UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMATICA

CAIO FELIPE FERREIRA NUNES
GUSTAVO MACHADO SILVA
LUCAS ROSSI KLEIN
YASMIN AGNES SIMÃO

TRABALHO PRÁTICO
FASE 2

Disciplina: Técnicas de Construção de Programas (INF01120)
Professor: Marcelo Soares Pimenta

Porto Alegre, outubro de 2025.

Contents

1 INTRODUÇÃO

Nesta etapa do trabalho prático da disciplina, temos como objetivo avaliar a modularidade do Software que estamos desenvolvendo. Para isso, primeiro apresentamos uma descrição da aplicação desenvolvida, logo após, apresentamos brevemente o que é cada critério, regra e princípio da modularidade, discutimos como nosso sofware atente ou não este requisito e apresentamos um exemplo que justifique as conclusões. Por fim avaliamos os resultados obtidos na etapa anterior e, para cada critério, regra ou princípio não atendido, propomos uma solução, apontado as vantagens e desvantagens de sua aplicação no código.

Modularidade é um conceito importante para a construção de software, uma vez que está presente nos esforços para garantir diversos fatores de qualidade. Entretanto, não é simples de ser definida ou aplicada, é necessária prática para um programdor conseguir inserir, de forma eficiente em seu software, todos os conceitos que serão apresentados ao longo deste relatório. Portanto, a realização desta etapa foi proveitosa para os autores, tanto avaliar e melhorar sua aplicação, quanto desenvolver suas habilidades de programação modularizada.

2 DESCRIÇÃO DO PROGRAMA

O objetivo do programa é converter um texto em uma composição musical, atribuindo a cada caractere do texto recebido em uma nota musical ou uma modificação em algum aspecto da música, como configurações e propriedades da música. Dessa forma, o programa traduz informações textuais em elementos sonoros a partir da sequência de caracteres.

A arquitetura foi pensada com foco na clareza das operações musicais e na extensibilidade, permitindo a escalabilidade do programa. A seguir, são apresentadas as principais classes que compõem o sistema, seus objetivos, atributos e métodos já definidos até o momento.

2.1 TextHandler

A classe TextHandler é responsável por lidar com a string inserida pelo usuário, controlando a leitura sequencial dos caracteres que serão convertidos em elementos musicais. Ela possui dois atributos principais: text, que armazena o texto completo a ser processado, e currentChar, que representa o caractere atual em análise. Para manipular esses dados, a classe oferece os métodos getCurrentChar(), que retorna o caractere atual da leitura, e hasNextChar(), que verifica se ainda há caracteres disponíveis para leitura na sequência.

2.2 Instrument

A classe Instrument é responsável por armazenar as propriedades que identificam um instrumento musical, incluindo os tipos disponíveis no software. Seus principais atributos são name, que representa o nome do instrumento, e idMIDI, que armazena sua identificação MIDI, usada na reprodução sonora. Os métodos getIdMIDI() e getName() permitem acessar, respectivamente, o identificador MIDI e o nome do instrumento.

2.3 Note

A classe Note define as propriedades fundamentais de uma nota musical, sendo responsável por representar informações musicais essenciais como oitava, duração e nome da nota. Possui os atributos octave, que indica a oitava da nota, duration, que define sua duração em tempo, e name, que representa a nota como caractere. Os métodos getOctave(), getDuration() e getName() permitem acessar essas propriedades individualmente.

2.4 Volume

A classe Volume é encarregada de controlar as configurações relacionadas ao volume da música. Ela armazena dois atributos principais: currentVolume, que representa o

volume atual, e maxVolume, que define o limite máximo permitido. Para interagir com esses valores, a classe oferece os métodos getVolume(), que retorna o volume atual, e setVolume(), que permite atualizar esse valor.

2.5 MusicalContext

A classe MusicalContext tem como responsabilidade manter e gerenciar o estado musical atual da aplicação, permitindo a obtenção e atualização de informações relevantes. Os atributos currentInstrument, currentVolume e currentNote armazenam, respectivamente, o instrumento, o volume e a nota musical em uso no momento. A classe fornece métodos de acesso e modificação para cada um desses atributos, como getCurrentInstrument() e setCurrentInstrument(), getCurrentVolume() e setCurrentVolume(), além de getCurrentNote() e setCurrentNote().

2.6 Action

A classe abstrata Action representa uma ação que modifica uma ou mais propriedades do contexto musical. Ela funciona como uma base para classes filhas que implementam comportamentos específicos, como mudar a nota, o volume ou o instrumento. Seu único atributo é context, uma instância de MusicalContext, que representa o estado musical a ser modificado. O método abstrato execute() deve ser implementado nas subclasses para executar a ação correspondente.

