Object-Oriented Programming
Group Assignment #3
My Little Mozart

CMP_SC/INFO_TC 3330

Spring 2025

1 Objective

The objective of this assignment is to implement a MIDI composition program using Java that applies both the Abstract Factory and Strategy design patterns. You will implement:

1. **Abstract Factory Pattern:** To create different types of MIDI event factories (*Standard*, *Legato*, and *Staccato*).
2. **Pitch Strategy:** A strategy for selecting the pitch and duration of notes.
3. **Instrument Strategy:** A strategy for selecting an instrument based on the MIDI channel.

2 Description

You will develop a MIDI composition system in Java that uses:

- An *Abstract Factory* to create different styles of MIDI event factories (Standard, Legato, Staccato).
- A *Note Strategy Pattern* to dynamically control the way notes are generated.
- An *Instrument Strategy Pattern* to assign instruments to channels dynamically.
- CSV file parsing to read MIDI event data and generate the sequence.

3 CSV File Format

The program should read a CSV file with the following format:

`startEndTick, Note_on_off, channel, note, velocity, instrument`

Example CSV content:

```
0, Note_on_c, 0, 43, 100, 34
576, Note_off_c, 0, 43, 0, 34
576, Note_on_c, 0, 43, 100, 34
```

4 Implementation Details

4.1 CSV Parser

You must implement a CSV parser to read the file and return a *List¡MidiEventData¿* containing the parsed events. The *MidiEventData* class should have the following structure:

```
public class MidiEventData {
        private int startEndTick, velocity, note, channel, noteOnOff;
        private int instrument;

        public MidiEventData(int startEndTick, int velocity, int note,
                    int channel, int instrument, int noteOnOff) {
                this.startEndTick = startEndTick;
                this.velocity = velocity;
                this.note = note;
                this.channel = channel;
                this.instrument = instrument;
                this.noteOnOff = noteOnOff;
        }

    // implement the getter/setters
}
```

The CSV parser (*MidiCsvParser*) should read each line, split the values by commas, and create *MidiEventData* objects.

4.2 Abstract Factory Pattern

Define an interface for creating MIDI event factories:

```
public interface MidiEventFactory {
    MidiEvent createNoteOn(int tick, int note, int velocity, int channel) throws InvalidMidiDataException;
    MidiEvent createNoteOff(int tick, int note, int channel) throws InvalidMidiDataException;
}

public interface MidiEventFactoryAbstract {
        MidiEventFactory createFactory();
}

public class StandardMidiEventFactory implements MidiEventFactory { ... }
public class LegatoMidiEventFactory implements MidiEventFactory { ... }
public class StaccatoMidiEventFactory implements MidiEventFactory { ... }
```

Use an abstract factory to select the factory type dynamically. The code below should also give you an idea of what classes are required for the factory design pattern.

```
MidiEventFactoryAbstract factoryAbstract = null;
factoryAbstract = new LegatoMidiEventFactoryAbstract();
MidiEventFactory factory = factoryAbstract.createFactory();
```

4.3 Legato and Staccato Behavior

The LegatoMidiEventFactory and *StaccatoMidiEventFactory* should implement different playing styles with specific arithmetic changes to the note durations:

4.3.1 Legato

Smooth and connected notes. The *LegatoMidiEventFactory* should create note events with longer durations and minimal gaps between the *NoteOff* and the next *NoteOn* event.

- **Arithmetic**: Increase the duration by 80 ticks of the original length.

4.3.2 Staccato

Short and detached notes. The *StaccatoMidiEventFactory* should create note events with shorter durations, introducing a gap between the *NoteOff* and the next *NoteOn* event.

- **Arithmetic**: Decrease the duration by 120 of the original length.

4.4 Pitch Strategy

Define an interface for pitch modification:

```
interface PitchStrategy {
    int modifyPitch(int note);
}
```

Implement at least two strategies:

- **HigherPitchStrategy:** Raises the pitch by 2 semitones (add 2 to the note).
- **LowerPitchStrategy:** Lowers the pitch by 2 semitones (subtract 2 from the note).

4.5 Instrument Strategy

Define an interface for instrument selection:

```
interface InstrumentStrategy {
    void applyInstrument(Track track, int channel);
}
```

Implement at least three strategies. You can create multiple instrument strategies if you want to have fun:

- **ElectricBassGuitarStrategy:** Assigns an electrical bass guitar (finger) (MIDI instrument 33).
- **TrumpetStrategy:** Assigns a trumpet (MIDI instrument 56).
- **AcousticGrandPianoStrategy:** Assigns an acoustic grand piano (MIDI instrument 0).

