Music Player

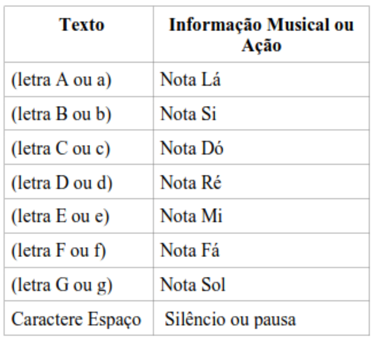
TCP - Part 2

Augusto Flach (314134), Tiago Flach (275896)

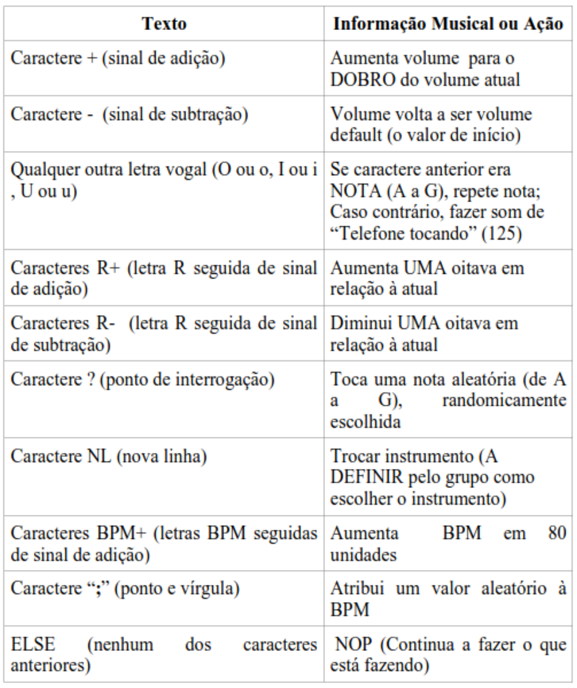
Activity 1 -
Done

Activity 2-

a) The program receives from the user a text containing a music pattern and plays it according to the following pattern:



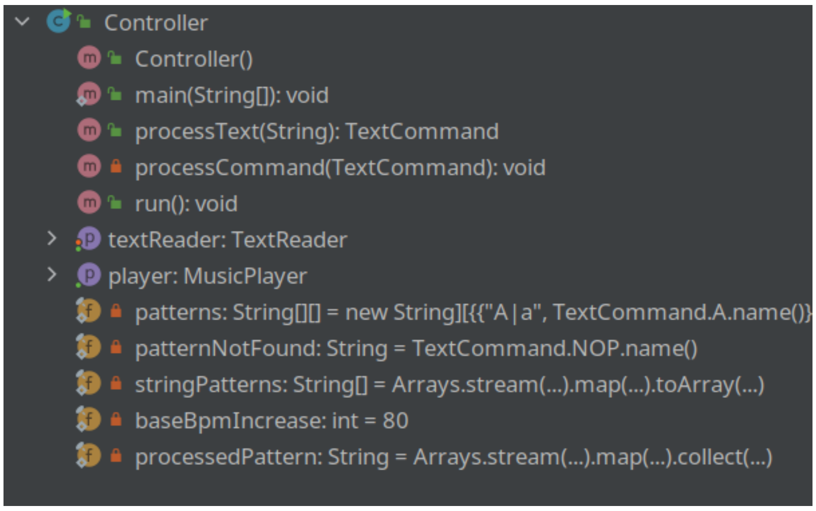
Texto	Informação Musical ou Ação
(letra A ou a)	Nota Lá
(letra B ou b)	Nota Si
(letra C ou c)	Nota Dó
(letra D ou d)	Nota Ré
(letra E ou e)	Nota Mi
(letra F ou f)	Nota Fá
(letra G ou g)	Nota Sol
Caractere Espaço	Silêncio ou pausa



