Descrição do programa

Objetivo do programa:

Converter um texto para uma música, usando um mapeamento de conjunto de caracteres para comandos musicais.

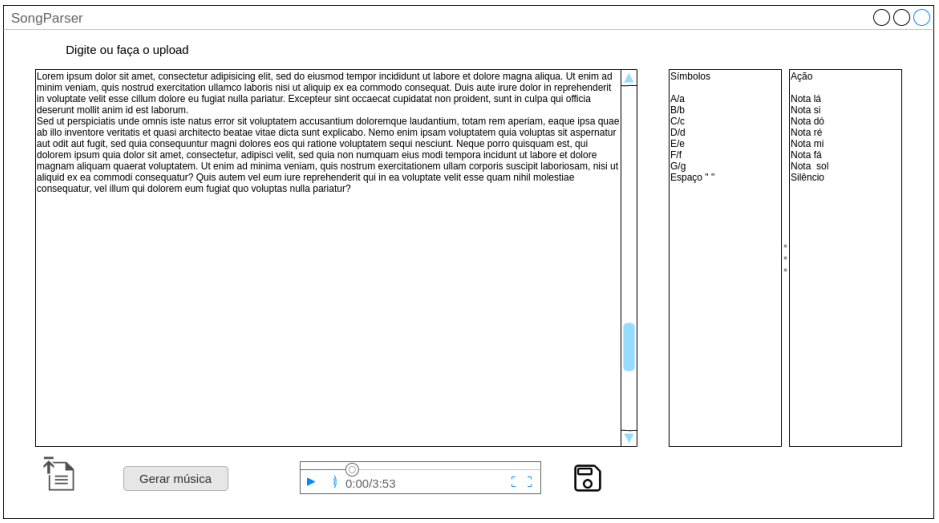
Descrição das classes principais:

```
// Ao criada deve inicializar a biblioteca de interface gráfica e criar e exibir a tela do programa, também deve responder a inputs do usuário;
public class MainWindow {
  String InputText;
  JButton GenerateMusicBtn;
  JButton SaveMusicBtn;
  JButton PauseMusicBtn;
  JTextArea inputText;
  WidgetPlayer widgetPlayer;
  JPanel instructionTable;
  // Callback que ocorre quando o usuário clica no botão para gerar a música, deve pegar o texto atual e passa-lo para a musicFactory;
  void OnGenerateMusicBtn();
  // Callback  que ocorre quando o usuário clica no botão para salvar a música, deve chamar o método save da classe Music;
  void OnSaveMusicBtn();
  // Callback que ocorre quando o usuário clica no botão para pausar/start a música, deve tocar ou pausar a música dependendo do estado anterior, chamando o método correspondente da classe Music;
  void OnPauseBtn();
  // Seta o tempo da música na barra de progressão;
  void setWidgetPlayerTime();
  //  Lê arquivo contendo texto da música
  void readFromTxt();

}

// É a música em si, seria a interface do programa com a biblioteca utilizada:
public class Music {
  List<Pattern> patterns;
  Player player;
  //  cria a música com os padrões dados;
```

```
  Music(List<Pattern> patterns);
  // Começa a tocar a música;
  void Start();
  //  Pausa a música;
  void Pause();
  // Salva a música em um arquivo de áudio que corresponde ao
caminho dado
  public void saveMusicOutput(String pathTo);
 // Salva a música em um arquivo MIDI que corresponde ao caminho
dado
 public void saveMusicMIDI(String pathTo);
  // Diz qual tempo estamos na música;
  Timestamp getTime();
}


// Classe que cuida do processo da geração da música através de um
texto;
public class MusicFactory {
  // Gera a Musica correspondente através do texto passado;
  public static Music createMusic(String textMusic);
  // Gera lista de padrões Musicais correspondente ao texto passado
  private static List<Pattern> convertStringToPatterns(String
textMusic);
}
```

