Figura 3.49 – Comparação entre visualização e edição de ideia. A) Página inicial da visualização de ideia. B) Usuário clica no botão de reproduzir áudio. O botão de desloca e se expande para revelar o player. C) Usuário desliza um pouco a tela para baixo. D) Usuário desliza um pouco mais a tela para baixo. Visualização do áudio é escondida com uma animação na opacidade. E) Usuário começa a visualizar a tablatura. F) Página inicial da tela de edição. Áudio está sempre visível. G) Página de edição de tablatura. H) Usuário desliza a tela para baixo. A barra de navegação é deslizada e a tablatura possui um cursor para edição.

Um detalhe muito importante no processo de design é dar um *feedback* para o usuário de acordo com suas ações, pois isto mostra que suas ações realmente ocorreram. Quando o usuário faz uma ação e não nota nenhuma mudança, na sua concepção, a ação não foi feita ou o aplicativo não funcionou da maneira esperada. No projeto, sempre que uma ação importante é realizada, um *snackbar* [25] com o resultado da ação é mostrado para o usuário. É importante ressaltar que erros também podem ocorrer, como por exemplo, não conseguir gravar um áudio por que o usuário não deu permissão para o aplicativo. Nesses casos, é importante também relatar que a ação não foi realizada. Para isso, um *snackbar* informa o erro ocorrido para o usuário.

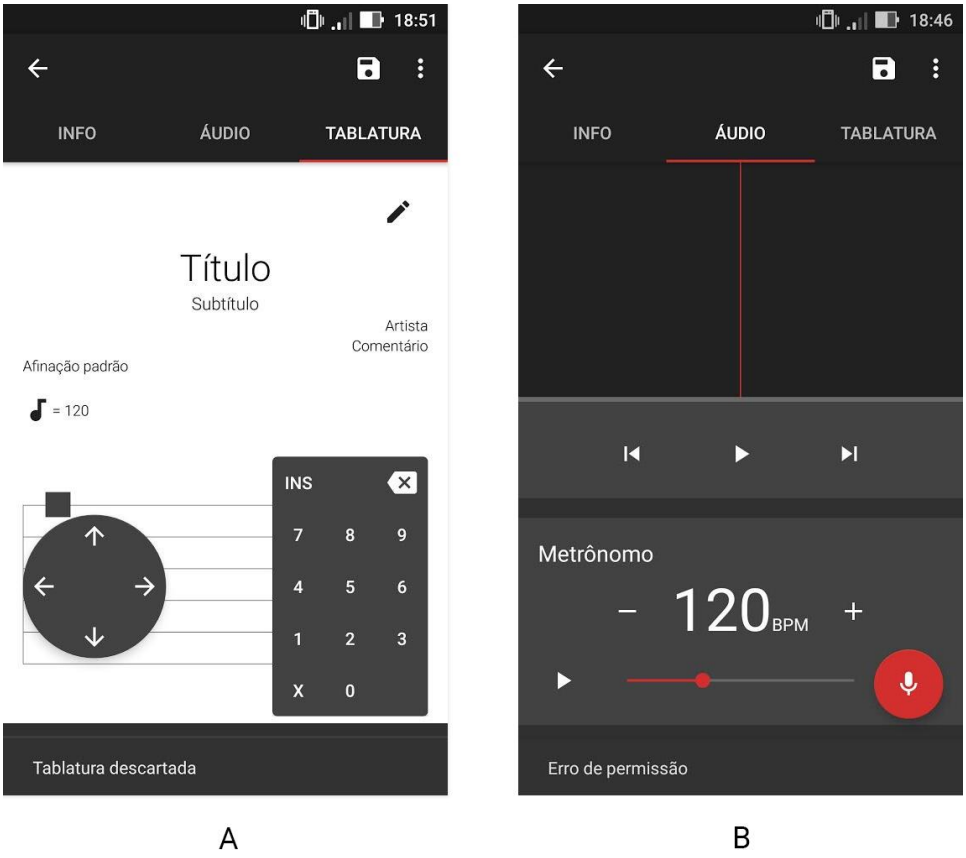


Figura 3.50 – Exemplos de s*nackbar* mostrados para o usuário. A) *Snackbar* de quando o usuário descarta a tablatura. B) *Snackbar* de erro de permissão

