

Universidade Federal do Rio Grande do Sul

João Kenji Suwa - 587808

Felipe Pasinato Rossoni - 587631

Otávio Rosa de Oliveira - 588807

Fase II do Trabalho Prático

Disciplina: Técnicas de Construção de Programas

Porto Alegre

2025

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.

```
class GeradorMusica:
    def __init__(self):
        self.estado_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 possível 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 LEITORTEXT0

A classe **LeitorText0** é responsável por ler o texto de entrada e convertê-lo para uma lista de caracteres para melhor manipulação dos dados.

```
class LeitorText0:
    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 exibirText0(self):
        if self.texto_input is not None:
            print(self.texto_input)
        else:
            print('Text0 inválid0.')

    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:

- **exibirText0()**: 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).

```
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.

```
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 ouvi-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.

```
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.