Aplicação dos Princípios SOLID no Desenvolvimento de um Metrônomo

Guilherme Santos de Godoy — 758710
Programação Orientada a Objetos Avançada - Prof. Delano
Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
São Carlos – SP – Brasil

Índice

1. Apresentação do Trabalho **3**

2. Conceitos Básicos **3**
2.1. Metrônomo 3
2.2. Compasso 3
2.3. Batida 3

3. Classes e Funções **4**
3.1. Classes 4
3.2. Funções 4

4. Aplicação dos Princípios SOLID **5**
4.1. Princípio da Responsabilidade Única (SRP) 5
4.2. Princípio do Aberto/Fechado (OCP) 6
4.3. Princípio da Substituição de Liskov (LSP) 7
4.4. Princípio da Segregação de Interface (ISP) 9
4.5. Princípio da Inversão de Dependência (DIP) 10

5. Padrões de Projeto **12**
5.1. Data Access Object 12
5.2. Strategy Pattern 13

6. Padrão MVC **13**

7. Conclusões **14**

8. Referências **15**

1. Apresentação do Trabalho

Ao longo deste trabalho será apresentada uma aplicação móvel que representa um metrônomo (maiores detalhes na seção 2), além de suas funcionalidades (seção 3) e as aplicações dos Princípios SOLID (seção 4) com códigos de exemplo em *Pseudocódigo*. Também serão abordados os padrões de projeto (Seção 5) e de arquitetura (seção 6) utilizados. Por fim, serão apresentadas algumas conclusões sobre este estudo e sobre o aproveitamento da disciplina Programação Orientada a Objetos Avançada, ministrada pelo professor Delano no ENPE 3 (2021/1), na Universidade Federal de São Carlos (Seção 7).

2. Conceitos Básicos

A seguir serão apresentados alguns conceitos básicos referentes ao que se espera de um metrônomo para a compreensão de algumas aplicações de conceitos e funcionamento do programa como um todo.

2.1. Metrônomo

Um metrônomo, de forma geral, é um instrumento que indica o andamento musical através de pulsos com intervalo regular **[1]**. Esta medida é definida pelo valor de BPM (Batidas Por Minutos) da música em questão e sua aplicação pode variar consideravelmente, sendo utilizado desde alunos e entusiastas por instrumentos musicais, a músicos profissionais em concertos. Para este trabalho será considerada uma aplicação móvel de um metrônomo voltado a estudantes, visto que apresentará funções de interesse para iniciantes em interpretação musical.

2.2. Compasso

O compasso é a maneira pela qual se divide um grupo de sons de uma composição musical, facilitando a performance de uma canção, a partir de um ritmo definido de maneira contínua **[2]**. Para fins de simplicidade, serão consideradas apenas compassos que podem ser divididos em 4 intervalos. Por exemplo, a medida 4/4 (lê-se "quatro por quatro") tem o primeiro valor "4" indicando a quantidade de batidas em cada um destas medidas e o segundo valor "4" indica a quantidade de divisões que o intervalo pode ter. Podem haver diversas medidas para um compasso, mas neste trabalho serão abordadas apenas as medidas 4/4 (que é a mais utilizada dentre as músicas populares), ou variantes próximas, como 3/4.

2.3. Batida

Uma batida é um pulso uniforme, definido pela unidade de medida BPM (Batidas Por Minutos) **[3]**. Juntamente do compasso, é um dos elementos que definem o andamento de uma canção, determinando a velocidade na qual uma música é tocada.

3. Classes e Funções

Nesta seção serão apresentadas as principais classes e funções que serão abordadas ao longo do trabalho.

3.1. Classes

Um metrônomo pode ser composto por classes que definem seu funcionamento com base na instrução musical que ele deve apresentar. Para isto, serão consideradas as seguintes classes base:

- **Compasso**: Define o ritmo apresentado, a partir de uma divisão de batidas e pausas em um determinado intervalo de tempo **[2]**. Apesar de não ser uma medida que varia muito, considerando que a maior parte das canções populares seguem uma métrica de 4/4, ou seja, quatro batidas por intervalo, a disponibilidade de medidas e contagens de ritmo diferentes podem ser de muito interesse para estudantes;
- **Batida**: Definição da quantidade de batidas por minuto a partir do valor indicado pelo usuário, utilizando funções internas do aplicativo para calcular o tempo exato de duração de cada pulso e apresentar-se de maneira estável e regular;
- **Feedback_Visual**: Elementos apresentados na tela do dispositivo e que indicam o andamento do pulso definido;
- **Feedback_Sonoro**: Elementos sonoros que indicam o andamento do pulso definido, geralmente representado por cliques.