Texto	Informação Musical ou Ação
Caractere + (sinal de adição)	Aumenta volume para o DOBRO do volume atual
Caractere - (sinal de subtração)	Volume volta a ser volume default (o valor de início)
Qualquer outra letra vogal (O ou o, I ou i , U ou u)	Se caractere anterior era NOTA (A a G), repete nota; Caso contrário, fazer som de "Telefone tocando" (125)
Caracteres R+ (letra R seguida de sinal de adição)	Aumenta UMA oitava em relação à atual
Caracteres R- (letra R seguida de sinal de subtração)	Diminui UMA oitava em relação à atual
Caractere ? (ponto de interrogação)	Toca uma nota aleatória (de A a G), randomicamente escolhida
Caractere NL (nova linha)	Trocar instrumento (A DEFINIR pelo grupo como escolher o instrumento)
Caracteres BPM+ (letras BPM seguidas de sinal de adição)	Aumenta BPM em 80 unidades
Caractere ";" (ponto e vírgula)	Atribui um valor aleatório à BPM
ELSE (nenhum dos caracteres anteriores)	NOP (Continua a fazer o que está fazendo)

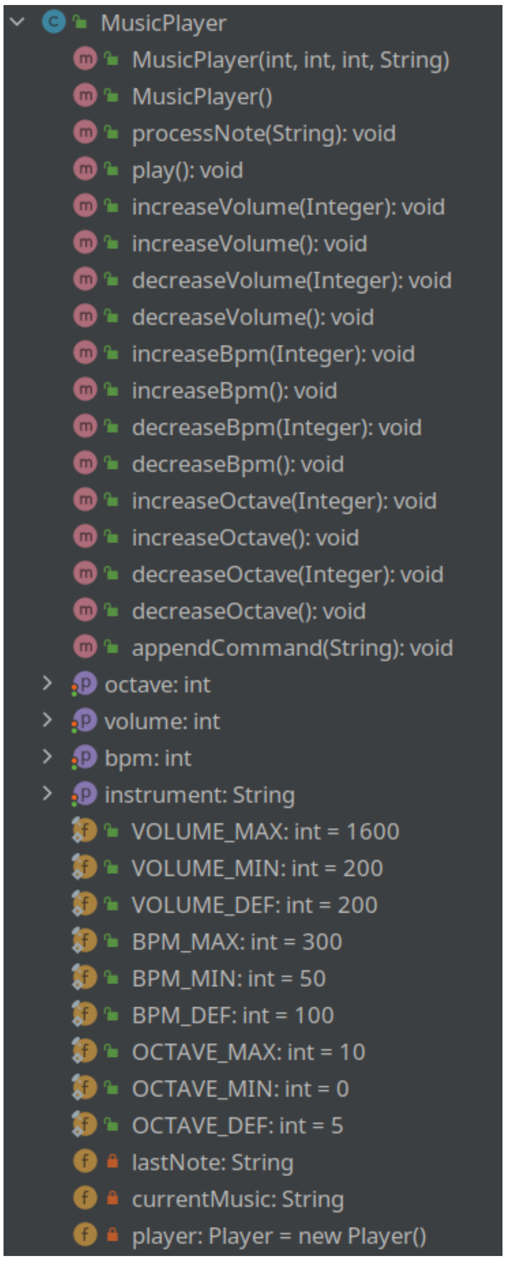
b) Até o momento, o programa possui 3 classes e um Enum:

i) Controller - It runs the main function of the program, receives the input text and manages to use both TextReader and MusicPlayer to translate and play the music. It controls various control variables such as mapping Input Patterns / Functions and processing input according to these parameters.



