Universidade Federal do Rio Grande do Sul

João Kenji Suwa - 587808
Felipe Pasinato Rossoni - 587631
Otávio Rosa de Oliveira - 588807

Trabalho Prático

Disciplina: Técnicas de Construção de Programas

Porto Alegre
2025

FASE I

Introdução

O trabalho propõe a implementação de um sistema que funcione como um gerador de música. A partir de uma entrada de texto, o software irá produzir notas correspondentes a caracteres específicos, seguindo uma série de parâmetros previamente definidos. Os integrantes do grupo decidiram utilizar a linguagem Python para a construção do programa, com auxílio de bibliotecas como (MIDIUtil 1.21, pygame 2.6.0, entre outras).

Requisitos do Sistema

Algumas das funcionalidades e a sua respectiva ideia de implementação estão descritas na lista abaixo:

Funcionalidades:

Cada caractere do texto será lido e interpretado pelo programa, a fim de realizar a conversão para uma ação musical. Classes diferentes de ações terão módulos próprios implementados no código. Alguns exemplos:

Leitor de texto:

O texto do usuário é lido e convertido para uma lista de caracteres que podem ser utilizados pelos programa posteriormente para tocar notas e assim criar uma música.

Dicionário de mapeamento:

A implementação de um dicionário (pode ser alterado, com adição, exclusão, update) que mapeia um caractere para uma ação musical específica como, por exemplo:

Ler(Letra A) → Toca(Nota Lá);
Ler(‘!’) → TrocaInstrumento(MIDI 24);
Ler(SPACE) → AlteraVolume(DobraVolume/VolumeMáximo).

Tocar notas: (Letras A,B,C,D,E,F,G,H)

O caractere é lido e a nota respectiva a uma das letras é tocada. Consulta o dicionário de mapeamento fornecido para realizar essa ação.

Troca de instrumento: (!, O/o, I/i, U/u, NL, “;” , pares e ímpares, “,”)

O programa analisa o símbolo e chama uma função que controla o instrumento atual (General MIDI) que está tocando e verifica se é possível realizar uma alteração para um outro.

Controle de volume e tonalidade: (Espaço, “?”, “.”)

Ao verificar esses caracteres, realiza uma verificação para dobrar o volume, caso isso não ocorra, coloca o volume no máximo. Também pode aumentar uma oitava ou voltar para a oitava default.

Exportar arquivo de resultado:

Após tocar a música correspondente a conversão do texto do usuário, também é possível salvar esta em um arquivo para uso futuro em outras aplicações. A lista de caracteres convertida e formatada também está disponível neste arquivo.

Classes previstas

Serão definidas classes para ajudar na organização e legibilidade do código. Essas serão divididas em módulos para guardarem diferentes dados e executarem múltiplas operações, além de ajudar a separar funcionalidades diferentes dentro do software.

GeradorMusica:

A classe GeradorMusica guarda informações sobre o programa de uma forma mais generalizada, coordenando ações a serem feitas, rotinas a serem chamadas e controle de erros da operação. Essa classe terá uma visão ampla do programa e será responsável pelo seu funcionamento correto, na ordem especificada e armazenando dados importantes sobre a execução.

Atributos: textoEntrada(string), textoConvertido(string), estadoLeitor(int), estadoExecucao(int).

Métodos: iniciarExecucao(), interromperExecucao(), tratarErros(), atualizaEstado().

LeitorTexto:

A classe LeitorTexto é responsável pela leitura do arquivo de texto fornecido pelo usuário no início do programa. Ela será encarregada de transformar a entrada bruta de dados em informações manipuláveis pelo programa, mantendo todos os caracteres do texto original, como letras, espaços, pontuações, etc. Desse jeito, o software poderá manipular mais eficientemente os dados fornecidos.

Atributos: textoInput(string), listaCaracteres(lista de char).

Métodos: lerTexto(), formatarTexto(), obterCaractere(), armazenaCaractere().

TransformaMusica:

A classe TransformaMusica recebe caracteres já formatados e prontos para serem convertidos para comandos musicais, realizando um mapeamento pré-definido de objetos como notas, pausas, alteração de instrumento/volume para gerar uma lista completa de quais destes serão reproduzidos em forma de sons para o usuário do programa.

Atributos: listaEntrada(lista de char), listaEventos(lista de eventos), mapaTransformacao(MapaCaracteres).

Métodos: converteCaracteres(), transformaEvento(), obterEvento().