Para realizar ações que possam apagar algum conteúdo de forma permanente, como apagar uma ideia, o usuário sempre é notificado para confirmar a ação (ver Figura 3.51). Desta forma, a chance cometer um erro e fazer uma ação indesejada é menor. É sempre importante ter em mente que o usuário pode apertar um botão sem querer, ou até clicar sem saber o seu significado (como um botão que só possui ícone, por exemplo).

Logo, alertar o usuário do que sua ação está prestes a fazer remedia esse comportamento.

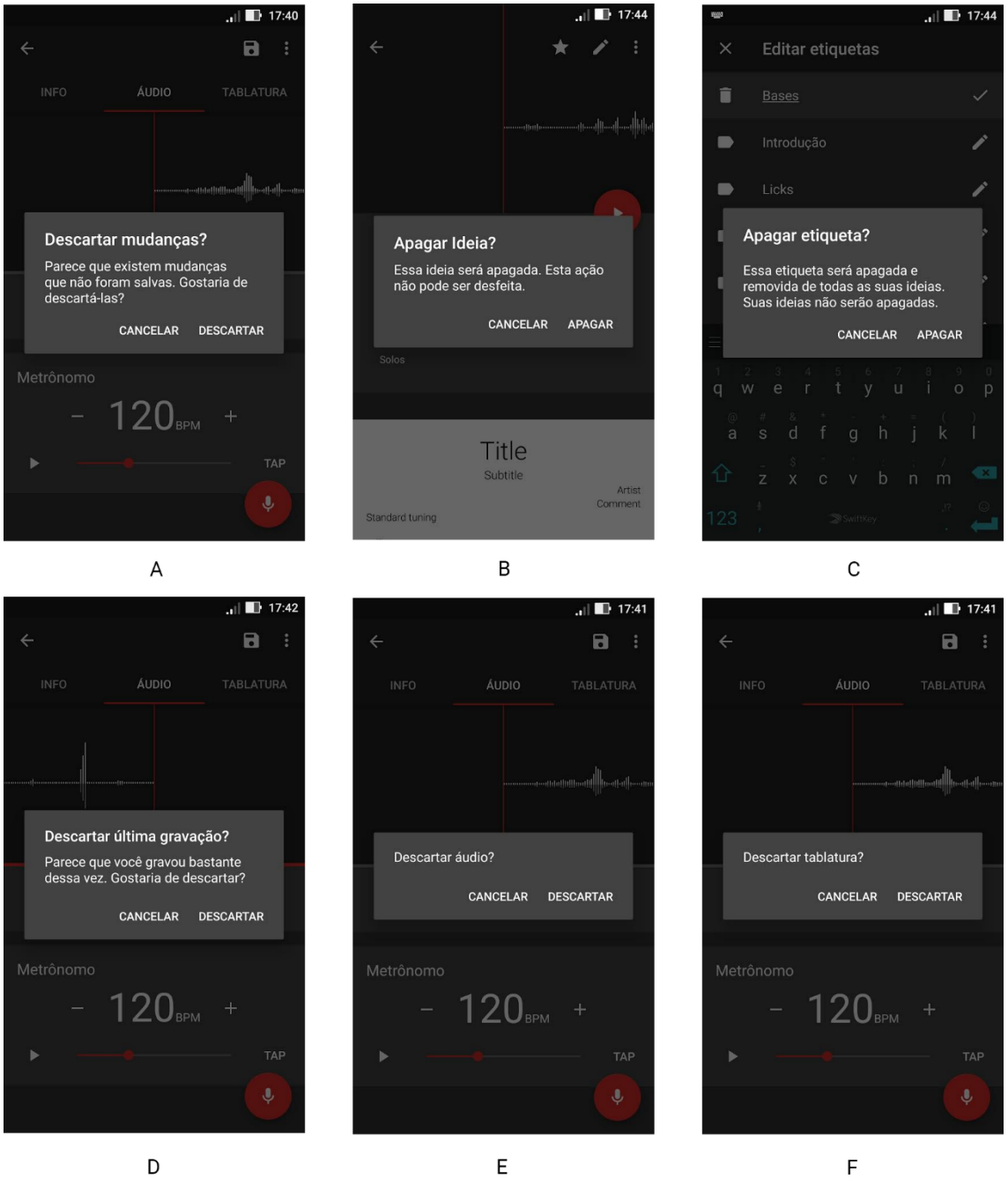


Figura 3.51 – Exemplos de ações perigosas que pedem a confirmação do usuário

É importante também deixar claro para o usuário que uma página está vazia. Uma página vazia pode dar a sensação de que o aplicativo não carregou direito, ou que existe alguma demora e ele precisa esperar o conteúdo carregar. Para evitar essa confusão, foi utilizado o padrão *empty state* [26]. Ele geralmente contém uma imagem ilustrativa de acordo com o context e um texto curto que explica que a página está vazia.

No projeto, este padrão foi utilizado também para ensinar o usuário a utilizar o aplicativo. Quando não há nenhuma ideia salva, imagens e textos mostram para o usuário o que ele deve fazer para criar uma nova ideia. O mesmo ocorre para as etiquetas (ver Figura 3.52).

