Técnicas de Construção de Programas

Trabalho Prático – Fase 2

Semestre 2020/2

Professor Marcelo Soares Pimenta

00302072 - Victória Duarte

Descrição do Trabalho

Objetivo

O programa tem como objetivo principal gerar uma música a partir de um texto fornecido pelo usuário. Além de fornecer o texto, o usuário também seleciona o instrumento inicial a ser utilizado e o BPM inicial da música.

Organização geral

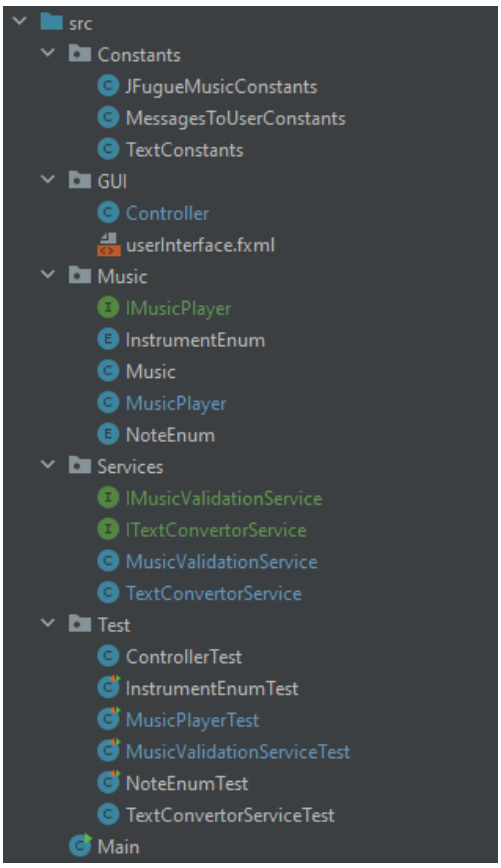


Imagem com a organização dos pacotes e classes do trabalho

O código é organizado a partir da pasta principal (*src*). Essa pasta contém a classe *Main* e 5 pacotes: *GUI*, *Music*, *Services* e *Test*.

Classe *Main*

Classe principal que inicia a execução do projeto pela interface com o usuário.

Pacote *GUI*

O pacote *GUI* contém a interface com o usuário e a classe que controla as interações possíveis entre usuário e interface.

Pacote *Music*

Pacote que contém as classes relacionadas a música. Nesse pacote, temos a classe *Music*, responsável por conter a *string* que será transformada em música pela biblioteca utilizada (*JFugue*).

Além disso, temos a classe *MusicPlayer*, que possibilita a música ser tocada e salva, e os *Enums NoteEnum* e *InstrumentEnum*, responsáveis por guardar, respectivamente, as notas e instrumentos possíveis de serem utilizados durante o código.

Pacote *Services*

No pacote de *Services*, temos classes que oferecem serviços para outras classes. Os serviços oferecidos são a validação dos *inputs* fornecidos pelo usuário e a conversão do texto de *input* em um texto que pode ser tocado como música pela biblioteca *JFugue*.