3.2. Funções

Para que o metrônomo tenha pleno funcionamento e seja possível aproveitá-lo adequadamente, algumas funções básicas precisam ser definidas:

- **inicia**: Dá início à contagem das batidas a partir do BPM e do compasso previamente definidos;
- **para**: Para a execução do metrônomo com as medidas definidas;
- **setCompasso**: Define o compasso que será utilizado, Por convenção, o metrônomo será inicializado com o compasso 4/4;
- **setBatida**: Determina o valor de BPM que será utilizado. Por convenção, será adotado um valor inicial de 100 BPM;
- **aumentaBatida**: Aumenta em uma unidade a batida apresentada pelo metrônomo. Por convenção, será considerado um máximo de 220 BPM (a opção escolhida pode ser salva pelo usuário para que a aplicação já esteja configurada da maneira desejada em sua próxima inicialização);
- **diminuiBatida**: Diminui em uma unidade a batida apresentada pelo metrônomo. Por convenção, será considerado um mínimo de 40 BPM (a opção escolhida pode ser salva pelo usuário para que a aplicação já esteja configurada da maneira desejada em sua próxima inicialização);
- **gerenciaBatida**: Responsável por administrar o valor de BPM, aumentando ou diminuindo seu valor de acordo com os comandos do usuário;

- **feedbackSom**: Responsável por apresentar um determinado efeito sonoro a partir do valor de BPM apresentado, dispondo de sons diversos a serem escolhidos (a opção escolhida pode ser salva pelo usuário para que a aplicação já esteja configurada da maneira desejada em sua próxima inicialização);
- **piscaTela**: Principal função do feedback visual, fazendo com que a tela pisque para cada batida correspondente do metrônomo, entre outras opções disponíveis (a opção escolhida pode ser salva pelo usuário para que a aplicação já esteja configurada da maneira desejada em sua próxima inicialização).

4. Aplicação dos Princípios SOLID

Nesta seção serão discorridos os detalhes da implementação deste metrônomo com base em códigos escritos em pseudocódigo e buscando respeitar os princípios SOLID.

4.1. Princípio da Responsabilidade Única (SRP)

Considerando a definição do SRP, de que uma classe deve ter apenas um motivo para mudar **[5]**, isto pode ser enxergado na função *gerenciaBatida*, representada abaixo:

```
Função gerenciaBatida:
inícioFunção
    # "aumenta" é uma variável booleana definida no programa principal e que verifica
    # se o usuário pressionou o botão de aumentar a quantidade de batidas.
    se aumenta:
        # A função aumentaBatida() é a responsável por incrementar o valor em uma
        # unidade, sendo assim, a cada vez que o usuário pressionar o botão de
        # aumentar o valor do BPM, esta função será ativada.
        total = aumentaBatida(1, valorBPM)
        se total = 220:
            # Assim que atinge o limite máximo de BPM, a função para de
            # incrementar seu valor. Esta função retorna o valor do BPM corrente
            # para a verificação.
            total = aumentaBatida(0, valorBPM)
    # Caso em que o usuário opta por reduzir o valor do BPM.
    senão:
        # Efetua o mesmo processo que a função aumentaBatida(), com a exceção de
        # que, desta vez, reduz em uma unidade o valor do BPM.
        total = diminuiBatida(1, valorBPM)
        se total = 40:
            # Assim que atinge o limite mínimo de BPM, a função para de reduzir
            # seu valor. Esta função retorna o valor do BPM corrente para a
            # verificação.
            total = diminuiBatida(0, valorBPM)
```

```
fimFunção
```

Código 1: Representação em pseudocódigo da função gerenciaBatida.

Desta forma, percebe-se que esta classe está isolada quanto à sua função principal, que é administrar a contagem de batidas do metrônomo. Assim, a classe refere-se à responsabilidade única de aumentar ou diminuir o valor do BPM de acordo com o comando do usuário, além de que, qualquer alteração em seu código seria referente a esta responsabilidade, como a alteração dos limites mínimo e máximo do valor, ou com a possível implementação de alguma nova função, como o incremento de batidas em um valor X após um determinado intervalo de tempo.