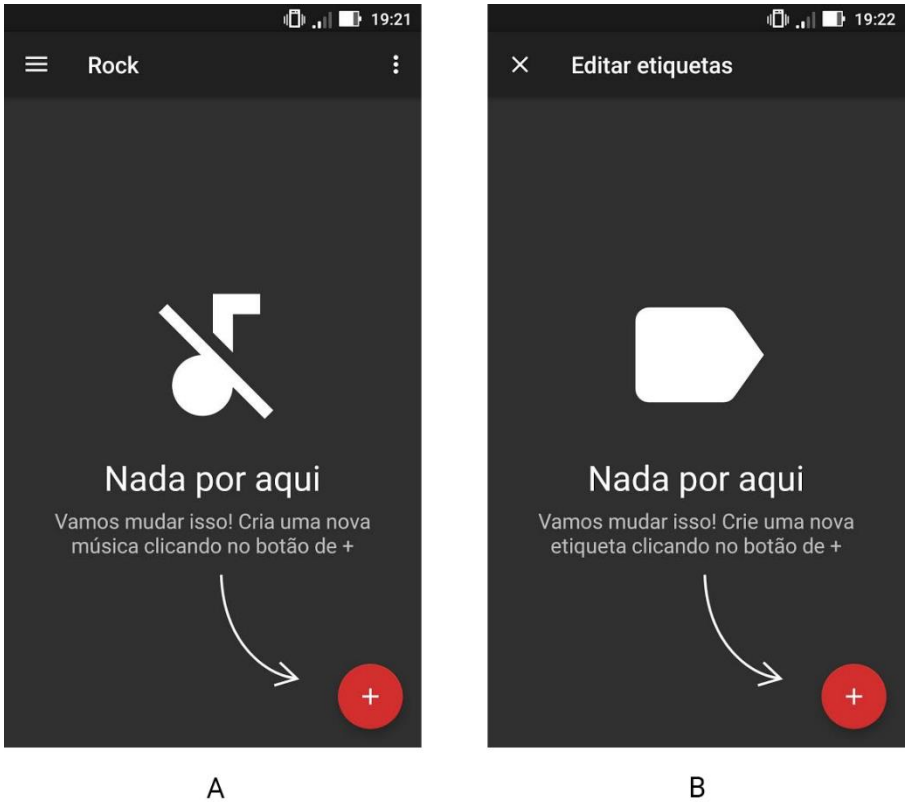


Figura 3.52 – Exemplos de e*mpty state* usados no projeto

Um estudo realizado por Josh Clark [27] e estendido em uma publicação da Smashing Magazine [28] revela que a maioria utiliza os dedos polegares para interagir com o *smatphone*. A partir disso, foram feitos experimentos para entender a dificuldade de alcançar determinadas partes da tela. Em resumo, existe uma região em que a interação com o usuário é natural, ou seja, ela é facilmente alcançada; existe uma segunda região onde o usuário alcança, mas precisa esticar os dedos de forma nada confortável; e existe uma terceira região, onde a interação é muito difícil. Esta comparação pode ser vista na Figura 3.53.

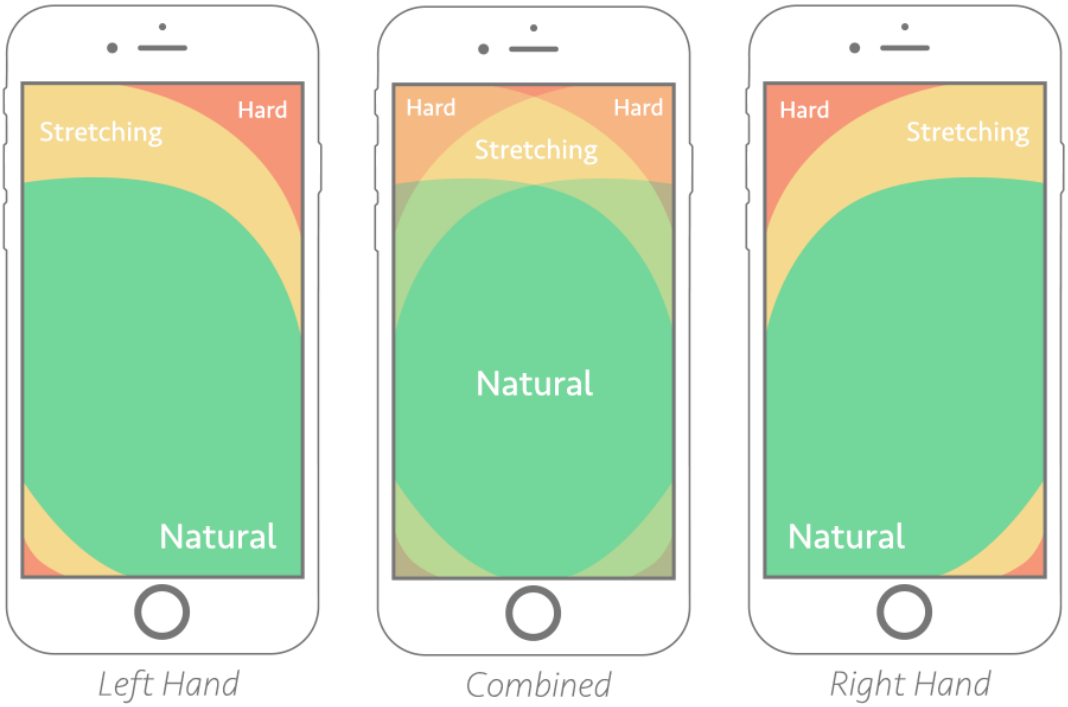


Figura 3.53 – Comparação entre facilidade de interação na tela de um smartphone. A representação mais à esquerda mostra o uso do aparelho utilizando a mão direita. Mais à direita, mostra o uso com a mão esquerda. No meio, mostra o uso com duas mãos
Fonte: Artigo da Smashing Magazine [28]

Um dos pontos do artigo é que a localização do conteúdo é muito importante. Quanto mais inferior for a localização, mais fácil será a interação com o usuário. Não há problema existir interação na parte superior, porém é recomendado que sejam posicionadas funcionalidades de uso menos frequente. Estas informações foram muito úteis para o desenvolvimento do aplicativo. Note que em todas as telas, as funcionalidades principais são facilmente alcançadas pelo usuário. Este conceito é bem explorado na edição de tablatura, onde toda a interação é localizada na parte inferior da tela. Na Figura 3.54, são mostradas as regiões de facilidade de toque na tela de edição de tablatura.