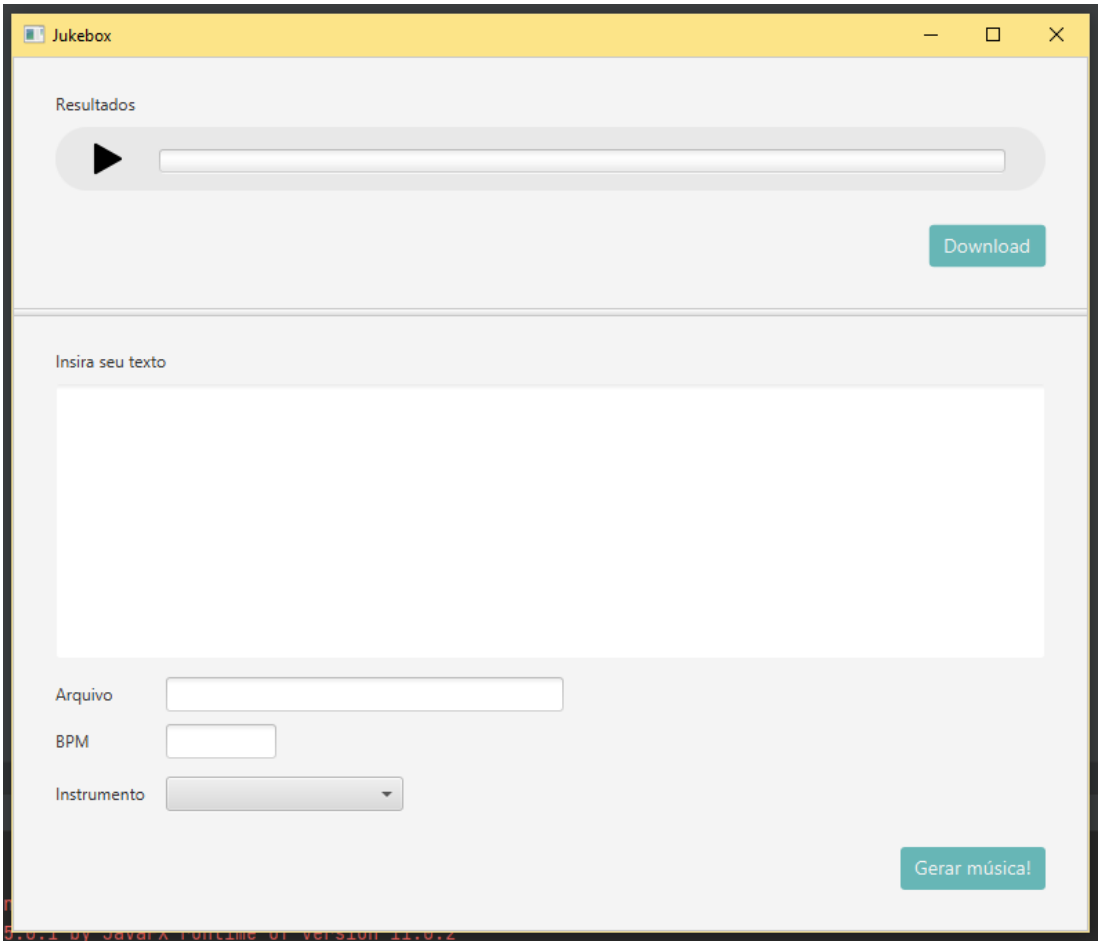
Pacote Test

O pacote de testes, por fim, armazena todos os testes unitários realizados nas classes do programa.

Interface com o Usuário

A interface com usuário se propõe a ser o mais simples e intuitiva possível. Ela é dividida em duas sessões: área dos inputs e área de resultado da música. Na primeira área, o usuário pode colocar os inputs, sendo todos obrigatórios menos o nome do arquivo (caso o usuário não ponha, é utilizado um nome default). Na segunda sessão, a música pode ser ouvida e baixada.

A barra de progresso para acompanhar quanto da música foi tocada ainda não foi implementada e as mensagens sobre o que deve ser colocado em cada input podem ser mais claras.



Interface com o usuário

Avaliação da Modularidade do Programa

Critérios

Decomposability e Composability

De forma geral, o código atinge satisfatoriamente os requisitos de decomponibilidade e compossibilidade.