2.7 ActionMapper

A classe ActionMapper tem como objetivo principal mapear caracteres específicos para ações musicais que modificam o contexto sonoro. Para isso, utiliza o atributo actionKeys, um mapa (HashMap) que relaciona String a objetos do tipo Action. Seus métodos principais incluem assignActionToAKey() que define um novo mapeamento, updateActionKey() que atualiza ações já atribuídas, deleteActionKey() que remove um mapeamento, e getAction(), que retorna a ação associada a uma determinada chave.

2.8 Interface

Na primeira tela, o usuário visualiza três blocos: um para inserir o texto, outro com funções para configurar parâmetros musicais (como escala, tonalidade e BPM) e um terceiro com a opção de visualizar o som gerado ou acessar um guia de uso do sistema, conforme Figura 1.

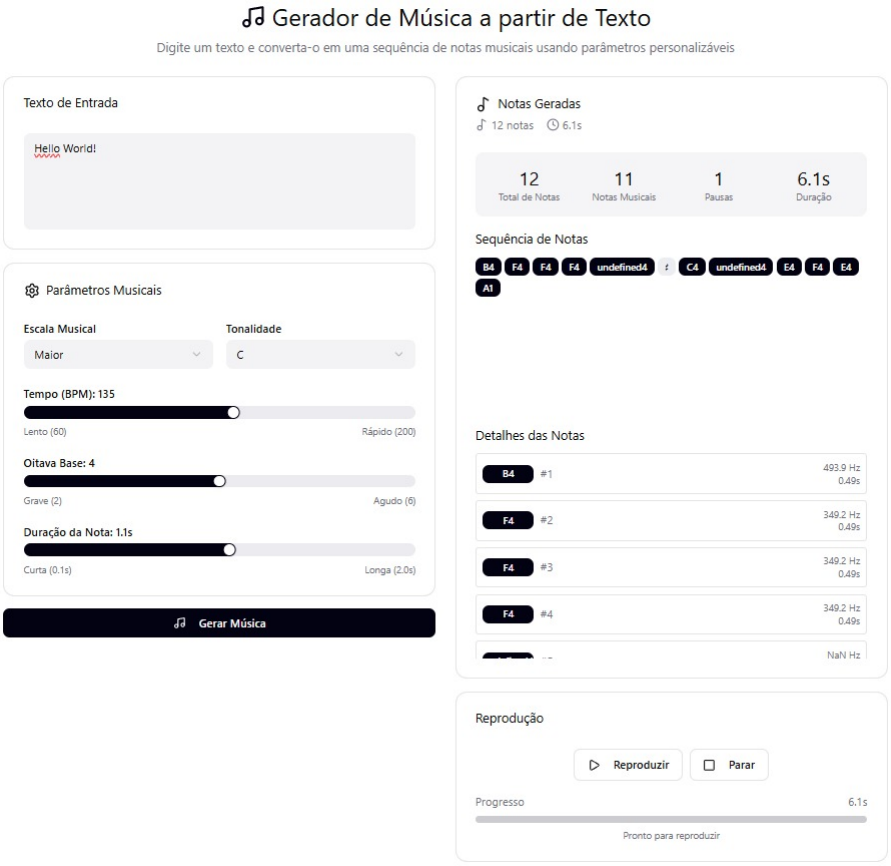

Figure 1: Tela Resultados - Com música gerada (Fonte: Os autores)

Ao selecionar o “Guia de Uso” o usuário consegue ter acesso às funcionalidades de cada caractere e um breve exemplo de uso, conforme Figura 2.