Figura 3.54 – Regiões de facilidade de toque na tela de edição de tablatura
Fonte: Adaptado da Smashing Magazine [28]

3.2.9 – Análises de Resultado

O aplicativo Song Note foi publicado na Google Play em 25 de janeiro de 2018 para usuários que possuam no mínimo Android 5.0 e pode ser baixado através do endereço https://play.google.com/store/apps/details?id=com.brunocalou.songnote de forma gratuita. Como sua publicação foi feita de forma recente, não houve tempo hábil para receber muitos *feedbacks*. Porém, o sentimento do público alvo com relação ao app foi bem positivo. Algumas pessoas disseram estar procurando por um aplicativo assim há meses. As maiores críticas não foram sobre as funcionalidades do aplicativo atualmente, e sim as que ele ainda não possui. A crítica mais marcante foi a de não reproduzir o som a partir da tablatura. Uma outra crítica foi não poder adicionar o tempo das notas. Apesar de ser uma funcionalidade importante, é dispensável para muitos usuários.

Com cerca de 30 dias de lançamento, o app foi baixado por 274 usuários, e desinstalado por 85. A maior parte dos downloads foi feita no Brasil, onde o app teve mais divulgação. Porém, outros países também se encontram na lista de downloads, como Estados Unidos, Índia e Rússia.