Podemos observar isso, por exemplo, na definição da classe *Music* e da classe *MusicPlayer*.

```
public class Music {
    private static final int MAX_VOLUME = 127;
    public final int initialVolume;
    public String musicString;

    public Music(String raw_text, int initialBpm, int initialInstrument) {
        TextConvertorService textConvertor = new TextConvertorService();
        initialVolume = (int) (0.2 * MAX_VOLUME);
        musicString = textConvertor.convert(raw_text, initialVolume, initialBpm, initialInstrument);
    }
}
```

Implementação da classe Música. Um de seus atributos é a String musicString, responsável por armazenar a música convertida em um formato que pode ser transformado em música pelo JFugue

```
public interface IMusicPlayer {
    void playMusic(String musicString);
    boolean saveMusic(String musicString, String filename);
}
```

Interface da classe MusicPlayer. Observa-se que para ambos os métodos da interface, pede-se uma string que corresponde ao texto previamente convertido em um formato reconhecível pelo JFugue (musicString).

Enquanto a *Music* fica responsável exclusivamente pela criação da música, o *MusicPlayer* se responsabiliza unicamente por tocar e salvar uma música pré-existente. Ou seja, cada classe tem uma função específica e independente, mas podem ser facilmente conectadas através de um único parâmetro, *musicString*, presente nos métodos do *MusicPlayer*.

```
@FXML
private void OnPlayButtonClicked() { player.playMusic(music.musicString); }
```

Chamada do método playMusic utilizando musicString do objeto music da classe Music.

Understandability

O código, em sua maioria, cumpre bem com o requisito de compreensibilidade. Para que esse objetivo pudesse ser atingido, houve uma preocupação muito grande em tornar os nomes dos métodos e das variáveis o mais claro e preciso possível, além do uso de constantes ao invés de *“string mágicas”* ou *“número mágicos”*.

```java
public boolean validateString(String text, String instrument) {
    if(text.trim().equals("")) {
        errorMessage = MessagesToUserConstants.EMPTY_STRING_MESSAGE;
        return false;
    } else if(text.length() > 240){
        errorMessage = MessagesToUserConstants.TOO_LONG_STRING_MESSAGE;
        return false;
    } else if(instrument == null) {
        errorMessage = MessagesToUserConstants.NULL_INSTRUMENT;
        return false;
    }
    return true;
}
```

O nome do método deixa claro que será feito uma validação de strings dadas como parâmetro, em que uma é um texto e outra é um instrumento. O uso de constantes identificam o tipo de erro gerado com o input inadequado oferecido pelo usuário. No entanto, ainda há números e strings “mágicas” que podem ser substituídas.

```java
private void setInstruments(List<String> list) {
    list.add( index: 0,  element: JFugueMusicConstants.INSTRUMENT + currentInstrument);

    if(list.contains(TextConstants.RANDOM_INSTRUMENT)){
        list.set(list.indexOf(TextConstants.RANDOM_INSTRUMENT), JFugueMusicConstants.INSTRUMENT + InstrumentEnum.getRandomInstrument());
    }

}
```

Uso de constantes ao invés dos caracteres definidos na especificação do trabalho ou das constantes definidas pela biblioteca JFugue facilitam a compreensão do fluxo do código.

No entanto, ainda há áreas no código que a compreensibilidade pode ser melhorada.

```java
public int parseBPM(String bpm_string){
    int bpm;
    try {
        bpm = Integer.parseInt(bpm_string);

        if(bpm < 40 || bpm > 220) {
            errorMessage = MessagesToUserConstants.INVALID_BPM_VALUE;
            bpm = -1;
        }
    } catch (NumberFormatException exception) {
        errorMessage = MessagesToUserConstants.INVALID_BPM_TYPE;
        bpm = -1;
    }
    return bpm;
}
```

No método parseBPM há os chamados "números mágicos" na verificação do BPM. Ao invés dos valores, é possível usar constantes que identifiquem o valor mínimo e máximo que o BPM pode atingir.