4.6 Usage in Main

Your *Main* should dynamically select and apply a pitch strategy, instrument strategy, and MIDI event factory. Example usage:

```
public class Main {
    public static void main(String[] args) {
        try {
            List<MidiEventData> midiEvents = MidiCsvParser.parseCsv("./files/mystery_song.csv");
            Sequence sequence = new Sequence(Sequence.PPQ, 384);
            Track track = sequence.createTrack();

            MidiEventFactoryAbstract factoryAbstract = new StandardMidiEventFactoryAbstract();
            // MidiEventFactoryAbstract factoryAbstract = new LegatoMidiEventFactoryAbstract();
            // MidiEventFactoryAbstract factoryAbstract = new StaccatoMidiEventFactoryAbstract();

            MidiEventFactory factory = factoryAbstract.createFactory();

            // Choose an instrument strategy (e.g., Trumpet, Bass Guitar, Piano)
            InstrumentStrategy instrumentStrategy = new ElectricBassGuitarStrategy();
            instrumentStrategy.applyInstrument(track, 0);
            instrumentStrategy = new TrumpetStrategy();
            instrumentStrategy.applyInstrument(track, 1);

            // Choose a pitch strategy (e.g., HigherPitch, LowerPitch)

            PitchStrategy pitchStrategy = new HigherPitchStrategy();

            for (MidiEventData event : midiEvents) {
                int modifiedNote = pitchStrategy.modifyPitch(event.getNote());
                // call this as much as you want if you want to get a higher pitch
                modifiedNote = pitchStrategy.modifyPitch(modifiedNote);

                if(event.getNoteOnOff() == ShortMessage.NOTE_ON) {
                    track.add(factory.createNoteOn(event.getStartEndTick(),
                            modifiedNote,
                            event.getVelocity(),
                            event.getChannel()));
                }
                else {
                    track.add(factory.createNoteOff(event.getStartEndTick(), modifiedNote, event.getChannel()));
                }
            }

            // Playing the sequence
            Sequencer sequencer = MidiSystem.getSequencer();
            sequencer.open();
            sequencer.setSequence(sequence);
            sequencer.start();

            while (sequencer.isRunning() | sequencer.isOpen()) {
                Thread.sleep(100);
            }
            Thread.sleep(500);
            sequencer.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5 Hints on Using the *javax.sound.midi* Library

To help you get started with the *javax.sound.midi* library, here are some important hints:

Creating a Sequence and Track:

- To create a MIDI sequence, use the *Sequence* class with the specified resolution.
- Add tracks to the sequence using the *createTrack()* method.

```
// may change based on the song (the CSV uses this)
Sequence sequence = new Sequence(Sequence.PPQ, 384);
Track track = sequence.createTrack();
```

Adding MIDI Events:

- Use the *ShortMessage* class to create MIDI messages such as *NOTE_ON*, *NOTE_OFF*, and instrument changes.
- *MidiEvent* wraps the message with a timestamp (tick).

Playing the Sequence:

- Use *Sequencer* to play the MIDI sequence.
- Make sure to *open()* the sequencer before starting it.

```
Sequencer sequencer = MidiSystem.getSequencer();
sequencer.open();
sequencer.setSequence(sequence);
sequencer.start();
```

Important Notes

- Follow Java naming conventions, or you will lose points.
- Use packages or you will lose points.
- Add Javadoc to your code, or you will lose points.
- Export your project properly, or you will lose points.
- Don't want to 1 commit project, or commit messages like "*Adding Java code*" or "*Update code*", otherwise you will lose points. Commits must be small and meaningful with a commit message that is relevant to the code you pushed. Only I am allowed to do the above, because I am the professor of this class, and I can do whatever I want. This is my class :D
- Write your code considering edge cases. Make sure you have error controls.
- Don't ask how many points will be deducted for the notes above. There is no negotiation here. These are good practices that you must adopt and follow to have a successful career. You can try to violate one of the good practices above and see what happens :) (not recommended).
- Everyone in the group must contribute to the project. Use Git efficiently and communicate!
- If there is a group drama, you have to wait until the next group assignment to split from your group or work alone. See syllabus for details.
- **Due date:** 4/4/2025, 11:59 PM.
- **Submission:** You must submit your GitHub repository and your exported project through Canvas. Submit your GitHub link repository in a file along with your project file submission.

6 Grading Rubric

Component	Points	Description
CSV Parsing	20	Correctly reads and parses the CSV file into 'MidiEventData' objects.

Abstract Factory Pattern	20	Proper implementation of the factory pattern with Standard, Legato, and Staccato factories.
Pitch Strategy Pattern	15	Correct implementation of pitch modification strategies (higher and lower pitch).
Instrument Strategy Pattern	15	Correct application of instruments based on the channel.
MIDI Event Generation	15	Creates valid MIDI events and sequence, plays the song correctly.
Code Quality	15	Clean, efficient, and well-documented (Javadoc) and commented code.
BONUS	3	If you guess/find what the *mystery song* (the CSV file) is correctly :)
Total	**103**	**Overall score**

7 Change log

- (3/28/2025) Missing column "*velocity*" is added in Section 3. There were five columns in the text, but the sample had six columns.
- (3/28/2025) In Section 4.1, the "*MidiEventData*" class has been updated. Redundant field "*endTick*" is removed, and "*noteOnOff*" field added. The constructor is also updated accordingly.