4.2. Princípio do Aberto/Fechado (OCP)

Dado que o OCP define que as entidades de software (como classes e funções) deve ser abertas para ampliação, mas fechadas para modificação **[5]**, isto pode ser aplicado à classe *Feedback_Sonoro*, que será simplificada às suas funções básicas para fins de melhor visualização:

```
Classe Feedback_Sonoro:
inícioClasse
    # A princípio, a classe importa, dos arquivos da aplicação, o efeito sonoro que será
    # utilizado. Por padrão, este efeito é um clique.

    Função feedbackSom:
    inícioFunção

        efeito = "som clique" # "efeito" é uma string que carrega o nome do efeito a
                              # ser importado.

        som = importa(efeito)

        # Em seguida, efetua a apresentação do efeito escolhido de acordo com a
        # batida definida pelo usuário com o auxílio da classe Batida.
        ativaEfeito(batidaAtual.valorBPM, efeito)

    fimFunção

fimClasse
```

Código 2: Representação em pseudocódigo da classe Feedback_Sonoro.

Sendo assim, nota-se que a classe em questão pode ter funções variadas que implementem extensões de sua função básica, sem que isto interfira em seu código original, ocasionando em mudanças indesejadas. Para que isto ocorra, basta alterar o arquivo de importação de efeito sonoro, como apresentado no exemplo abaixo, que utilizará um efeito sonoro de palmas no lugar do efeito de clique padrão.

```
Classe Feedback_Sonoro_Palmas:
inícioClasse

    Função feedbackSom:
    inícioFunção

        efeito = "som_palmas"

        som = importa(efeito)

        ativaEfeito(batidaAtual.valorBPM, efeito)

    fimFunção

fimClasse
```

Código 3: Representação em pseudocódigo da classe Feedback_Sonoro_Palmas.

Deste modo, basta alterar uma única linha de código, no caso, o nome do arquivo cujo efeito deseja-se implementar na aplicação, para que o efeito desejado seja obtido. Isto pode ser feito diversas vezes, com efeitos sonoros diversos, como batidas de uma bateria ou teclas de um piano, como pode ser enxergado no diagrama abaixo:

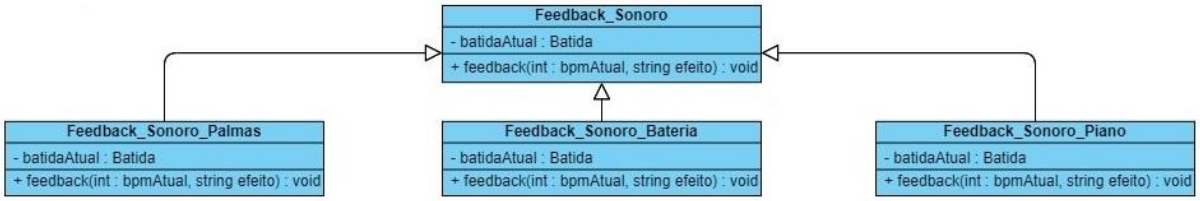

Figura 1: Diagrama representativo da classe Feedback_Sonoro e algumas de suas possíveis variáveis.

4.3. Princípio da Substituição de Liskov (LSP)

A partir da definição do LSP, de que os subtipos devem ser substituíveis pelos seus tipos de base **[5]**, é possível enxergá-lo com a classe *Feedback_Visual*:

```
Classe Feedback_Visual:
inícioClasse

    Função exibicão:
    inicioFunção
        # Será implementada com maiores detalhes pelas classes, apresentada aqui
        # apenas como uma exibição padrão em que a tela pisca a cada batida do
        # metrônomo.
        piscaTela(valorBPM, true)
    fimFunção

fimClasse
```

Código 4: Representação em pseudocódigo da classe Feedback_Visual.

Porém, é de se esperar que o usuário opte por desabilitar esta função, principalmente em casos de pessoas com fotossensibilidade, como apresentado a seguir.

```
Classe Feedback_Visual_Desativado:
inícioClasse

    Função exibicão:
    inicioFunção
        # Desta vez, a opção fica desativada graças ao valor booleano false.
        piscaTela(valorBPM, false)
    fimFunção

fimClasse
```

Código 5: Representação em pseudocódigo da classe Feedback_Visual_Desativado.