Continuity

O problema da continuidade ainda pode ser mais trabalhado. Há trechos em que se tem uma boa continuidade no código, como, por exemplo, com a definição de constantes para os caracteres identificados no input de texto do usuário que devem ser convertidos.

```java
public class TextConstants {
    public static final String BLANK_SPACE = " ";
    public static final String INCREASE_OCTAVE = "T+";
    public static final String DECREASE_OCTAVE = "T-";
    public static final String INCREASE_BPM = "BPM+";
    public static final String DECREASE_BPM = "BPM-";
    public static final String INCREASE_VOLUME = "+";
    public static final String RESET_VOLUME = "-";
    public static final String RANDOM_INSTRUMENT = "\\n";
    public static final String RANDOM_NOTE_OP1 = ".";
    public static final String RANDOM_NOTE_OP2 = "?";
    public static final String REPEAT_NOTE_OP1 = "I";
    public static final String REPEAT_NOTE_OP2 = "O";
    public static final String REPEAT_NOTE_OP3 = "U";
}
```

Classe que define todas as constantes da especificação do trabalho

Porém, há locais em que a continuidade ainda precisa ser melhorada. Com o *NoteEnum*, por exemplo, há a definição das notas e seus respectivos valores. Porém, no método *convertNotes*, há novamente a definição das letras correspondentes ao *Enum*. Algo parecido ocorre com a definição dos instrumentos: é feita a definição dos instrumentos na *choiceBox* (onde o usuário pode escolher um instrumento inicial) e, novamente, no *InstrumentEnum*.

```
public enum NoteEnum {
    C( i: 0), D( i: 2), E( i: 4), F( i: 5), G( i: 7), A( i: 9), B( i: 11);
```

Definição das possíveis notas no NoteEnum.

```
private List<String> convertNotes(List<String> list) {
    // substituir as notas com as oitavas certas
    HashMap< String, NoteEnum> notes = new HashMap<>();
    notes.put("A", NoteEnum.A);
    notes.put("B", NoteEnum.B);
    notes.put("C", NoteEnum.C);
    notes.put("D", NoteEnum.D);
    notes.put("E", NoteEnum.E);
    notes.put("F", NoteEnum.F);
    notes.put("G", NoteEnum.G);
```

Nova definição das possíveis notas, usando o NoteEnum.

```
public enum InstrumentEnum {
    Piano( i: 0), Guitar( i: 24), Violin( i: 40), Trumpet( i: 56), Bass( i: 32);
```

Definição dos possíveis instrumentos no InstrumentEnum.

```
private void initialize() {
    String[] instruments = {"Violin", "Piano", "Guitar", "Bass", "Trumpet"};
    choiceBox.setItems(FXCollections.observableArrayList(instruments));
}
```

Mais uma definição dos possíveis instrumentos, na choiceBox.

Protection

A proteção dos métodos foi feita de maneira satisfatória. Locais que pudessem gerar erros faziam as devidas verificações antes de prosseguir com outras atividades, garantindo, assim, que os erros não se propagam código adentro.