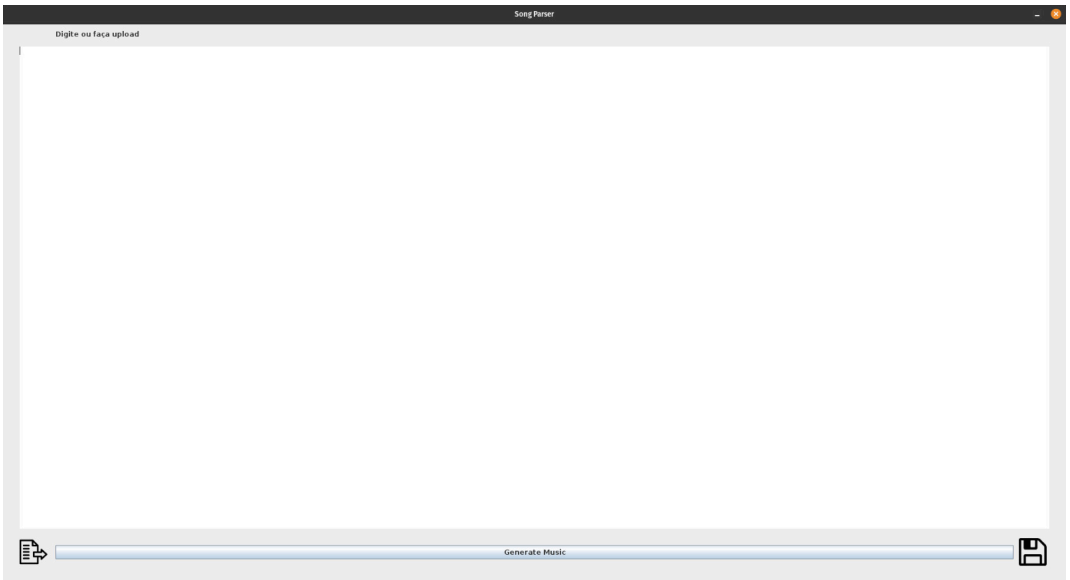
Interface:

O programa terá somente uma tela, sendo este o mockup:



No centro temos uma caixa de texto, lado direito a tabela das instruções, abaixo temos um botão de carregar arquivo, um botão para gerar a música, um player e um botão para salvar a música.

Até o momento, o desenvolvimento da tela está assim:



Caixa de texto ao centro, botão de enviar arquivo no canto esquerdo inferior, botão de gerar a música no meio e botão de salvar no canto direito inferior. Ainda falta o player e as tabelas com o mapeamento. E claro corrigir o espaçamento e adicionar mais alguns retoques.

Avaliação da modularidade do programa

5 critérios:

- **Decomposability**:

  O programa é dividido em 3 módulos, sendo eles:
  - MainWindow (Gerencia a GUI);
    ```
    public class MainWindow{}
    ```
  - MusicFactory (Converte texto para música);
    ```
    public class MusicFactory {}
    ```
  - Music (Gerencia a música);
    ```
    public class Music {}
    ```

  O que segue o critério da decomposição pois quebra o problema em problemas menores que são solucionados por cada um dos módulos de forma independente;

- **Composability**:

  Os módulos MusicFactory e Music, seguem o critério da composição pois poderiam ser usados em contextos diferentes, MusicFactory não precisaria estar associado a uma GUI, e Music poderia gerenciar músicas que não são criadas a partir de um texto. O MainWindow é uma classe que não segue o critério de composability.

- **Understandability:**

  Segue o critério pois cada classe tem um funcionamento claro e independente dos outros módulos. MusicFactory cria uma lista de padrões Musicais. Music gerencia padrões musicais. MainWindow gerencia a janela principal.

- **Continuity:**

  Segue o critério da continuidade, pois se fosse alterada os requisitos da GUI, apenas MainWindow mudaria, já se fosse alterado os requisitos de conversão, apenas o MusicFactory mudaria.

- **Protection:**

  Ainda não segue o critério de proteção pois deveria validar o arquivo recebido como input e o caminho dado para salvar músicas (2 maiores fontes possíveis de run-time errors).

5 regras:

- **Direct Mapping**.

A estrutura do problema contém um texto, um conjunto de caracteres para mapeamento e uma música resultante da conversão. Entretanto, não temos um mapeamento direto para módulos texto e mapeamento.

- **Few Interfaces**

A classe MainWindow se comunica com as outras duas, entretanto elas (MusicFactory e Music) não se comunicam entre si, logo, temos 3 classes e somente duas comunicações, seguindo a regra.

```
// Callback que ocorre quando o usuário clica no botão para gerar a música, deve pegar o texto atual e passa-lo para a musicFactory;
void OnGenerateMusicBtn();
// Callback  que ocorre quando o usuário clica no botão para salvar a música, deve chamar o método save da classe Music;
void OnSaveMusicBtn();
```

- **Small interfaces (weak coupling)**

As comunicações que ocorrem entre as classes manipulam somente uma variável por vez:

```
public Music(List<Pattern> patterns)
```

Todas os métodos definidos, passam uma variável ou zero. E muitos métodos não retornam nada, logo segue a regra de trocar o menor número de informações possíveis entre módulos.

- **Explicit Interfaces**

Segue a regra de Explicit Interface, pois o único data-sharing que temos no código, consiste ao chamar a função createMusic do MusicFactory, que cria então a lista de padrões e então passa essa mesma lista de padrões para a classe Music:

```
public static Music createMusic(String textMusic);
```

Entretanto, não viola a regra, pois está clara a conexão entre MusicFactory e Music, seja pelo nome das classes, seja pelo nome do método, bem como, a referência a lista de padrões musicais nunca é controlada pelas 2 classes ao mesmo tempo, primeiro a MusicFactory cria, depois passa a referência a Music, que se torna a única classe a manter referência dessa lista. Logo não há como haver efeitos colaterais, pois sempre somente uma classe possui a referência a lista de padrões.

