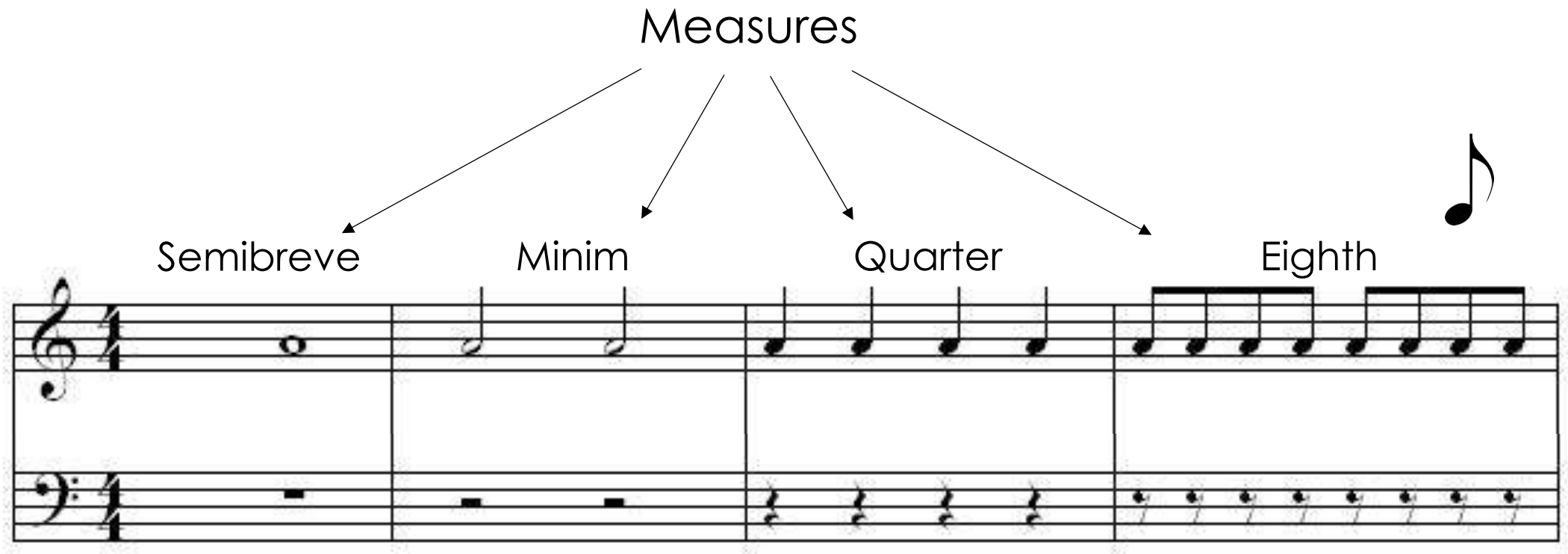
Federico
Stella

A.I.DA

Java-based evolutionary generation of monophonic melodies with limited domain knowledge, customizable scales and fitness assigned by a human

What's music?



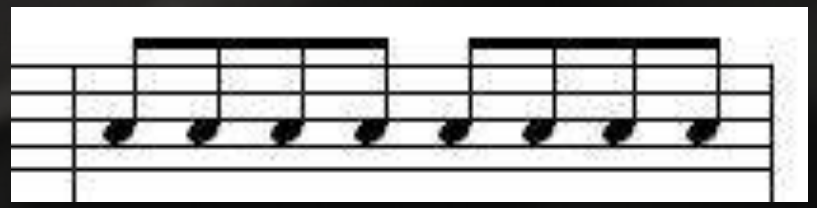
Digital music: simplified MIDI

MidiEvent
- Message (i.e. Note ON, Note OFF, etc.; channel)
- Tick (time instant of event onset)

```
sequence = new Sequence(Sequence.PPQ, 2);
sequencer.setTempoInBPM(120);
```

Beats per minute

Pulses per beat (quarter)



2 pulses for each quarter note
=> max resolution = eighth note

Genotype

MidiEvents are unnecessarily cumbersome, and notes keep playing unless a NoteOff event is sent for *that specific note*.
=> Simplifying the interface with ad-hoc TickEvents

Chromosome

TickEvent	TickEvent	TickEvent	TickEvent	TickEvent	TickEvent	TickEvent	TickEvent

NoteOffevent -> Stops the previous note

NoteOnEvent -> Stops the previous note and starts playing a new one
Contains a **number** that will be mapped to a note

HoldEvent -> Holds the last event (longer pause or longer note)

Phenotype mapping

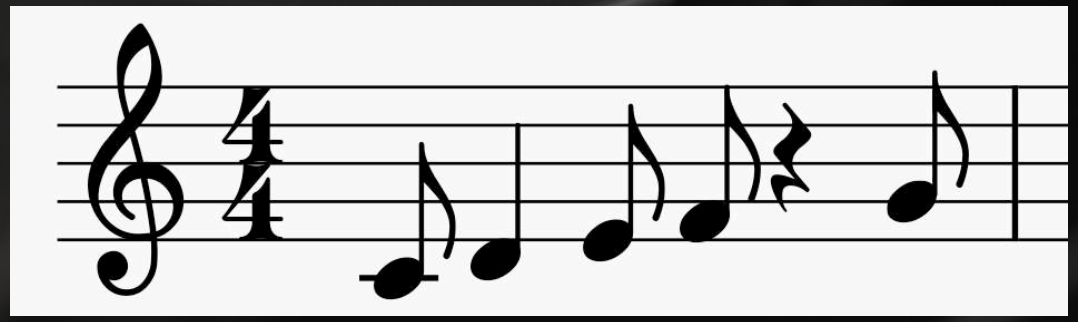
Chromosome

Note_0	Note_1	Hold	Note_2	Note_3	OFF	Hold	Note_4

Scale (example: C Major)