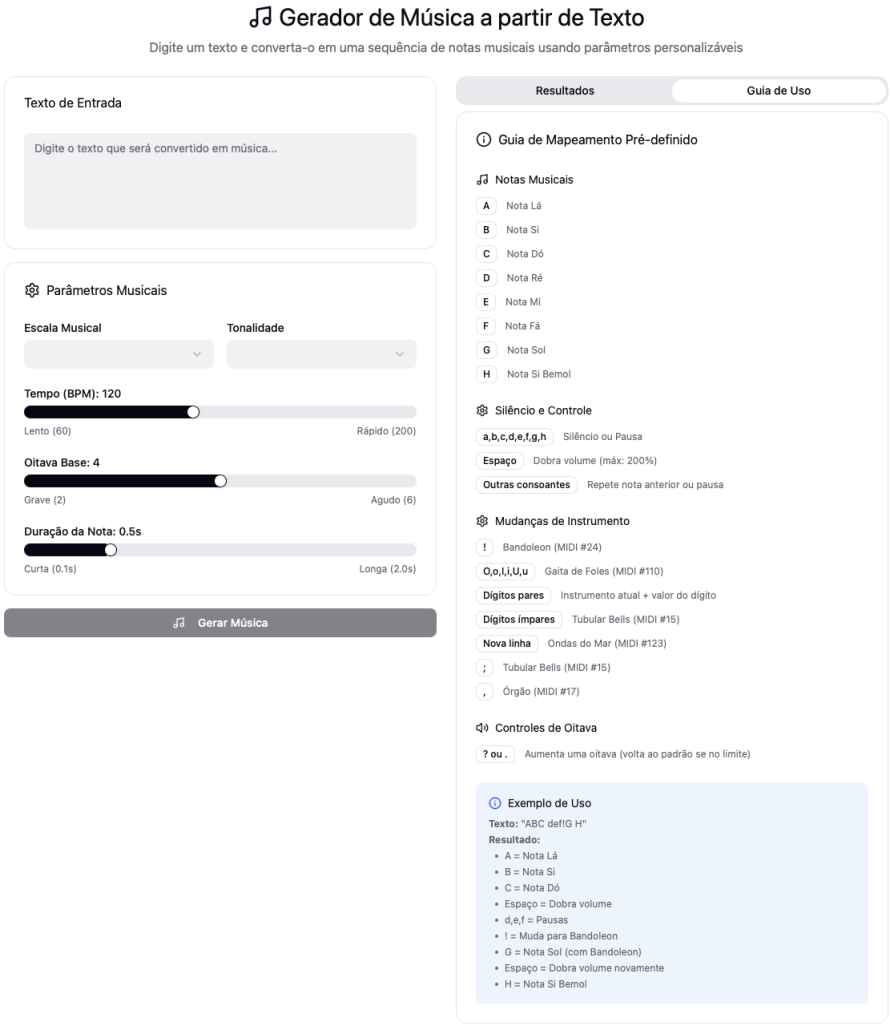

Figure 2: Tela Inicial - Guia de uso da aplicação (Fonte: Os autores)

3 AVALIAÇÃO DA MODULARIDADE DO PROGRAMA

Para fazermos uma avaliação da modularidade de um programa, primeiro precisamos definir quais os pesos a serem considerados nessa avaliação. Com base no texto "Modularity", fornecido pelo professor da disciplina, estabelecemos a nossa avaliação pelos critérios, regras e princípios de modularidade apresentados. Essa seção apresenta nossa avaliação junto com trechos do código para exemplificar nosso argumento.

3.1 Critérios da Modularidade

Definimos critérios de modularidade como os requisitos que definem se um método de construção de software pode ser considerado modular. Vamos abordar os cinco critérios e como eles são aplicados no software.

```
private void repeatOrPause(char c) {
    //testa se A ultima nota eh uma letra
    if (repeatCondition(c)) {
        playNote(lastNote, volume);
    } else pause();
}

private boolean repeatCondition(char c) {
    return (((lastChar >= 'A') &&
            (lastChar <= 'H')) &&
            !(c >= 'a' && c <= 'h'));
}
```

Listing 1: Trecho de código para analisar os 5 Critérios

3.1.1 Decomposability

O método de construção de software deve facilitar a divisão de rotinas em subrotinas menores e mais simples, com dependência mínima e estrutura clara. Nosso software atende, pois tentamos manter as funções curtas e objetivas.

3.1.2 Composability

Os módulos devem poder ser reutilizados em outros contextos sem dependências excessivas. Nosso software atende parcialmente a esse princípio, pois estamos focando na

decomposição dos problemas; no entanto, algumas partes acabam ficando específicas demais.

3.1.3 Understandability

Cada módulo deve ser compreensível de forma isolada, ou seja, sem precisar ter conhecimento de outros métodos so sistema. Nosso software atende, dividimos os módulos de maneira que cada parte atenda a seu objetivo específico.

3.1.4 Continuity

Leves mudanças nos requisitos do software devem causar mudanças em apenas poucos métodos ou somente um. Nosso software atende, novamente por conta da *Decomposability* o programa é composto por rotinas específicas; se alteradas, impactam poucos métodos.

3.1.5 Protection

Falhas no software devem estar contidas pelo método em que ocorrem, sem comprometer o sistema inteiro. Nosso software atende pelo mesmo motivo do critério *Continuity*.

3.2 Regras de Modularidade

As regras da modularidade são diretrizes práticas que derivam dos critérios, sendo voltadas ao controle das interações entre os módulos. Vamos abordar as cinco regras e como elas são aplicadas no software.