MapaCaracteres:

A classe auxiliar MapaCaracteres ajuda a classe TransformaMusica a fazer um mapeamento coerente dos caracteres para as ações musicais. Nela, está contido o dicionário de caracteres e suas respectivas correspondências de notas, assim esse pode ser facilmente alterado no caso de novas implementações de notas ou de remoção/atualização destes.

Atributos: dicionarioMapeamento(lista caracteres/ lista eventos).

Métodos: obterAcao(), atualizarCaractere(), excluirCaractere(), adicionarCaractere(), mostraMapa().

ControleMusica:

A classe ControleMusica gerencia o estado atual dos parâmetros dos sons, como o volume, a oitava e o instrumento. Também controla a possibilidade de alteração desses valores, retornando ‘true’ em caso positivo e ‘false’ em caso negativo. Assim, a consistência da música será mantida independentemente das mudanças de características solicitadas.

Atributos: instrumentoAtual(int), volumeAtual(int), oitavaAtual(int).

Métodos: alterarInstrumento(), alterarVolume(), verificaVolume(bool), dobraVolume(), alterarOitava(), verificaOitava(bool), setOitavaDefault().

<u>TocadorNotas</u>:

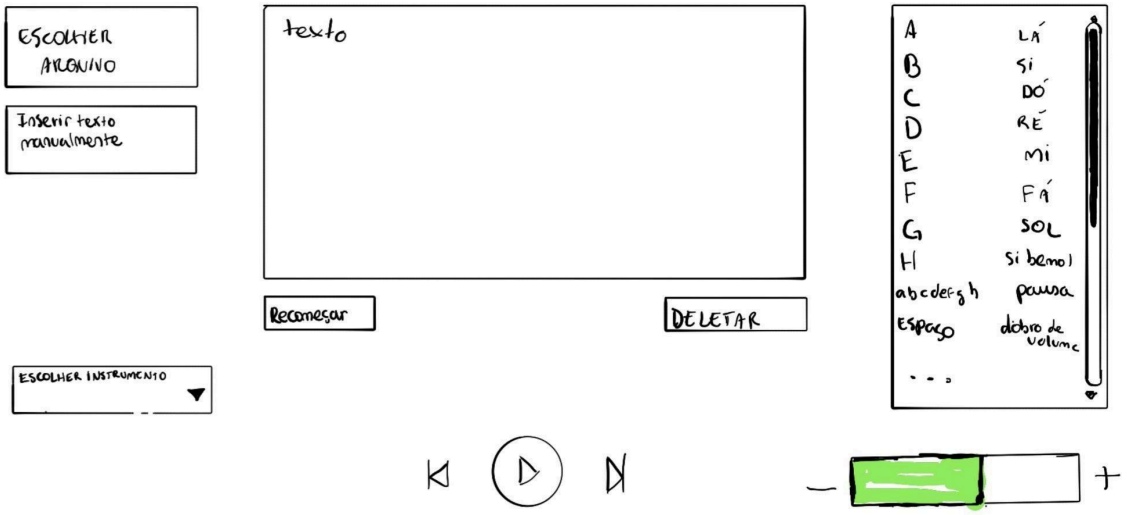
A classe TocadorNotas tem como ação principal tocar a nota que foi convertida do texto para o usuário poder ouvi-la. Também recebe informações de ControleMusica de modo que possa allterar as características necessárias dessa nota/som. Com isso, o usuário poderá finalmente transformar seu texto de entrada em uma música.

Atributos: playerAudio(biblioteca), estadoMusica(int).

Métodos: selecionaEvento(), selecionaSom(), exportaMusica(arquivo).

Croqui da interface

Abaixo, está um croqui que representa uma ideia da implementação da interface no software proposto no trabalho.



FASE II

INTRODUÇÃO

De acordo com o texto de Bertrand Meyer, Modularidade é a decomposição de um sistema em partes autônomas e coesas, chamadas de módulos, com o objetivo de reduzir a complexidade e aumentar a manutenibilidade.

A Modularidade deve ser avaliada por critérios como:

- **Decomposição**: o sistema deve ser dividido em módulos que possam ser compreendidos isoladamente;
- **Composição**: os módulos devem poder ser combinados para formar o sistema completo;
- **Compreensibilidade**: cada módulo deve ter um propósito claro;
- **Continuidade e Proteção**: mudanças locais devem afetar o mínimo possível outros módulos;
- **Desempenho e independência**: a estrutura modular não deve comprometer a execução do sistema.