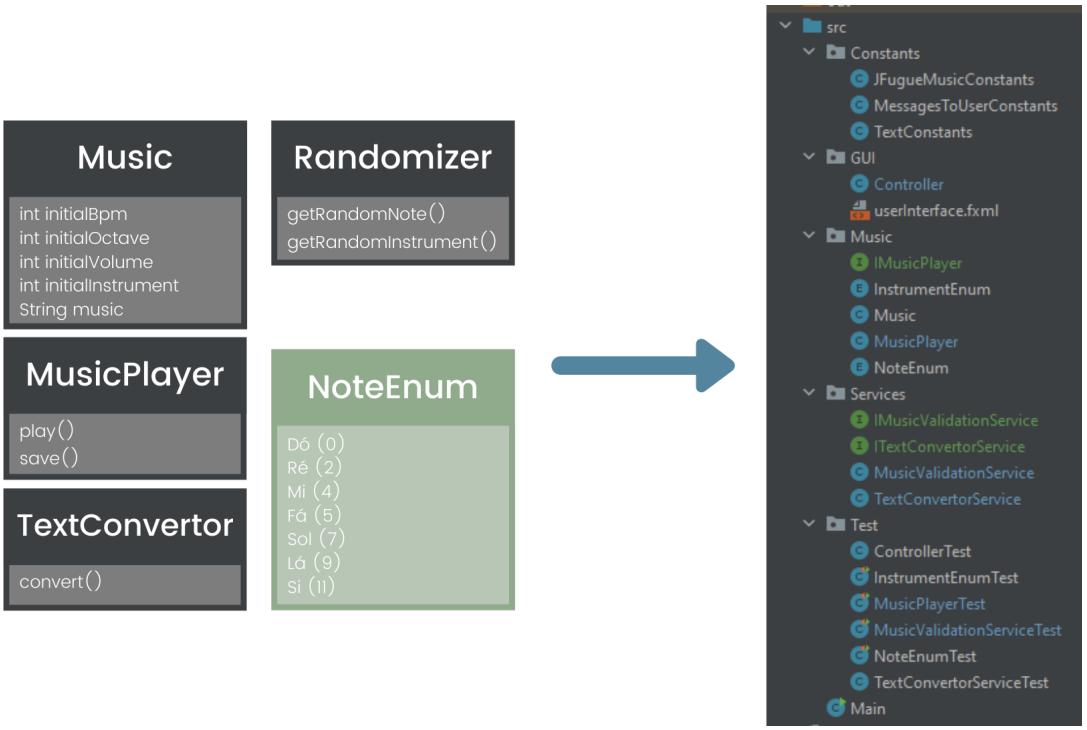
```java
@FXML
private void OnGenerateMusicButtonClicked() {
    MusicValidationService musicValidationService = new MusicValidationService();
    String textInput = getTextInput();
    String selectedInstrument = onSelectInstrument();
    int initialBPM = musicValidationService.parseBPM(getBPMInput());

    if(musicValidationService.validateString(textInput, selectedInstrument) && initialBPM != -1) {
        music = new Music(textInput, initialBPM, InstrumentEnum.valueOf(selectedInstrument).getValue());

        if(!music.musicString.isBlank()) {
            createSuccessAlert(MessagesToUserConstants.SUCCESSFUL_MUSIC_CREATION);
        } else {
            createErrorAlert(MessagesToUserConstants.FAILURE_MUSIC_CREATION);
        }
    } else {
        createErrorAlert(MusicValidationService.errorMessage);
    }
}
```

Duas verificações são feitas quando o usuário clica no botão "Gerar música!": primeiramente, verifica-se se o usuário forneceu os inputs necessários e se os inputs fornecidos são válidos. Caso o primeiro teste não gere problemas, é feita a conversão da música e verifica-se se a string gerada não é vazia, já que o usuário pode fornecer um texto com nenhum caractere que seja convertido em música.

Regras

Direct Mapping

Apesar de algumas modificações, a estrutura do código foi mantida bem parecida com a estrutura do domínio de problema proposto pelo relatório da fase 1.