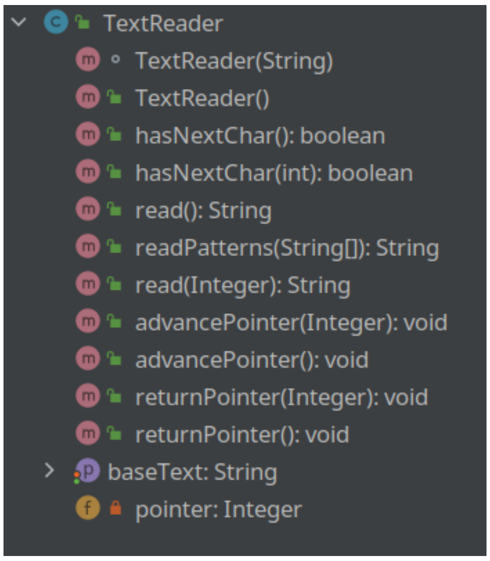
```
Controller
    Controller()
    main(String[]): void
    processText(String): TextCommand
    processCommand(TextCommand): void
    run(): void
  > textReader: TextReader
  > player: MusicPlayer
    patterns: String[][] = new String[]{{"A|a", TextCommand.A.name()}
    patternNotFound: String = TextCommand.NOP.name()
    stringPatterns: String[] = Arrays.stream(...).map(...).toArray(...)
    baseBpmIncrease: int = 80
    processedPattern: String = Arrays.stream(...).map(...).collect(...)
```

ii) MusicPlayer - Controls JFugue and music states such as octave, volume, bpm, etc.



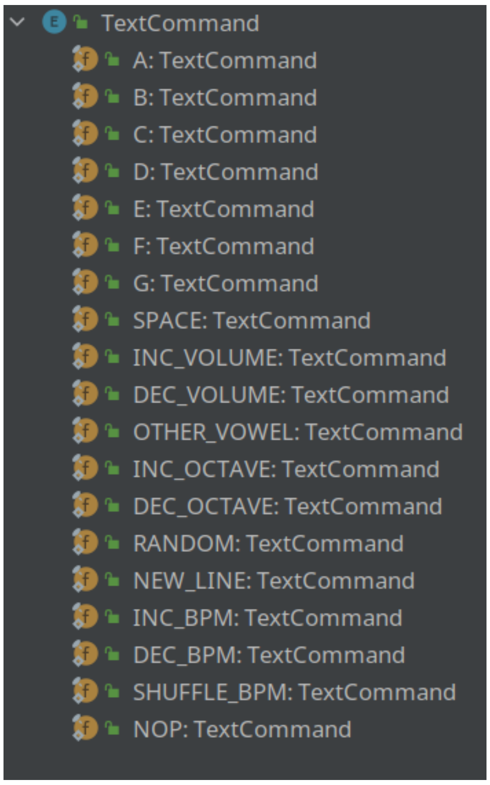
```
MusicPlayer
    MusicPlayer(int, int, int, String)
    MusicPlayer()
    processNote(String): void
    play(): void
    increaseVolume(Integer): void
    increaseVolume(): void
    decreaseVolume(Integer): void
    decreaseVolume(): void
    increaseBpm(Integer): void
    increaseBpm(): void
    decreaseBpm(Integer): void
    decreaseBpm(): void
    increaseOctave(Integer): void
    increaseOctave(): void
    decreaseOctave(Integer): void
    decreaseOctave(): void
    appendCommand(String): void
  > octave: int
  > volume: int
  > bpm: int
  > instrument: String
    VOLUME_MAX: int = 1600
    VOLUME_MIN: int = 200
    VOLUME_DEF: int = 200
    BPM_MAX: int = 300
    BPM_MIN: int = 50
    BPM_DEF: int = 100
    OCTAVE_MAX: int = 10
    OCTAVE_MIN: int = 0
    OCTAVE_DEF: int = 5
    lastNote: String
    currentMusic: String
    player: Player = new Player()
```

iii) TextReader - Controls the input text reading (through pattern tokens). So it receives a series of Regex tokens and reads the input text according to that pattern. It always goes to the biggest matched text (It matches BPM+ instead of just B).



```
TextReader
    TextReader(String)
    TextReader()
    hasNextChar(): boolean
    hasNextChar(int): boolean
    read(): String
    readPatterns(String[]): String
    read(Integer): String
    advancePointer(Integer): void
    advancePointer(): void
    returnPointer(Integer): void
    returnPointer(): void
  > baseText: String
    pointer: Integer
```

iv) TextCommand - Enum created to map regex String pattern inputs (such as “A|a”) to TextCommand elements (like TextCommand.A, in the given example).



```
TextCommand
    A: TextCommand
    B: TextCommand
    C: TextCommand
    D: TextCommand
    E: TextCommand
    F: TextCommand
    G: TextCommand
    SPACE: TextCommand
    INC_VOLUME: TextCommand
    DEC_VOLUME: TextCommand
    OTHER_VOWEL: TextCommand
    INC_OCTAVE: TextCommand
    DEC_OCTAVE: TextCommand
    RANDOM: TextCommand
    NEW_LINE: TextCommand
    INC_BPM: TextCommand
    DEC_BPM: TextCommand
    SHUFFLE_BPM: TextCommand
    NOP: TextCommand
```