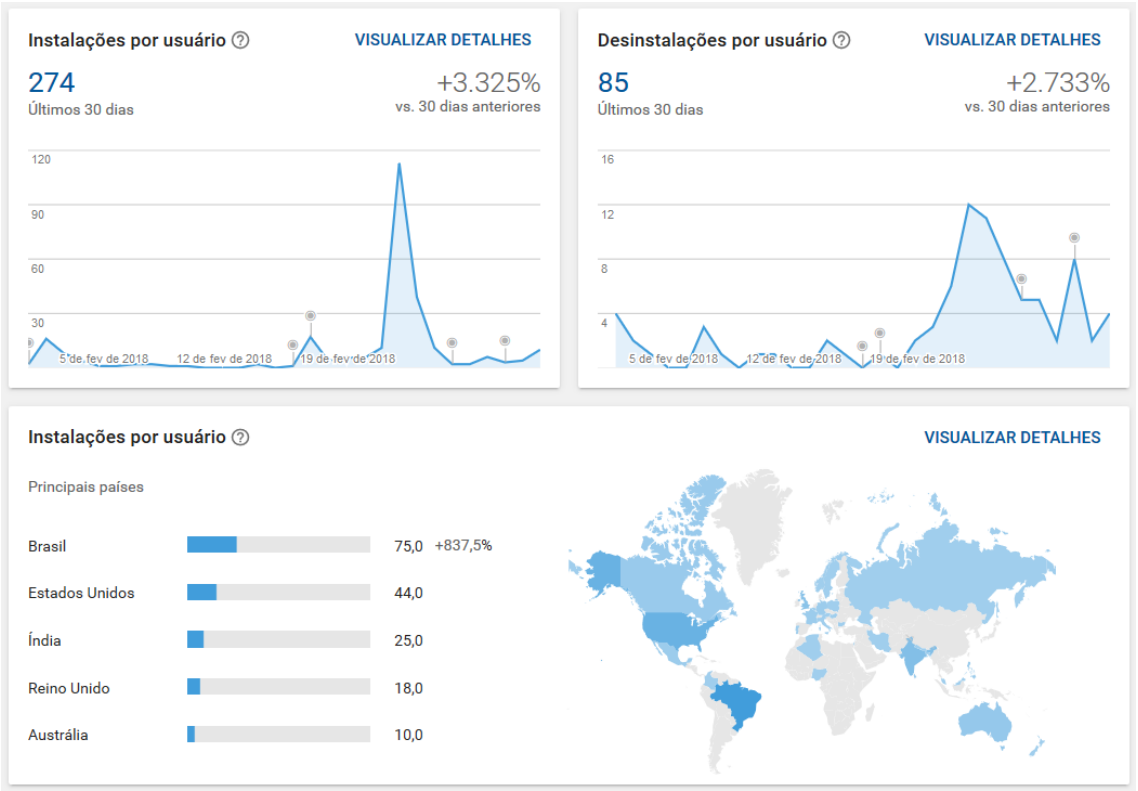


Figura 3.55 – Estatísticas de uso do aplicativo
Fonte: Aplicativo no Google Play Console [29]

Apesar de ter um número relativamente alto de instalações, o número de desinstalações também é alto. Diversos experimentos [30] [31] [32] [33] [34] mostram que a taxa de retenção de aplicativos é bastante baixa e varia de acordo com a categoria do aplicativo, público alvo, memória disponível no aparelho e engajamento do usuário. Como o aplicativo possui um nicho muito limitado (músicos que queiram salvar suas ideias musicais), possui uma forte concorrência e ainda não possui todas as funcionalidades de outros aplicativos, é esperado que a taxa de retenção não seja muito alta, levando a um grande número de desinstalações.

Capítulo 4

Conclusão e Trabalhos Futuros

Dentro do escopo definido, o aplicativo atende bem os requisitos, sendo capaz de guardar e organizar ideias musicais, associando um áudio a uma tablatura. A edição de tablatura é feita de forma rápida pelo usuário, pois todas as suas funcionalidades são posicionadas de forma coerente no canto inferior da tela. Além de poder gravar o áudio em sequência, ou seja, sendo capaz de parar e continuar uma gravação, o aplicativo permite descartar a última gravação, se o usuário desejar. O sistema de etiquetas permite categorizar as ideias e filtrá-las rapidamente, o que contribui para a organização das ideias. O usuário recebe um *feedback* de todas as suas ações, o que dá a certeza de que aquela ação foi feita. Além disso, o aplicativo possui uma boa interface baseada no Material Design.

Por estar na Google Play a pouco tempo, o aplicativo ainda não possui muitos downloads, porém seu potencial para o sucesso é considerável. O *feedback* dos usuários foi bastante positivo e, apesar de não possuir todas as funcionalidades de outros aplicativos, o app realmente resolve o problema de escrever tablaturas de instrumentos de cordas trasteados e fazer a associação com uma gravação, além de manter a organização de ideias musicais.

Como trabalhos futuros, serão desenvolvidas as funcionalidades mais pedidas pelos usuários, além de outras que não foram implementadas, mas que são importantes, como poder ordenar as músicas por data de criação, por exemplo. Algumas das funcionalidades previstas para o futuro do aplicativo estão listadas abaixo:

- Traduzir o aplicativo para outros idiomas
- Compartilhar o áudio
- Compartilhar a tablatura em forma de texto e imagem
- Compartilhar tablatura em forma de PDF para impressão
- Permitir a documentação do tempo e da intensidade de cada nota
- Otimizar design do aplicativo para tablets
- Adicionar mais efeitos na tablatura

- Dicionário de acordes
- Editor de cifras
- Permitir configuração do aplicativo (valores padrão na tablatura, no metrônomo, etc.)
- Poder ordenar ideias por nome, data de criação, de modificação, etc.
- Incluir uma barra de pesquisa na tela inicial para buscar ideias a partir de um texto
- Melhorar a implementação do metrônomo, dando mais opções de configuração
- Adicionar contagem regressiva antes de começar uma gravação
- Selecionar, copiar, colar e remover partes da tablatura e do áudio
- Desfazer e refazer ações no editor de ideia
- Segmentar a tablatura em partes diferentes
- Reprodução de tablatura
- Filtros de áudio (distorção, *delay*, *noise gate*, *reverb*, etc.)
- Permitir a importação de áudio em diversos formatos (e.g. compartilhar um áudio com o aplicativo)
- Escrever tablatura a partir do som gravado
- Fazer *backup* das ideias na nuvem