Ou, então, o usuário pode desejar que a aplicação pisque a tela apenas na primeira batida do compasso, indicando uma nova medida:

```
Classe Feedback_Visual_Prim_Batida_Compasso4:
inícioClasse

    Função exibicão:
    inicioFunção
        # Agora, a função recebe o valor de BPM dividido por 4, considerando que o
        # compasso escolhido pelo usuário é um dos que podem ser divididos em 4
        # medidas.
```

```
            piscaTela(valorBPM / 4, true)
        fimFunção

fimClasse
```

Código 6: Representação em pseudocódigo da classe Feedback_Visual_Prim_Batida_Compasso4.

Desta maneira, a estrutura principal da classe pode ser mantida, de modo que apenas a parte de interesse para que a nova funcionalidade seja cumprida precise de alguma alteração.

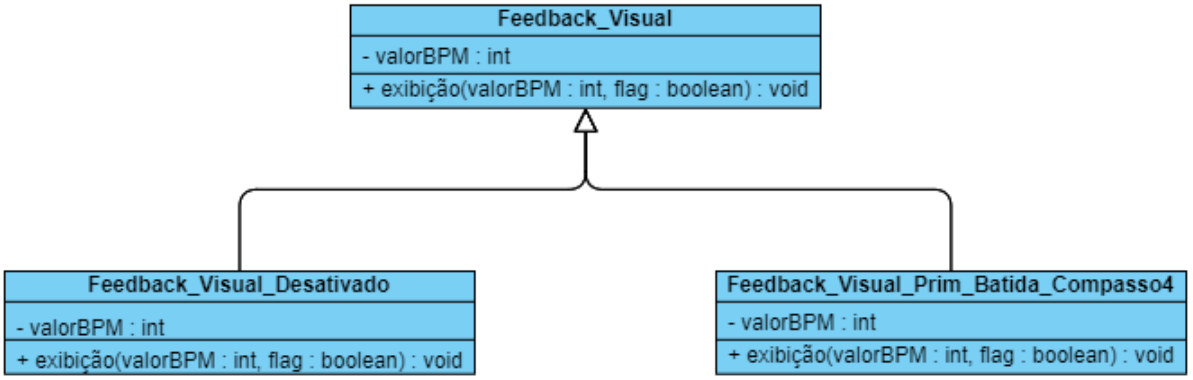

Figura 2: Diagrama representativo da classe Feedback_Visual e algumas outras funcionalidades possíveis.

4.4. Princípio da Segregação de Interface (ISP)

De acordo com o conceito do ISP, de que os clientes não devem ser obrigados a depender de métodos que não utilizam (5), isto pode ser aplicado nas funções de aumentar e diminuir o valor de BPM do metrônomo. A princípio, será apresentado um exemplo que fere o ISP:

```
# Para este exemplo, será considerada uma função que aumenta E diminui o valor de BPM,
# de acordo com um valor booleano que representa o comando dado pelo usuário.
Função alteraBPM:
inícioFunção
        if aumentaValor:
                BPM = BPM + 1
        else:
                BPM = BPM - 1

fimFunção
```

Código 7: Representação em pseudocódigo da função alteraBPM, ferindo o ISP.

Assim, percebe-se que o ISP é violado, pois a função precisa cuidar de duas responsabilidades (ferindo também o SRP), sendo que poderia ser facilmente dividida em duas funções diferentes. Deve-se considerar também que o usuário pode nunca aumentar ou diminuir o valor do BPM. Talvez ele se sinta satisfeito com o valor predefinido (100) para a aplicação, ou sua intenção seja sempre a de aumentar este valor de alguma forma. Sendo assim, é necessário que estas funcionalidades do aplicativo sejam separadas em duas funções diferentes. A seguir serão apresentadas as versões adequadas destas funcionalidades:

Função aumentaBatida:
inícioFunção

    retorna BPM + 1

fimFunção

Código 8: Representação em pseudocódigo da função aumentaBatida.

Função diminuiBatida:
inícioFunção

    retorna BPM - 1

fimFunção

Código 9: Representação em pseudocódigo da função diminuiBatida.

Desta forma, ambos códigos correspondentes às funções podem ser executados separadamente e apenas quando forem necessários.