Activity 3:

a) **Criteria**:

Decomposability: Mainly the MusicPlayer - TextReader - Controller structure of decomposing the problem in three main parts.

Composability: Mainly the MusicPlayer - TextReader structures, because each handles a specific task that has very little to do with the problem or solution. They handle their tasks apart from their context. The Controller class is the most linked to the solution logic, given that it controls the others with a specific logic to solve the problem.

Understandability: As each of the classes have their specific tasks and solve them separately, it should be clear by looking at each of them that the MusicPlayer handles music playing, the TextReader handles a text reading, and the Controller controls all of the others to act together to solve the given problem.

Continuity: If the specification of the problem changes, probably one of the only parts that change will be the Controller's creation of a new regex input pattern.

Protection: Currently there is no specific error handling (as the code still has not used any method that throws exceptions), but when writing the code, much of the logic was used to avoid unpredicted situations. But for security reasons, a possible TODO would be to examine object instantiation and to think more about each possible error.

b) Rules:

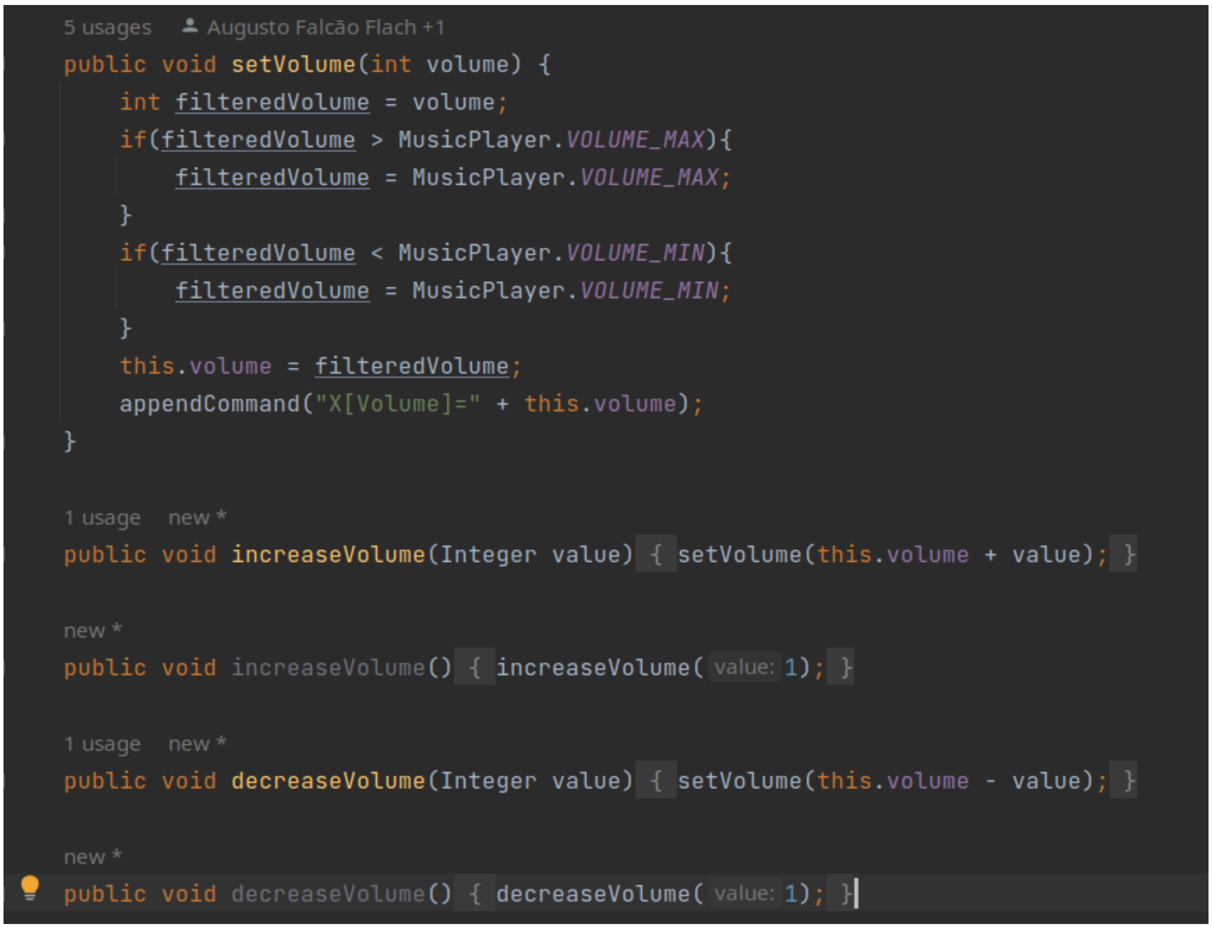
Direct Mapping: The software objective is to be a "Music generator from given text". The software has a TextReader, TextCommand, MusicPlayer and (the least directly mapped) a Controller.