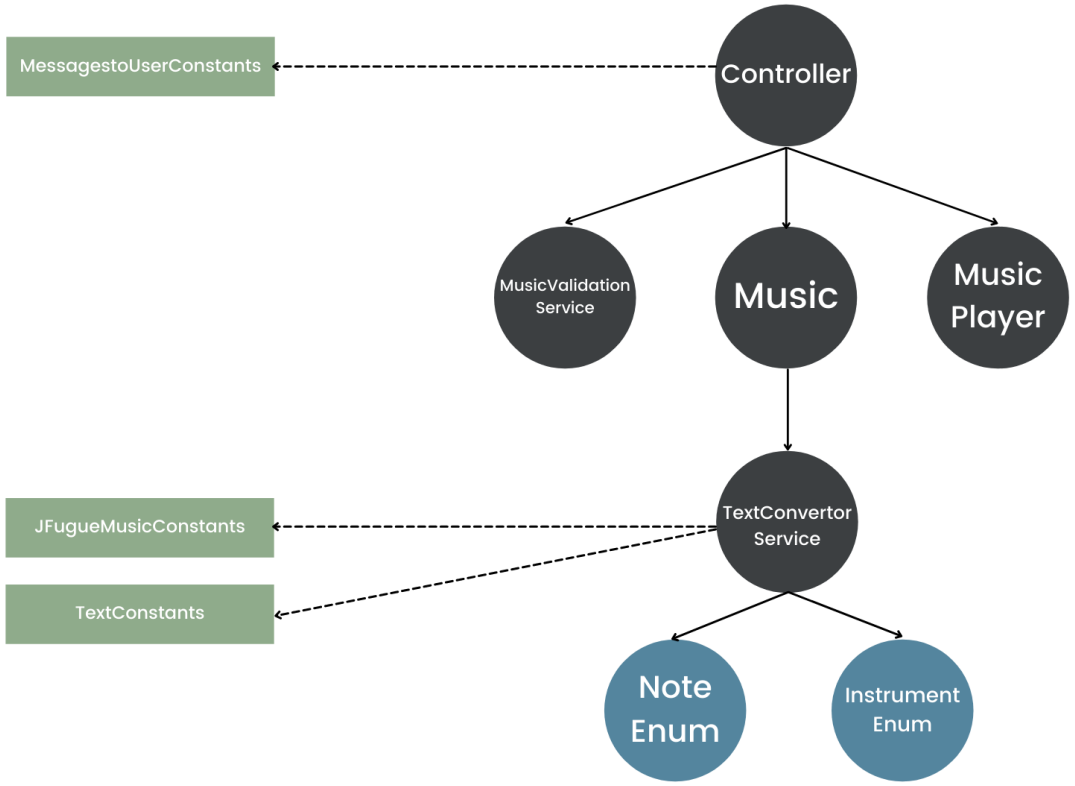
Esquema representando o mapeamento feito entre a definição do domínio do problema e o código gerado. É possível notar que a classe Randomizer não foi implementada, pois seus métodos foram transferidos para os enum correspondentes. Além disso, classes como MusicValidationService não apareceram inicialmente no planejamento, mas foram adicionadas quando se viu a necessidade durante a produção.

```java
public class MusicPlayer implements IMusicPlayer {
    private final static Player player = new Player();

    public void playMusic(String musicString){...}

    public boolean saveMusic(String musicString, String filename) {...}
}
```

Um exemplo do código mostrando o mapeamento direto entre a proposta do MusicPlayer e a implementação da classe MusicPlayer.

Few Interfaces

Os acessos entre as classes podem ser estruturados de acordo com o esquema apresentado abaixo



O esquema demonstra que cada classe só pode ser acessada por uma única classe, revelando uma baixa comunicação entre classes. Além disso, o esquema revela uma estrutura de árvore nos acessos.

Small Interfaces

Na comunicação entre as classes só é passado parâmetros que sejam efetivamente necessários para a classe que os receberá.

```
public Music(String raw_text, int initialBpm, int initialInstrument) {
    TextConvertorService textConvertor = new TextConvertorService();
    initialVolume = (int) (0.2 * MAX_VOLUME);
    musicString = textConvertor.convert(raw_text, initialVolume, initialBpm, initialInstrument);
}
```

Ao chamar o método convert, são passados apenas os valores essenciais para a geração da música

Explicit Interfaces

Toda a comunicação entre classe é explicitada durante o código.

```
public Music(String raw_text, int initialBpm, int initialInstrument) {
    TextConvertorService textConvertor = new TextConvertorService();
    initialVolume = (int) (0.2 * MAX_VOLUME);
    musicString = textConvertor.convert(raw_text, initialVolume, initialBpm, initialInstrument);
}
```

Usando o mesmo trecho anterior, há na primeira linha do construtor de Music uma instanciação de um objeto do tipo TextConvertorService. Só então é feito uso do método convert.