No contexto do trabalho, o projeto **“Gerador de Música a partir de Texto”** foi estruturado seguindo esses princípios, com classes que representam claramente funções independentes, como a leitura de texto, o mapeamento de caracteres, a conversão musical e o controle de execução, garantindo assim uma base modular sólida para o programa.

O sistema tem como objetivo converter um texto livre em uma sequência musical, interpretando cada caractere como uma nota, pausa ou algum outro evento musical (como por exemplo: alteração de instrumento, de oitava ou de volume). Essas transformações irão se basear em um mapeamento entre caracteres e eventos musicais definidos segundo a especificação proposta no enunciado do trabalho.

CLASSE GERADORMUSICA

A classe **GeradorMusica** é o controle geral do programa. Ela é responsável por controlar o fluxo de execução, atualizar estados e tratar erros.

```python
class GeradorMusica:
    def __init__(self):
        self.estador_leitor = 0  # 1 para leitor em execução, 0 para leitor fora de execução
        self.estado_execucao = 0 # 1 para programa em execução, 0 para programa fora de execução

    def iniciarExecucao(self):
        if self.estado_execucao:
            print('O programa já esta em execução')

        else:
            print('Iniciando a execução do programa')
            self.estado_execucao = 1

    def interromperExecucao(self):
        if not (self.estado_execucao):
            print('O programa não está executando')

        else:
            print('Interrompendo a execução do programa')
            self.estado_execucao = 0

    def atualizaEstado(self):
        #recebe os estados dos métodos das outras classes, verifica se é possivel a execução e atualiza os estados
        ...

    def tratarErros(self):
        #verifica se foi recebido alguma flag indicando erros e, em caso positivo, toma as medidas para corrigir esses erros
        ...
```

Seus atributos são:

- **estado_leitor**: Recebe o estado de execução da classe LeitorTexto.
- **estado_execucao**: Define o estado de execução do programa como um todo.

Seus métodos são:

- **iniciarExecucao()**: Inicia a execução do fluxo do programa.
- **interromperExecucao()**: Interrompe a execução do fluxo do programa.
- **atualizaEstado()**: Recebe os estados dos métodos das outras classes, verifica se é possível a execução e atualiza os estados.
- **tratarErros()**: Verifica se foi recebido alguma flag indicando erros e, em caso positivo, toma as medidas para corrigir esses erros.

CLASSE LEITORTEXTO

A classe **LeitorTexto** é responsável por ler o texto de entrada e convertê-lo para uma lista de caracteres para melhor manipulação dos dados.

```python
class LeitorTexto:
    def __init__(self, texto_input: str):
        if texto_input is None or texto_input == "":
            self.texto_input = None
            self.lista_caracteres = None
        else:
            self.texto_input = texto_input
            self.lista_caracteres = list(texto_input)

    def exibirTexto(self):
        if self.texto_input is not None:
            print(self.texto_input)
        else:
            print('Texto inválido.')

    def listarCaracteres(self):
        if self.lista_caracteres is not None:
            print(self.lista_caracteres)
        else:
            print('Lista inválida.')
```

Seus atributos são:

- **texto_input**: Armazena o texto de entrada do programa recebida pelo usuário.
- **lista_caracteres**: Lista de caracteres convertida do texto de entrada.

Seus métodos são:

- **exibirTexto()**: Mostra o texto inserido pelo usuário previamente.
- **listarCaracteres()**: Mostra a lista de caracteres convertida do texto.

CLASSE TRANSFORMAMUSICA

A classe **TransformaMusica** recebe uma lista de entrada e um mapa de transformação. Em seguida converte os caracteres da lista de acordo com o mapa de transformação e coloca para uma lista de saída (eventos correspondentes dos caracteres).

```
class TransformaMusica:
    def __init__(self, lista_entrada, mapa_transformacao):
        self.lista_entrada = lista_entrada
        self.mapa_transformacao = mapa_transformacao
        self.lista_saida = []

    def converteCaracteres(self):
        self.lista_saida = []
        for caractere in self.lista_entrada:
            nota_atual = self.mapa_transformacao.get(caractere, -1)
            self.lista_saida.append(nota_atual)
```

Seus atributos são:

- **lista_entrada**: Lista de entrada que vai ser manipulada pela classe.
- **mapa_transformacao**: Dicionário que será utilizado para converter os caracteres para seus respectivos eventos.
- **lista_saida**: Lista de saída com os eventos que foram convertidos da lista de entrada usando o mapa de transformação.