Few Interfaces: The ONLY module that has ANY instance of the other three modules is the Controller. NONE of the others have any instance of anything, they just handle their own tasks.

Small interfaces (weak coupling): The only communication between Controller and TextReader is asking for another text token. Regarding Controller <-> MusicPlayer, they communicate more often (as there are many possible input commands). A possible TODO would be to pass to the MusicPlayer a list of commands, and for it to bulk-execute all commands passed by the Controller at once (with a single communication).

Explicit Interfaces: All the connections are very direct and explicitly, called by the Controller. Each call is always to another module as an ADT should.

Information Hiding: Only the GET/SET access of each module's properties is public. And even in some cases, they are not public at all (if that's a property that shouldn't be accessed, such as some Controller's internal state variables). A code that exemplifies it's access control well should be this one (even though volume acces could be done through getVolume()):



```
5 usages   Augusto Falcão Flach +1
public void setVolume(int volume) {
    int filteredVolume = volume;
    if(filteredVolume > MusicPlayer.VOLUME_MAX){
        filteredVolume = MusicPlayer.VOLUME_MAX;
    }
    if(filteredVolume < MusicPlayer.VOLUME_MIN){
        filteredVolume = MusicPlayer.VOLUME_MIN;
    }
    this.volume = filteredVolume;
    appendCommand("X[Volume]=" + this.volume);
}

1 usage   new *
public void increaseVolume(Integer value) { setVolume(this.volume + value); }

new *
public void increaseVolume() { increaseVolume( value: 1); }

1 usage   new *
public void decreaseVolume(Integer value) { setVolume(this.volume - value); }

new *
public void decreaseVolume() { decreaseVolume( value: 1); }
```