Bibliografia

[1] “Guitar Pro - Sheet music editor software for guitar, bass, keyboards, drums and more...”, https://www.guitar-pro.com, 2017, (Acesso em 11 Outubro 2017)

[2] “Google Play”, https://play.google.com/store, 2017, (Acesso em 11 Outubro 2017)

[3] “Guitar Pro – Apps para Android no Google Play”, https://play.google.com/store/apps/details?id=com.arobasmusic.guitarpro, 2017, (Acesso em 11 Outubro 2017)

[4] “Guitar Tabs X – Apps para Android no Google Play”, https://play.google.com/store/apps/details?id=com.finestandroid.guitartabs, 2017, (Acesso em 11 Outubro 2017)

[5] “Tab Maker (tablature editor) – Apps para Android no Google Play”, https://play.google.com/store/apps/details?id=wolf.games.mobile.tabmaker, 2017, (Acesso em 11 Outubro 2017)

[6] “TuxGuitar – Apps para Android no Google Play”, https://play.google.com/store/apps/details?id=org.herac.tuxguitar.android.activity.admob, 2017, (Acesso em 11 Outubro 2017)

[7] “IgaraFu(Tablature Editor) free – Apps para Android no Google Play”, https://play.google.com/store/apps/details?id=org.artica.muu.igarafufree, 2017, (Acesso em 11 Outubro 2017)

[8] “Guitar Partner Lite – Apps para Android no Google Play”, https://play.google.com/store/apps/details?id=com.datoh.app.android.tabdroid.lite, 2017, (Acesso em 11 Outubro 2017)

[9] “Guitar Notepad - Tab Editor – Apps para Android no Google Play”, https://play.google.com/store/apps/details?id=com.codefad.guitarnotepad, 2017, (Acesso em 11 Outubro 2017)

[10] “Tabify guitar composition – Apps para Android no Google Play”, https://play.google.com/store/apps/details?id=com.tabify.writer, 2017, (Acesso em 11 Outubro 2017)

[11] “Conheça o Android Studio | Android Studio”, https://developer.android.com/studio/intro/, 2017, (Acesso em 11 Outubro 2017)

[12] “IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains”, https://www.jetbrains.com/idea/, 2017, (Acesso em 11 Outubro 2017)

[13] “Model–view–viewmodel – Wikipedia”,

https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel, 2018, (Acesso em 12 Fevereiro 2018)

[14] “ReactiveX/RxJava: RxJava – Reactive Extensions for the JVM – a library for composing asynchronous and event-based programs using observable sequences for the Java VM.”, https://github.com/ReactiveX/RxJava, 2018, (Acesso em 12 Fevereiro 2018)

[15] “AudioRecord | Android Developers”, https://developer.android.com/reference/android/media/AudioRecord.html, 2018, (Acesso em 14 Fevereiro 2018)

[16] “AudioTrack | Android Developers”, https://developer.android.com/reference/android/media/AudioTrack.html, 2018, (Acesso em 14 Fevereiro 2018)

[17] “SoundPool | Android Developers”, https://developer.android.com/reference/android/media/SoundPool.html, 2018, (Acesso em 15 Fevereiro 2018)

[18] “Realm: Create reactive mobile apps in a fraction of the time”, https://realm.io/, 2018, (Acesso em 15 Fevereiro 2018)

[19] “google/gson: A Java serialization/deserialization library to convert Java Objects into JSON and back”, https://github.com/google/gson, 2018, (Acesso em 15 Fevereiro 2018)

[20] “JSON”, https://www.json.org/json-pt.html, 2018, (Acesso em 15 Fevereiro 2018)

[21] “Material Design”, https://material.io/, 2018, (Acesso em 16 Fevereiro 2018)