Seu método é:

- **converteCaracteres()**: Avalia cada caractere individualmente e transforma este em um evento, depois o coloca na lista de saída.

CLASSE MAPACARACTERES

A classe **MapaCaracteres** faz o mapeamento de um caractere para um evento, podendo ser um caractere individual ou uma lista de caracteres. Também pode adicionar novos caracteres, assim como excluí-los. Ela pode receber ou uma lista de caracteres e uma lista de eventos, ou um caractere e um evento, ou um caractere sozinho (para exclusão).

```python
class MapaCaracteres:
    def __init__(self, lista_caracteres=None, lista_eventos=None, caractere=None, evento=None):
        self.dicionario_mapeamento = {}

        if lista_caracteres is not None and lista_eventos is not None:

            if len(lista_caracteres) == len(lista_eventos):
                self.mapeiaCaracteres(lista_caracteres, lista_eventos)
            else:
                print('As listas de caracteres e eventos devem ter o mesmo tamanho.')

        if caractere is not None and evento is not None:
            self.adicionaCaractere(caractere, evento)

    def mapeiaCaracteres(self, lista_caracteres, lista_eventos):
        if len(lista_caracteres) == len(lista_eventos):
            for i in range(len(lista_caracteres)):
                caractere = lista_caracteres[i]
                evento = lista_eventos[i]
                self.dicionario_mapeamento[caractere] = evento

        else:
            print('As listas devem ter o mesmo tamanho.')

    def adicionaCaractere(self, caractere, evento):
        if caractere is None or evento is None:
            print('Caractere e evento não podem ser nulos.')
        else:
            self.dicionario_mapeamento[caractere] = evento
            print(f'Caractere "{caractere}" adicionado.')

    def excluiCaractere(self, caractere):
        if caractere in self.dicionario_mapeamento:
            self.dicionario_mapeamento.pop(caractere)
            print(f'Caractere "{caractere}" excluído.')
        else:
            print(f'Caractere "{caractere}" não encontrado.')

    def mostraMapa(self):
        if not self.dicionario_mapeamento:
            print('O mapa está vazio.')
        else:
            for chave, valor in self.dicionario_mapeamento.items():
                print(f'{chave} --> {valor}')
```

Seus atributos são:

- **lista_caracteres**: A lista de caracteres que será mapeada 1:1 com a lista de eventos.
- **lista_eventos**: A lista de eventos com a qual lista_caracteres será mapeada.
- **caractere**: Um caractere individual que foi passado para a classe.
- **evento**: Um evento individual que será mapeado com o caractere.

Seus métodos são:

- **mapeiaCaracteres()**: Se foram recebidas uma lista de caracteres e uma lista de eventos de mesmo tamanho, vai mapear 1:1 os caracteres e os eventos (em ordem crescente).

- **adicionaCaractere()**: Se foram recebidos apenas um caractere e apenas um evento, vai adicionar o caractere com seu respectivo evento.
- **excluiCaractere()**: Se for recebido apenas um caractere, vai excluí-lo do mapeamento, junto com o seu evento correspondente.
- **mostraMapa()**: Apresenta o atual mapeamento de caracteres para eventos.

CLASSE CONTROLEMUSICA

A classe **ControleMusica** serve para controlar propriedades da música, como o instrumento, o volume e a oitava. Também possui um parâmetro de volume máximo e oitava máxima.

```python
class ControleMusica:
    def __init__(self, instrumento_atual, volume_atual, oitava_atual):
        self.instrumento_atual = instrumento_atual
        self.volume_atual = volume_atual
        self.oitava_atual = oitava_atual
        self.max_oitava = 7
        self.oitava_default = 1
        self.max_volume = 100

    def alterarInstrumento(self, instrumento):
        if instrumento == self.instrumento_atual:
            print(f'O instrumento "{instrumento}" já esta em uso.')

        else:
            self.instrumento_atual = instrumento

    def alterarVolume(self):
        if self.volume_atual * 2 > self.max_volume:
            self.volume_atual = self.max_volume
            print('O volume foi aumentado para o máximo')

        else:
            self.volume_atual = self.volume_atual * 2
            print('O volume foi dobrado')

    def alterarOitava(self):
        if self.oitava_atual == self.max_oitava:
            self.oitava_atual = self.oitava_default

        else:
            self.oitava_atual += 1
```

Seus atributos são:

- **instrumento_atual**: O instrumento que está sendo utilizado para tocar as notas.
- **volume_atual**: O volume no qual as notas estão sendo tocadas.
- **oitava_atual**: A oitava na qual as notas estão sendo tocadas.
- **oitava_default**: A oitava inicial do programa.
- **max_volume**: O maior volume possível em que as notas podem ser tocadas.