4.5. Princípio da Inversão de Dependência (DIP)

Considerando as condições do DIP, de que módulos de alto nível não devem depender de módulos de baixo nível, mas sim de abstrações, e que as abstrações não devem depender de detalhes, e sim os detalhes que devem depender das abstrações **[5]**, podemos enxergá-lo nas definições da classe e métodos da classe *Compasso*. A princípio, esta classe poderia ser apresentada da seguinte forma:

Classe Compasso:
inícioClasse

    Função compasso1-4:
    inícioFunção
        # Aqui serão implementadas as especificidades desta função.

```
fimFunção

Função compasso2-4:
inícioFunção
        # Aqui serão implementadas as especificidades desta função.
fimFunção

Função compasso3-4:
inícioFunção
        # Aqui serão implementadas as especificidades desta função.
fimFunção

Função compasso4-4:
inícioFunção
        # Aqui serão implementadas as especificidades desta função.
fimFunção

fimClasse
```

Código 10: Representação em pseudocódigo da classe Compasso, ferindo o DIP.

Acima, foi apresentada uma versão da classe *Compasso* que contém todas as variações de um compasso que pode ser dividido em 4 medidas. Isto, além de ferir vários outros princípios, fere, principalmente, o DIP. Isto ocorre pois todas as funções são variações delas mesmas, contidas em um único elemento. Se faz necessário, portanto, transformar as derivadas do compasso 4/4 em uma interface abstrata de modo que a classe *Compasso* não dependa delas diretamente, facilitando a manutenção do código e sua visualização:

```
Classe Compasso:
inícioClasse

        # Cada caso apresentado a seguir seria o responsável por determinar o compasso
        # escolhido pelo usuário a partir de suas respectivas interfaces, que seriam
        # implementadas de acordo com o que se espera de suas funcionalidades.
        caso 1: # Compasso 1/4
                interfaceCompasso()

        caso 2: # Compasso 2/4
                interfaceCompasso()

        caso 3: # Compasso 3/4
                interfaceCompasso()
```

```
        caso 4: # Compasso 4/4
                interfaceCompasso()

fimClasse
```

Código 11: Representação em pseudocódigo da classe Compasso, utilizando uma interface auxiliar.

Assim, a aplicação de abstrações torna o código menos robusto e mais estável, dado que possíveis alterações não afetarão elementos que não dependem diretamente delas.

5. Padrões de Projeto

Nesta seção serão apresentados os padrões de projeto adotados.

5.1. Data Access Object

O padrão de Objeto de Acesso a Dada (ou, em inglês, Data Access Object - DAO) será utilizado para implementar uma interface abstrata que permite a manipulação de uma classe base sem interferir nos detalhes da classe **[4]**.

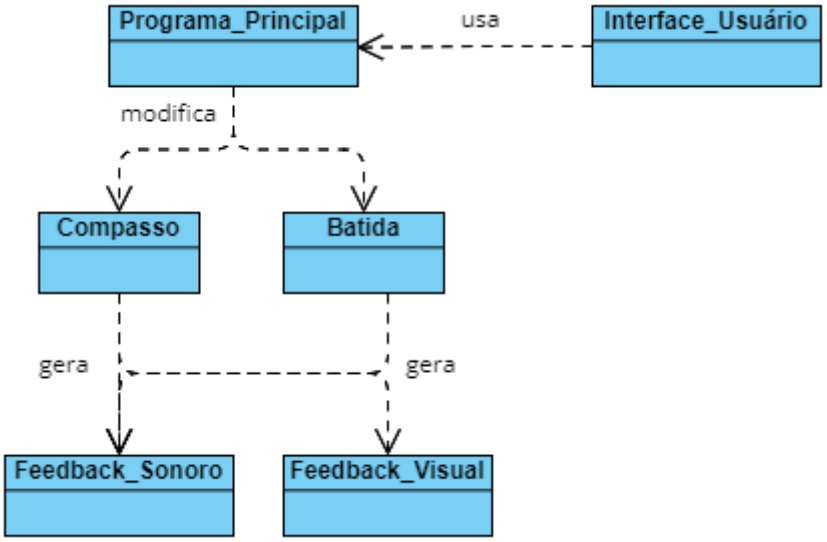

Figura 3: Diagrama representativo da aplicação de acordo com o padrão DAO.

No diagrama, percebe-se que o usuário utilizará a aplicação a partir de uma interface, que ativará o programa principal, responsável por administrar as medidas do compasso e batida, gerando, por fim, um feedback sonoro e visual de acordo com o especificado pelo usuário.