3.2.1 Direct Mapping

O software construído com uma estrutura modular, deve representar a estrutura modular do problema que o mesmo procura resolver. Nosso software atende esse requisito pois foi pensado com base na proposta de resolução do problema.

3.2.2 Few Interfaces

Cada módulo do sistema deve se comunicar com o menor número possível dos outros módulos. Nosso software atende esse requisito pois os módulos utilizam dependências somente de módulos necessários. Um exemplo é a classe Action, código 2 que se comunica apenas com a MusicalContext.

```
public class Action {

    private MusicalContext musicalContext;

    public Action() {
    }

    public Action(MusicalContext musicalContext) {
        this.musicalContext = musicalContext;
    }

    public MusicalContext getMusicalContext() {
        return musicalContext;
    }

    public void setMusicalContext(MusicalContext musicalContext) {
        this.musicalContext = musicalContext;
    }

    public void execute() {
        System.out.println("Action has not been defined!");
    }
}
```

Listing 2: Classe Action

3.2.3 Small Interfaces

Ao ser necessário uma comunicação entre módulos, a quantidade de informação trocada deve ser mínima. Nosso software atende esse requisito pois os módulos foram construídos mantendo suas propriedades internas e funções específicas de contrução do módulo privadas. Deixando público somente getters, setters e outras funções que o módulo busca oferecer. Um bom exemplo é a classe Action, código 2, que deixa o musicalContext privado, mas oferece getter e setter.

3.2.4 Explicit Interfaces

Todas as interações entre módulos devem estar explicitamente visíveis no código. Evitando acoplamento implícitos, como compartilhamento de oculto de dados. Nosso software atende esse requisito pois todas as interações utilizadas entre módulos podem ser

facilmente encontradas nos módulos.

3.2.5 Information Hiding

Cada módulo deve expor (deixar público) apenas os métodos necessários para integração e esconder seus detalhes de implementação. Nosso software atende esse requisito pois os módulos mantém suas propriedades privadas assim como suas funções de construção. Deixando público apenas os métodos necessários. Um bom exemplo é a classe Volume, código 3.

3.3 Princípios de Modularidade

Os princípios de modularidade são consequências das regras e são aplicáveis na implementação e evolução de sistemas. Vamos abordar os cinco princípios e como eles são aplicados no software.

3.3.1 Linguistic Modular Units

A linguagem utilizada no desenvolvimento do software deve oferecer suporte sintático para a definição e utilização de módulos. Nosso software atende este princípio, pois foi implementado na linguagem de programação Java, que oferece suporte para calsses, que, por sua vez, são os módulos da programação orientada a objetos. Como exemplo temos, em 3, a classe Volume, que é um módulo de nossa aplicação.

```
public class Volume {

    private int currentVolume;
    private int maxVolume;

    public Volume(int currentVolume, int maxVolume) {
        this.currentVolume = currentVolume;
        this.maxVolume = maxVolume;
    }

    public int getCurrentVolume() {
        return currentVolume;
    }

    public void setCurrentVolume(int currentVolume) {
        this.currentVolume = currentVolume;
    }

    public int getMaxVolume() {
        return maxVolume;
    }

    public void setMaxVolume(int maxVolume) {
        this.maxVolume = maxVolume;
    }
}
```

Listing 3: Classe Volume

3.3.2 Self-Documentation

Toda a informação necessária para compreensão do módulo deve estar contida dentro do próprio módulo. Nosso software atende parcialmente a este requisito, não utilizamos documentação externa às classes, todas podem ser entendidas por si só, uma vez que seus atributos e métodos são simples e diretos e possuem nomes significativos, porém ainda existem pequenos detalhes que não estão tão bem documentados, como valores não contidos em constantes. A seguir, em 4, temos um método que exemplifica as questões apontadas.

```
private static final int[] NOTE_VALUES = {
        69, // A (Lá)
        71, // B (Si)
        60, // C (Dó)
        62, // D (Ré)
        64, // E (Mi)
        65, // F (Fá)
        67, // G (Sol)
        70  // H (Si)
};}
```

Listing 4: Valores de cada nota

3.3.3 Uniform Access

O acesso aos elementos de um módulo devem ser feitos por uma notação padrão. Independentemente de como foram implementados. Nosso software atende a este princípio, pois utiliza o padrão da linguagem java, object.method(parameters) ou Class.method(parameters), para métodos estáticos, para acessar qualquer função realizada por um módulo. Assim, o cliente, isto é, quem precisa utilizar a função, apenas dá os parâmetros e recebe o resultado, sem se preocupar com o que acontece por dentro do método. Aqui contamos uma linha de código fr uma função sendo utilizada, apresentada em 5, como exemplo.

```
currentAction = actionMapper.getAction("a");
```