Seus métodos são:

- **alterarInstrumento()**: Altera o instrumento atual para outro instrumento.

- **alterarVolume()**: Dobra o volume se for solicitado, entretanto, se essa ação for passar do volume máximo, define o volume para o volume máximo.

- **alterarOitava()**: Aumenta uma oitava na oitava atual e, no caso de não ser possível aumentar a oitava, reseta para o valor da oitava default.

CLASSE TOCADORNOTAS

A classe **TocadorNotas** reproduz os sons que foram baseados no texto para que o usuário possa ouví-los. Ela utiliza um player para fazer essa ação e necessita saber qual instrumento, volume e oitava foram selecionados pela classe **ControleMusica**, assim como a nota que irá tocar.

```python
class TocadorNotas:
    def __init__(self, estado_player, nota, instrumento, volume, oitava):
        self.estado_player = estado_player
        self.nota = nota
        self.instrumento = instrumento
        self.volume = volume
        self.oitava = oitava

    def tocaSom(self, instrumento, volume, oitava):
        # executa comando da biblioteca para tocar uma NOTA com o INSTRUMENTO, com as configurações de VOLUME X e OITAVA Y
        ...
```

Seus atributos são:

- **estado_player**: Estado atual do player: tocando, pausado, interrompido.
- **nota**: Nota a ser tocada.
- **instrumento**: Instrumento que será utilizado para tocar a nota
- **volume**: Volume com o qual a nota será tocada.
- **oitava**: Oitava na qual a nota será tocada.

Seu método é:

- **tocaSom()**: Executa um comando da biblioteca para tocar uma NOTA com o INSTRUMENTO, com as configurações de VOLUME X e OITAVA Y.

INTERFACE

A interface ainda não foi implementada, mas o design proposto inclui:

- Um campo de texto para entrada;
- Botões: **Gerar Música**, **Configurações**, **Mapeamento**, **Exportar Resultado**;
- Cada botão abrirá uma tela funcional.

CRITÉRIOS DE MODULARIDADE

Critério de Meyer	Como é atendido no programa	Exemplo no código
Decomposição	O sistema é dividido em seis classes independentes com papéis claros.	Classes LeitorTexto, TransformaMusica, ControleMusica etc.
Compreensibilidade	Cada módulo tem propósito único e nome autoexplicativo.	MapaCaracteres → mapeamento de símbolos musicais.
Proteção (Information Hiding)	Atributos internos são acessados apenas via métodos.	MapaCaracteres manipula dicionario_mapeamento apenas internamente.
Continuidade	Alterar uma parte (como a leitura do texto) não afeta outras (como controle de volume).	LeitorTexto e ControleMusica não têm dependências diretas.
Independência / Baixo acoplamento	As classes comunicam-se por passagem de parâmetros e não compartilham estados globais.	TransformaMusica recebe mapa_transformacao externamente.
Extensibilidade	O design permite facilmente adicionar novos tipos de mapeamentos ou instrumentos.	Nova função exportaMusica() poderá ser inserida em TocadorNotas.
Modularidade funcional	Cada classe representa uma função distinta do processo musical.	ControleMusica controla parâmetros; TocadorNotas executa som.

PROBLEMAS

Problema identificado	Proposta de solução	Vantagens	Possíveis desvantagens
Integração ainda ausente entre TransformaMusica e ControleMusica.	Criar interface entre ambas, para que a conversão de caracteres leve em conta o volume, oitava e instrumento.	Torna a música dinâmica e realista.	Aumenta a complexidade de integração.
Falta método para exportar resultados musicais.	Criar exportaMusica() na classe TocadorNotas para gerar arquivos .mid.	Permite reuso e compartilhamento de músicas.	Requer padronização e manipulação de arquivos.
Métodos atualizaEstado() e tratarErros() ainda não implementados.	Implementar tratamento de exceções e sincronização de estado.	Melhora robustez e controle de execução.	Pode exigir mais validações internas.
Falta de integração central entre módulos.	Criar uma função principal que conecte todas as classes (GeradorMusica coordena o fluxo).	Garante execução sequencial completa.	Introduz necessidade de testes de sistema.