5.2. Strategy Pattern

O padrão Strategy será utilizado com a intenção de delegar as responsabilidades adquiridas pelas entidades **[6]**, ou seja, distribuir as responsabilidades de cada método e classe de acordo com suas reais funcionalidades e também permitindo que novas funcionalidades sejam implementadas de acordo com as especificações previamente apresentadas. Isto pode ser enxergado através da classe *Compasso* e suas abstrações, já descritas anteriormente:

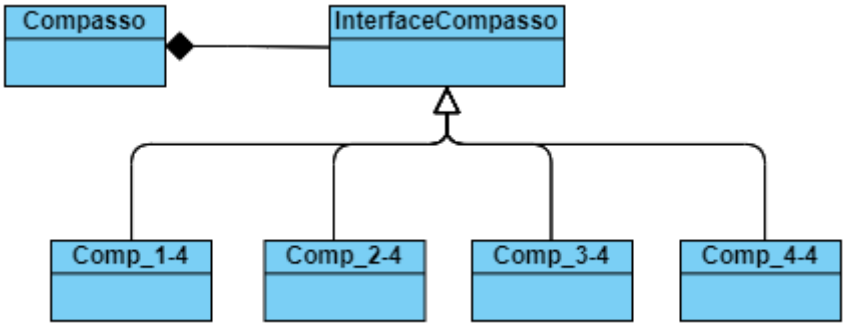

Figura 4: Diagrama representativo da classe Compasso de acordo com o padrão Strategy.

6. Padrão MVC

O padrão arquitetural MVC (Modelo-Visão-Controle, ou Model-View-Controller) tem como aplicação fundamental nesta aplicação a separação de conceitos em camadas interconectadas **[7]**, especificando a forma como o usuário interage com a aplicação e como as partes da aplicação interagem entre si.

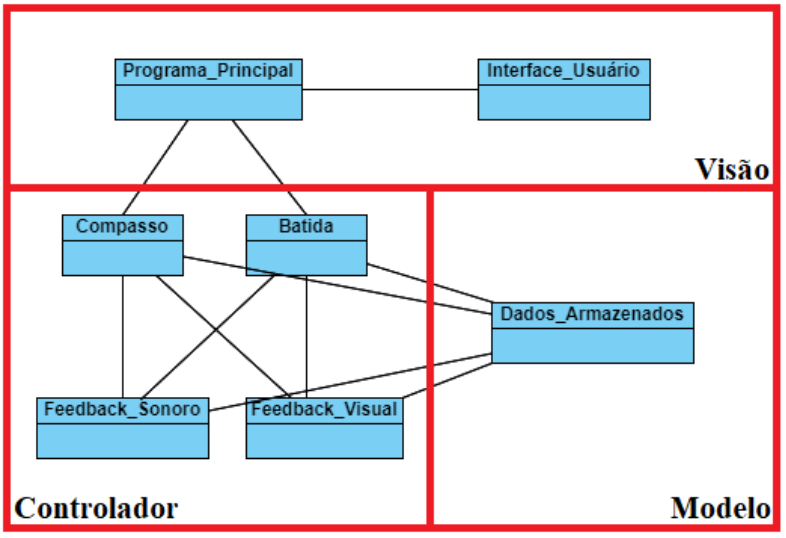

Figura 5: Diagrama representativo da aplicação do padrão MVC no projeto.

7. Conclusões

Este trabalho, portanto, exemplifica alguns dos pontos mais importantes dos princípios SOLID, essenciais para tornar os programas desenvolvidos em disciplinas futuros, e até mesmo no mercado de trabalho, mais robustos e flexíveis, de modo que a implementação de novas funcionalidades seja uma evolução para o sistema, e não uma etapa de remodelagem completa. Assim, por mais complicados e confusos que possam parecer a princípio, são essenciais no caminho de um bom programador e demandam estudos e aplicações muito além dos conhecimentos já adquiridos na disciplina.

8. Referências

[1] https://pt.wikipedia.org/wiki/Metr%C3%B4nomo, acesso em: 25/11/2021.

[2] https://pt.wikipedia.org/wiki/Compasso_(m%C3%BAsica), acesso em: 25/11/2021.

[3] https://pt.wikipedia.org/wiki/Batida_(m%C3%BAsica), acesso em: 25/11/2021.

[4] https://en.wikipedia.org/wiki/Data_access_object, acesso em: 25/11/2021.

[5] MARTIN, Roberto; MARTIN, Micah. Princípios, Padrões e Práticas Ágeis em C#, Bookman Editora, 2009.

[6] https://en.wikipedia.org/wiki/Strategy_pattern, acesso em: 25/11/2021.

[7] https://pt.wikipedia.org/wiki/MVC, acesso em: 25/11/2021.