- **Information Hiding**

Segue a regra de Information Hiding, pois todas as propriedades das classes são privadas e só são controladas por meio de funções, permitindo assim total controle delas pela classe que as contém.

5 princípios:

- **The Linguistic Modular Units principle**

Está sendo implementado em Java, que tem suporte a orientação a objetos, logo tem correspondência aos módulos que haviam sido planejados.

- **The Self-Documentation principle**

O nome das classes indica suas funções e comentários que ajudam em suas compreensões está escrito diretamente no código, como exemplo, o método createMusic, que conta com um comentário descrevendo o objetivo do método, além de que seu nome e do atributo de entrada são auto-descritivos.

```
// Gera a Musica correspondente através do texto passado;
public static Music createMusic(String textMusic);
```

- **The Uniform Access principle**

Como todas as propriedades das classes são privadas, o acesso é feito de maneira uniforme através de funções, o que não revela se o campo sendo acessado foi computado ou já estava armazenado. Mantendo o acesso uniforme para ambos os casos.

- **The Open-Closed principle**

É possível adicionar novos métodos e operações nos módulos, como, por exemplo, para manipular a música de diferentes formas em Music, entretanto, os métodos estão bem definidos para que já sejam utilizados.

- **The Single Choice principle**

Para transformar o texto em música, utilizamos padrões que podem ser alterados; esses padrões são conhecidos e manipulados somente pela classe MusicFactory, as outras classes não interferem nisso, logo, isso segue o princípio.

Proposta de soluções

Para o caso da proteção, a solução seria adicionar métodos privados que validassem o arquivo de input e o caminho passado pelo usuário para salvar a música. A solução seria efetivada adicionando esses 2 métodos a MainWindow:

private boolean validadeFile(File file);

private boolean validatePath(String path);

Para a questão do direct mapping, criamos classes correspondentes ao mapeamento e a o texto.

Para o mapeamento temos:

```
public class Mapping {
        private HashMap<String, String> mapCharsToJFugueCommand;

            public Mapping() {
                initializeMap();
            }

            // Inicializa o map com valores hardcoded
            private void initializeMap();

             // Retorna uma lista com as Strings que mapeaiam pra
        um comando no map
            public Set<String> getChars();

            // Retorna uma lista com os comandos JFugue no map
            public Collection<String> getCommands();

             // Dado uma string, retorna a string do comando JFugue
        correspondente ou String vazia caso não esteja presente
        no map
            public String getCommand(String chars);
}
```

Para o texto não há tanto sentido em criar uma classe, pois já temos a classe String no java, usada para representar o texto.

Para a questão da composibility, temos o problema da MainWindow ser uma classe bem específica, por aglutinar todas as outras classes, logo não há muita forma de contornar a especificidade da classe MainWindow. Uma forma que encontramos de contornar esse problema foi criar mais classes GUI. Permitindo uma maior composibility.

2 Classes que criamos foi a classe MouseAdapterIconLabel e a classe PlayerGUI

```
// Classe criada para tornar a classe JLabel do Swing, em botões com um efeito de hover. A biblioteca Swing não suporta botões feito apenas por imagens.
public class MouseAdapterIconLabel extends MouseAdapter {
    Image originalImage;
    Image hoveredImg;
    Runnable actionCallback;
    JLabel label;
        public MouseAdapterIconLabel(JLabel label, String iconFile, int normalSize, int incrementFactor, Runnable actionCallback){
        this.label = label;
        label.setMinimumSize(new Dimension(normalSize + 5, normalSize + 5));
        Image img =  null;
        try {
            img = ImageIO.read(new File(iconFile));
        } catch (IOException e) {
             System.err.println("Error when opening icon: " +  iconFile +  " error: " + e.getMessage());
            throw new RuntimeException();
        }
          this.originalImage = img.getScaledInstance(normalSize,normalSize, Image.SCALE_SMOOTH);
        label.setIcon(new ImageIcon(originalImage));
          this.hoveredImg = originalImage.getScaledInstance(normalSize + 5, normalSize + 5, Image.SCALE_SMOOTH);
        this.actionCallback = actionCallback;

    }
    @Override
    public void mouseClicked(MouseEvent mouseEvent) {
        actionCallback.run();
    }

    @Override
    public void mouseEntered(MouseEvent mouseEvent) {
        label.setIcon(new ImageIcon(hoveredImg));
    }

    @Override
    public void mouseExited(MouseEvent mouseEvent) {
        label.setIcon(new ImageIcon(originalImage));
    }
}
```

```
public class Player extends  JPanel {
   private Music music;
   private JLabel pauseButton;
   // Indica o tempo atual da música
   private JLabel timestampLabel;
   private JProgressBar progressBar;

   public Player();

   public void setMusic(Music music);

}
```