Information Hiding

O código foi construído tentando deixar o maior número de métodos privados possível. Só foi deixado público métodos que precisavam ser acessados. Um exemplo desse encapsulamento é a classe *TextConvertorService*. Em sua interface, consta apenas um método, *convert*, mas ao acessar a classe que implementa essa interface, é possível encontrar uma série de métodos auxiliares que são de uso exclusivo da classe.

```
public interface ITextConvertorService {
    String convert(String raw_text, int initialVolume, int initialBpm, int initialInstrument);
}
```

Interface do TextConvertorService, com apenas um método declarado.

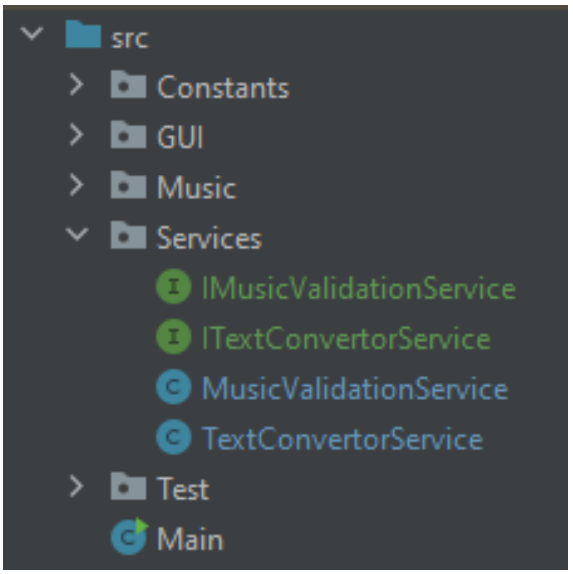
```java
public class TextConvertorService implements ITextConvertorService{
    private int currentOctave;
    private int currentInstrument;
    private int currentVolume;
    private int currentBpm;

    public TextConvertorService() { currentOctave = 1; }

    public String convert(String raw_text, int initialVolume, int initialBpm, int initialInstrument) {...}

    private ArrayList<String> cleanString(String text) {...}

    private List<String> addPauses(List<String> list) {...}

    private List<String> convertNotes(List<String> list) {...}

    private void setBpm(List<String> list) {...}

    private void setInstruments(List<String> list) {...}

    private List<String> setVolume(List<String> list, int initialVolume) {...}
}
```

Implementação da classe TextConvertorService conta com vários métodos privados que auxiliam na construção do método convert.

Princípios

The Linguistic Modular Units Principle

A forma como o código foi estruturado divide o programa em unidades semânticas, tentando manter uma estrutura coerente e relativamente independente.



Exemplo da estruturação do código em classes e pacotes seguindo "The Linguistic Modular Units Principle"

The Self-Documentation Principle

Manter o código auto-documentável foi uma das maiores preocupações durante a estruturação do programa. Houve uma grande preocupação na escolha de nomes que fossem precisos e revelassem suas intenções.

Além disso, houve o cuidado em, sempre que possível, usar constantes que deixassem mais claro o que um método estava fazendo.

```java
public class Music {
    private static final int MAX_VOLUME = 127;
    public final int initialVolume;
    public String musicString;

    public Music(String raw_text, int initialBpm, int initialInstrument) {
        TextConvertorService textConvertor = new TextConvertorService();
        initialVolume = (int) (0.2 * MAX_VOLUME);
        musicString = textConvertor.convert(raw_text, initialVolume, initialBpm, initialInstrument);
    }
}
```

Exemplo de código mostrando uma preocupação em usar variáveis que revelem claramente sua função e constantes que facilitem o entendimento geral.