CONCLUSÃO

Com base nos critérios de Meyer, o projeto atual segue as regras de modularidade, há decomposição adequada, baixo acoplamento e encapsulamento correto.

A evolução natural nas próximas fases envolverá melhorar a composição (integração entre módulos) e refinar a proteção (tratamento de erros e consistência de estados), assim como implementar novas funcionalidades ou atualizar funcionalidades existentes propostas na fase 3.

FASE III

DEFINIÇÃO DAS CLASSES

As classes finais do trabalho foram as seguintes:

- LeitorTexto;
- TransformaMusica;
- TocadorNotas;
- GeradorMusica;
- AppGUI.

Classe LeitorTexto:

Atributos:

texto, caminho_arquivo

Métodos:

receberTextoEscrito(self, texto_digitado)
abrirArquivo(self)
salvarArquivo(self)
salvarComo(self)

Classe TransformaMusica:

Atributos:

lista_eventos

Métodos:

processarTexto(self, texto_entrada)

Classe TocadorNotas:

Atributos:

player, inicializado, bpm_atual, volume_atual, oitava_atual, instrumento_atual_id, tocando, evento_pausa

Métodos:

atualizarConfiguracao(self, bpm, volume, oitava, instrumento_id)
restaurarEstadoInicial(self)
fechar(self)
pararExecucaoCompleta(self)
pausarMusica(self)
despausarMusica(self)
calcularBPM(self)
tocarNota(self, nome_nota, duracao)
ajustarBPM(self, acao)
ajustarVolume(self, acao)
ajustarOitava(self, acao)
trocarInstrumento(self, valor)
somarInstrumento(self, valor)
pausarTempo(self, duracao)
tocarEventos(self, lista_eventos, callback_fim=None)

Classe GeradorMusica:

Atributos:

leitor, transformador, tocador, thread

Métodos:

carregar_arquivo_texto(self)
salvar_arquivo_texto(self, texto_atual)
tocar_musica(self, texto, config, callback_fim)
pausar_musica(self)
despausar_musica(self)
parar_musica(self)
parar_e_resetar_musica(self)
fechar_programa(self)
gerar_midi(self, texto, config)

Classe AppGUI:

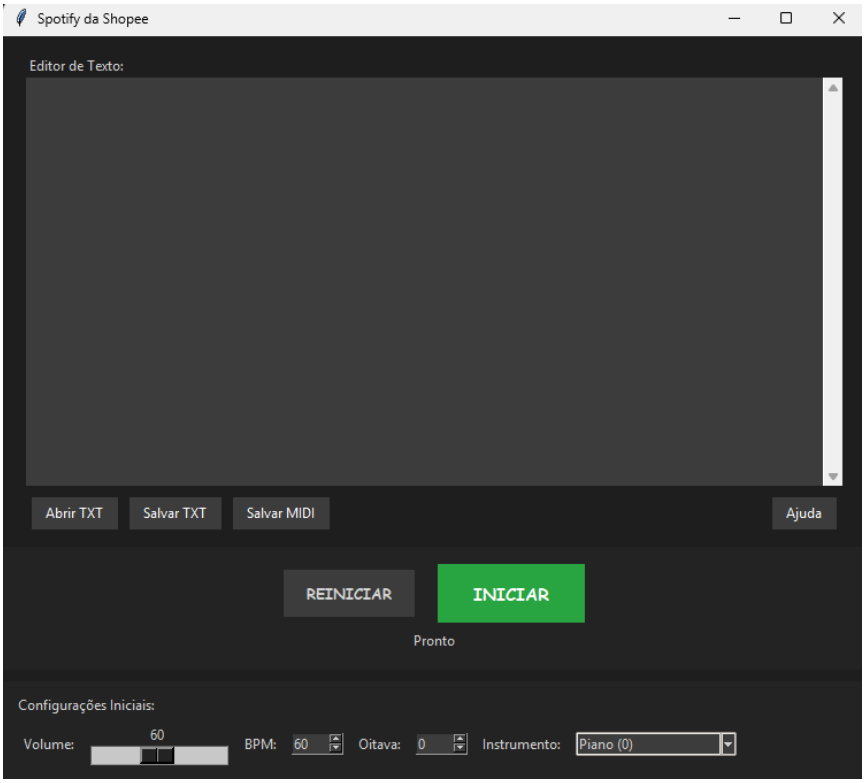
Atributos:

estado_player, root, controlador

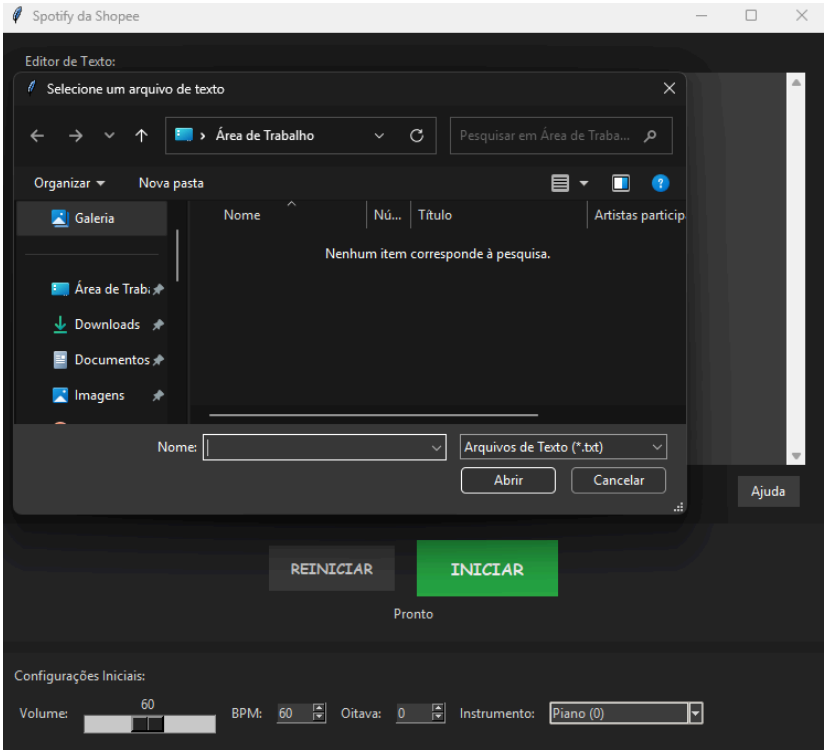
Métodos:

setup_janela(self)
criar_componentes(self)
criar_botao(self, parent, texto, comando)
criar_input_config(self, parent, label, col, valor_padrao, v_min, v_max)
acao_botao_principal(self)
iniciar_musica(self)
pausar_musica(self)
continuar_musica(self)
acao_reiniciar(self)
callback_fim_musica(self)
resetar_interface(self)
acao_abrir(self)
acao_salvar_texto(self)
acao_salvar_midi(self)
acao_ajuda(self)
get_configs(self)
on_closing(self)