[22] “Lists - Components - Material Design”, https://material.io/guidelines/components/lists.html, 2018, (Acesso em 16 Fevereiro 2018)

[23] “Navigation drawer - Patterns - Material Design”, https://material.io/guidelines/patterns/navigation-drawer.html, 2018, (Acesso em 16 Fevereiro 2018)

[24] “Buttons: Floating Action Button - Components - Material Design”, https://material.io/guidelines/components/buttons-floating-action-button.html, 2018, (Acesso em 16 Fevereiro 2018)

[25] “Snackbars & toasts - Components - Material Design”, https://material.io/guidelines/components/snackbars-toasts.html, 2018, (Acesso em 16 Fevereiro 2018)

[26] “Empty states - Patterns - Material Design”,

https://material.io/guidelines/patterns/empty-states.html, 2018, (Acesso em 16 Fevereiro 2018)

[27] “How We Hold Our Gadgets · An A List Apart Article”, http://alistapart.com/article/how-we-hold-our-gadgets, 2015, (Acesso em 16 Fevereiro 2018)

[28] “The Thumb Zone: Designing For Mobile Users — Smashing Magazine”, https://www.smashingmagazine.com/2016/09/the-thumb-zone-designing-for-mobile-users/, 2016, (Acesso em 16 Fevereiro 2018)

[29] “Google Play Console”, https://play.google.com/apps/publish/, 2018, (Acesso em 18 Fevereiro 2018)

[30] “New data shows losing 80% of mobile users is normal, and why the best apps do better at andrewchen”, http://andrewchen.co/new-data-shows-why-losing-80-of-your-mobile-users-is-normal-and-that-the-best-apps-do-much-better/, 2016, (Acesso em 13 Março 2018)

[31] “Enter the Matrix: App Retention and Engagement | Flurry Blog”, http://flurrymobile.tumblr.com/post/144245637325/appmatrix, 2016, (Acesso em 13 Março 2018)

[32] “Now Available – Android Uninstall Benchmarks | Apsalar”, https://apsalar.com/2016/06/now-available-android-uninstall-benchmarks/, 2016, (Acesso em 13 Março 2018)

[33] “23% of Users Abandon an App After One Use | Localytics”, http://info.localytics.com/blog/23-of-users-abandon-an-app-after-one-use, 2016, (Acesso em 13 Março 2018)

[34] “Mobile App Retention: 75% Users Uninstall An App Within 90 Days [REPORT] - Dazeinfo”, https://dazeinfo.com/2016/05/19/mobile-app-retention-churn-rate-smartphone-users/, 2016, (Acesso em 13 Março 2018)

Apêndice A

Implementação de Média Móvel

```
package com.brunocalou.songnote.utils;

import java.util.ArrayList;

public class MovingAverage {
    private ArrayList<Double> numbers;
    private ArrayList<Double> weights;
    private int currentIndex = 0;
    private int size = 0;

    public MovingAverage(int size) {
        this.size = size;
        numbers = new ArrayList<>();
        numbers.ensureCapacity(size);

        weights = new ArrayList<>();
        weights.ensureCapacity(size);

        // Sum of the arithmetic progression from 1 to size
        double sum = size * (1 + size) / 2;

        for (int i = 0; i < size; i += 1) {
            numbers.add(0d);
            weights.add((size - i) / sum);
        }
}

    public void add(double number) {
```

```
        numbers.set(currentIndex, number);
        currentIndex += 1;

        if (currentIndex == numbers.size()) {
            currentIndex = 0;
        }
    }

    public double getAverage() {
        double sum = 0;

        for (int i = 0; i < numbers.size(); i += 1) {
            int weightIndex = i - currentIndex;
            if (weightIndex < 0) weightIndex += numbers.size();

            sum += numbers.get(i) * weights.get(weightIndex);
        }

        return sum / numbers.size();
    }

    public void clear() {
        for (int i = 0; i < numbers.size(); i += 1) {
            numbers.set(i, 0d);
        }
    }

    public int getSize() {
        return size;
    }
}
```