c) **Principles:**

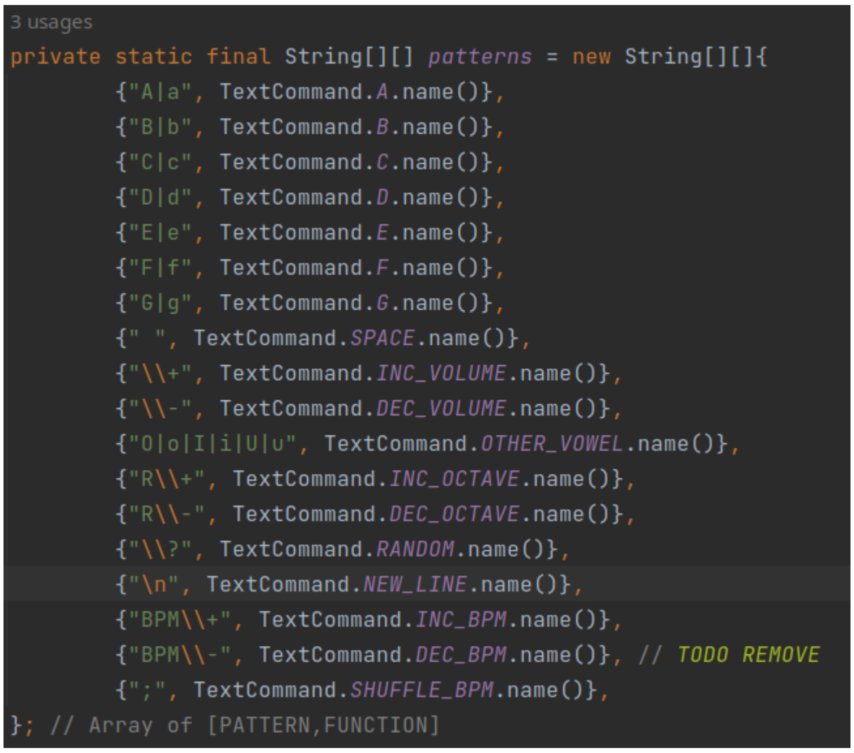
The Linguistic Modular Units principle: Probably OO is the best paradigm to model this problem solution according to that principle. In this case, all modules correspond and communicate as Classes or Enums (no Interface at the moment).

The Self-Documentation principle: Everything self-contained, as an ADT should be (at least we tried to do that). We also tried as much as possible (such as presented in Clean Code) to make our code and variables "readable", this way avoiding ambiguous and unuseful comments.

The Uniform Access principle: That one is harder to address for us, as it depends more on the language used. As we are using Java (and OO) we believe that this is well addressed, even though we are not currently using any specific interface.

The Open-Closed principle: All the three modules except Controller are Open-Closed (both ready to be used, and to extension). The only one that isn't usable by others is the Controller. It's expandable, but it's use is only related to the problem/solution logic control, and not as an ADT (such as the others).

The Single Choice principle: Here it goes the set of choices of this problem (private and static inside the Controller, as it should be):



```
3 usages
private static final String[][] patterns = new String[][]{
        {"A|a", TextCommand.A.name()},
        {"B|b", TextCommand.B.name()},
        {"C|c", TextCommand.C.name()},
        {"D|d", TextCommand.D.name()},
        {"E|e", TextCommand.E.name()},
        {"F|f", TextCommand.F.name()},
        {"G|g", TextCommand.G.name()},
        {" ", TextCommand.SPACE.name()},
        {"\\+", TextCommand.INC_VOLUME.name()},
        {"\\-", TextCommand.DEC_VOLUME.name()},
        {"O|o|I|i|U|u", TextCommand.OTHER_VOWEL.name()},
        {"R\\+", TextCommand.INC_OCTAVE.name()},
        {"R\\-", TextCommand.DEC_OCTAVE.name()},
        {"\\?", TextCommand.RANDOM.name()},
        {"\n", TextCommand.NEW_LINE.name()},
        {"BPM\\+", TextCommand.INC_BPM.name()},
        {"BPM\\-", TextCommand.DEC_BPM.name()}, // TODO REMOVE
        {";", TextCommand.SHUFFLE_BPM.name()},
}; // Array of [PATTERN,FUNCTION]
```

Activity 4)

There are two main parts that we see that could be refactored, and both were marked as TODO in the above section.

The first is regarding Small Coupling:
"A possible TODO would be to pass to the MusicPlayer a list of commands, and for it to bulk-execute all commands passed by the Controller at once (with a single communication)."
Doing it like that would obviously lower the number of communications between both modules, but as we are using it as an ADT we don't believe it would be so beneficial. It would be like bulk-passing push and pull commands to a Stack for it to execute everything at once. Doesn't make much sense, right?

The second one would be regarding protection:
"But for security reasons, a possible TODO would be to examine object instantiation and to think more about each possible error."
I think this would be very beneficial, but it's a tradeoff because of the time cost. The perfect case scenario would be to find a balance between testing the most explicit parts where bugs might happen and not taking too long searching for those parts.