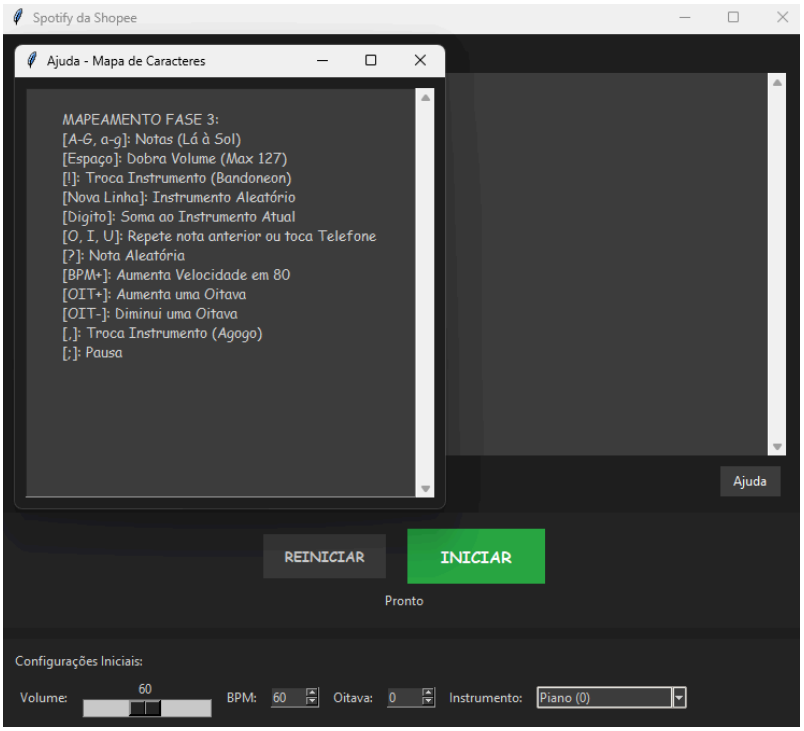
INTERFACE



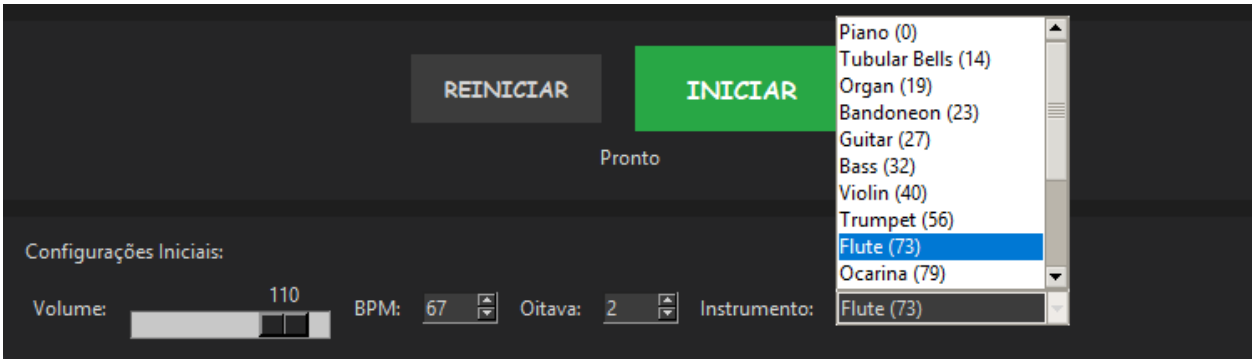
Interface base do programa, como o editor de texto, botões de manipulação de arquivos, botões de controle de estado da música e botões de configurações iniciais.



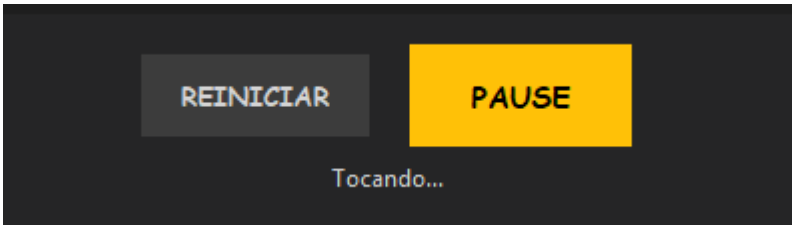
Interface de seleção de arquivo de texto.



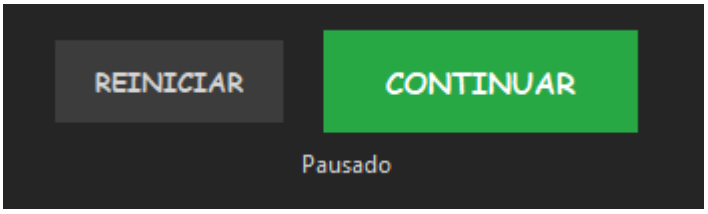
Interface do botão de mapa de caracteres.



Interface de configurações iniciadas com valores escolhidos e lista de instrumentos.



Interface com botão de pausa.



Interface com botão de continuar.

PROCEDIMENTOS DE TESTE

Os testes foram realizados de forma unitária durante o desenvolvimento do código. Cada classe teve testes práticos individuais, o que ajudou a detectar erros e confirmar o comportamento esperado ao longo da implementação.

Para a classe LeitorTexto, foram feitos testes básicos de abertura de arquivos .txt, salvamento de textos em novos arquivos e leitura de textos digitados, garantindo que o conteúdo fosse armazenado e retornado corretamente.

Para a classe TransformaMusica, foram testadas conversões simples de caracteres em eventos musicais. Foram usados textos contendo notas (A–G), comandos especiais como BPM e oitava, além de casos com espaços, dígitos e símbolos, verificando se a lista de eventos gerada correspondia ao esperado.

Para a classe TocadorNotas, foram feitos testes de cálculo de BPM, mudança de instrumento, ajuste de oitava e volume. Também foram feitos testes simples de tocar uma única nota e executar uma pequena sequência de eventos, verificando se as funções eram chamadas sem erros.

Para a classe GeradorMusica, foram realizados testes de execução da música. Também foi testada a criação de um arquivo MIDI simples, verificando se o fluxo era concluído sem falhas.

Para a classe AppGUI, foram testadas as funções principais da interface: abrir texto, salvar texto, iniciar a execução da música, pausar e reiniciar. Esses testes foram feitos manualmente através da própria interface para confirmar que cada botão funcionava corretamente os métodos do controlador.