Listing 5: Chamada de função

3.3.4 Open-Closed Principle

Um módulo deve ser aberto para extensão e fechado para modificação. Ou seja, deve ser possível adaptar e estender o módulo sem que sejam necessárias alterações internas no módulo. Nosso software atende esse requisito e como exemplo podemos citar a classe ActionMapper, código 6, que é responsável por mapear ações a botões e devido a utilização de um hashMap, para cada nova ação a ser mapeada não é necessário alterar a classe.

```
public class ActionMapper {

    private Map<String, Action> actionKeys = new HashMap<>();

    public void assignActionToAKey(String key, Action action) {
        if (!actionKeys.containsKey(key)) {
            actionKeys.put(key, action);
        } else {
            System.out.println("This key has already been mapped!");
        }
    }
```

Listing 6: Classe ActionMapper

3.3.5 Single Choice Principle

Quando um sistema possui várias alternativas ou variantes, apenas um módulo deve conhecer e gerenciar uma lista completa delas. Nosso software atende, porque cada funcionalidade que o sistema precisa realizar é de responsabilidade de uma só classe, como por exemplo tratar do instrumento atual da música e alterar o instrumento utilizado são responsabilidade da classe Instrument implementada.

4 DISCUSSÃO E PROPOSTA DE SOLUÇÃO

Na seção anterior descrevemos como cada critério, regra e princípio da modularidade é ou não atendido pelo software desenvolvido até agora. Discutimos e trouxemos exemplos, o que nos levou a reler e revisar o código desenvolvido, e, a partir disso, encontrar meios de melhorá-lo.

Obteve-se um bom resutado na avaliação dos requisitos, uma vez que a maioria deles foi atendido pelo sistema. Isso se deve principalmente à construção atenta à modularidade, que foi aplicada desde a definição do modelo realizada na fase 1. Que por sua vez, veio dos estudos dos autores sobre desenvolvimento de software com qualidade, fortemente relacionado à modularidade, realizados durante a disciplina de técnicas de construção de programas. Entretanto a avalição não foi perfeita, a seguir serão apresentadas propostas de solução às questões encontradas.

4.1 Critério: Composability

Este é um critério apontado como parcialmente atendido pelo software. Isso ocorre pois o sistema foi construído com um foco maior em decomposabilidade, outro criério de modularidade. As classes foram divididas de modo a atender funcionalidades de um propósito maior, e, por isso, não ficaram gerais o suficiente para atender completamente este critério.

Para resolver o problema podemos reescrever algumas classes de modo a cumprir funções mais genéricas. A vantagem desta abordagem é melhorar a reusabilidade de nossos módulos, porém existe a desvantagem de precisar reescrever partes do código e perder um pouco da eficiência que um módulo mais específico tem de resolver aquele problema.

4.2 Princípio: Self-Documentation

Este é um princípio de modularidade apontado como parcialmente atendido. Este caso não corresponte a uma questão geral da aplicação, mas sim a pequenas questões presentes ao longo do código. É comum encontrar alguns erros neste sentido em um programa recém desenvolvido, por isso a revisão feita na seção 3 foi importante, e agora os autores são capazes de resolver as questões.

Em 7, apresentamos a proposta de solução do exemplo 4, que consiste em transformar os valores das notas em constantes e especificar o comentário no nome, para melhorar a documentação e, consequentemente, o entendimento daquele elemento dentro da classe.

```
public class Note {

    public static final int MIDI_A_5_OCTAVE = 69;
    public static final int MIDI_B_5_OCTAVE = 71;
    public static final int MIDI_C_5_OCTAVE = 60;
    public static final int MIDI_D_5_OCTAVE = 62;
    public static final int MIDI_E_5_OCTAVE = 64;
    public static final int MIDI_F_5_OCTAVE = 65;
    public static final int MIDI_G_5_OCTAVE = 67;
    public static final int MIDI_H_5_OCTAVE = 70;

    private int octave;
    private float duration;
    private String name;

    public Note() {
    }
    // Notas MIDI baseadas na oitava 5
    private static final int[] NOTE_VALUES = {
            MIDI_A_5_OCTAVE // Lá
            MIDI_B_5_OCTAVE // Si
            MIDI_C_5_OCTAVE // Dó
            MIDI_D_5_OCTAVE // Ré
            MIDI_E_5_OCTAVE // Mi
            MIDI_F_5_OCTAVE // Fá
            MIDI_G_5_OCTAVE // Sol
            MIDI_H_5_OCTAVE // Si
    };
}
```

Listing 7: Proposta de solução na classe Note