The Uniform Access Principle

Todos os métodos são acessados da mesma forma (objeto.método()), assim como seus atributos (objeto.atributo).

```java
@FXML
private void OnGenerateMusicButtonClicked() {
    MusicValidationService musicValidationService = new MusicValidationService();
    String textInput = getTextInput();
    String selectedInstrument = onSelectInstrument();
    int initialBPM = musicValidationService.parseBPM(getBPMInput());

    if(musicValidationService.validateString(textInput, selectedInstrument) && initialBPM != -1) {
        music = new Music(textInput, initialBPM, InstrumentEnum.valueOf(selectedInstrument).getValue());

        if(!music.musicString.isBlank()) {
            createSuccessAlert(MessagesToUserConstants.SUCCESSFUL_MUSIC_CREATION);
        } else {
            createErrorAlert(MessagesToUserConstants.FAILURE_MUSIC_CREATION);
        }
    } else {
        createErrorAlert(MusicValidationService.errorMessage);
    }
}
```

Exemplo do código demonstrando o uso de métodos e atributos de maneira uniforme.

The Open-Closed Principle

O código está funcionando e realizando as tarefas necessárias, mas ainda há possibilidade de expansão. A expansão pode incluir adição de métodos, modificações de métodos existentes ou até criação de novas classes.

```
private List<String> setVolume(List<String> list, int initialVolume) {
    List<String> result = new ArrayList<>();
    for (String item : list) {
        if(item.equals(TextConstants.INCREASE_VOLUME)) {
            if(currentVolume * 2 < 127) {
                currentVolume *= 2;
            } else {
                currentVolume = 127;
            }
            result.add(":CON(7," + currentVolume + ")");
        } else if (item.equals(TextConstants.RESET_VOLUME)) {
            result.add(":CON(7," + initialVolume + ")");
        } else {
            result.add(item);
        }
    }

    return result;
}
```

No exemplo, caso fosse necessário adicionar um caractere que diminuísse o volume em 10 unidades, isso poderia ser facilmente feito alterando poucos métodos (o método apresentado no exemplo e a classe de TextConstants).

The Single Choice Principle

Esse princípio não foi satisfatoriamente seguido. Isso porque o conteúdo contido nas classes Enum precisaram ser repetidos em mais de um lugar.

```
public enum NoteEnum {
    C( i: 0), D( i: 2), E( i: 4), F( i: 5), G( i: 7), A( i: 9), B( i: 11);
```

Declaração das notas possíveis no Enum.

```java
private List<String> convertNotes(List<String> list) {
    // substituir as notas com as oitavas certas
    HashMap< String, NoteEnum> notes = new HashMap<>();
    notes.put("A", NoteEnum.A);
    notes.put("B", NoteEnum.B);
    notes.put("C", NoteEnum.C);
    notes.put("D", NoteEnum.D);
    notes.put("E", NoteEnum.E);
    notes.put("F", NoteEnum.F);
    notes.put("G", NoteEnum.G);
```

As notas possíveis são mais uma vez declaradas.

```java
public enum InstrumentEnum {
    Piano( i: 0), Guitar( i: 24), Violin( i: 40), Trumpet( i: 56), Bass( i: 32);
```

Os instrumentos possíveis são declarados.

```java
private void initialize() {
    String[] instruments = {"Violin", "Piano", "Guitar", "Bass", "Trumpet"};
    choiceBox.setItems(FXCollections.observableArrayList(instruments));
}
```

Os instrumentos possíveis são novamente declarados.

Discussão e Proposta de Solução

Muito progresso foi feito com o programa, que já é capaz de fazer sua função principal, gerar música. Porém, ainda há muitas possibilidades de melhora na organização do código, tornando-o mais limpo, claro e estruturado. Para que isso seja feito, vale pensar em novas formas de estruturar a lógica, continuar com o uso das constantes e bons nomes para métodos e variáveis e manter atenção aos critérios, regras e princípios de modularidade.

Além disso, também falta implementar a barra de progresso, onde será possível acompanhar quanto da música já foi tocado.