C4	D4	Hold	E4	F4	OFF	Hold	G4

Eighth | Quarter | Eighth | Eighth | Quarter rest | Eighth



Genetic algorithm

```
tournamentSize = 4
mutationProbability = 0.2
noteRange = 6 to 10

public List<Measure> produceNewIndividuals(List<Measure> population, int num) {
        List<Measure> result = new ArrayList<>();
        for (int i=0; i<num; i++) {
                List<Measure> candidates = RandomGenerator.getGenerator()
                        .randomFromPopulation(population, tournamentSize); //Uniformly random
                Measure parent1 = new Tournament(candidates).getWinner(); //Highest fitness
                candidates = RandomGenerator.getGenerator()
                        .randomFromPopulation(population, tournamentSize);
                Measure parent2 = new Tournament(candidates).getWinner();
                Measure child = SimpleCrossOver.getInstance().apply(parent1, parent2);
                Measure childMutated = SimpleMutation.getInstance()
                        .mutate(child, mutationProbability, noteRange);
                result.add(childMutated);
        }
        return result;
}
```

Crossover

Random selection of the split point
Generation of a single child
Parents aren't kept

Parent 1	Note_0	Note_3	Hold	Note_1	OFF	Note_6	Note_4	Note_4
Parent 2	OFF	Note_1	Note_2	OFF	Note_0	Hold	Hold	Note_7
Child								

Mutation

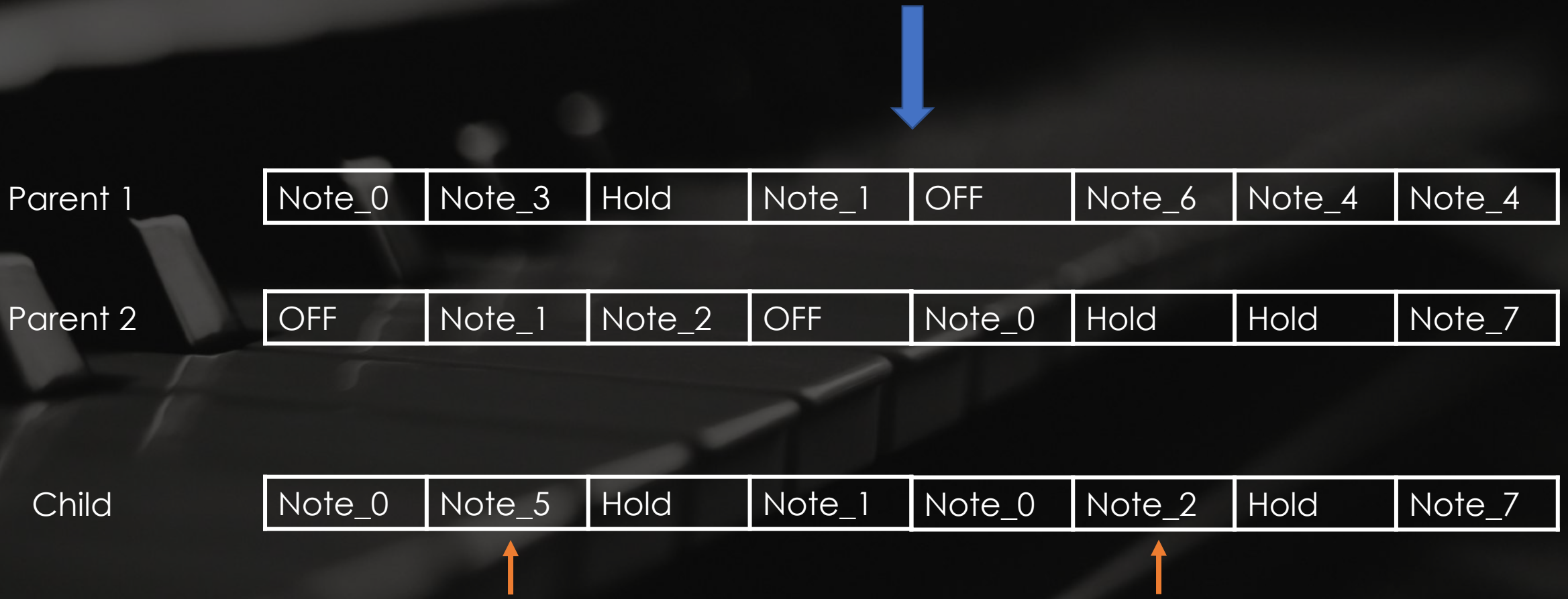
Block-based
Probability of replacement of each TickEvent

Crossover

Random selection of the split point
Generation of a single child
Parents aren't kept

Mutation

Block-based
Probability of replacement of each TickEvent



Practical limitations

With `noteRange = 8` and `measureLength = 16`, the state space contains:

```
geneOptions = noteRange + OFF + HOLD = 10
numGenes = measureLength = 16
```

State space dimension $\cong 16^{10} = 2^{4^{10}} = 2^{40} \cong \mathbf{1}\ \boldsymbol{trillion}$

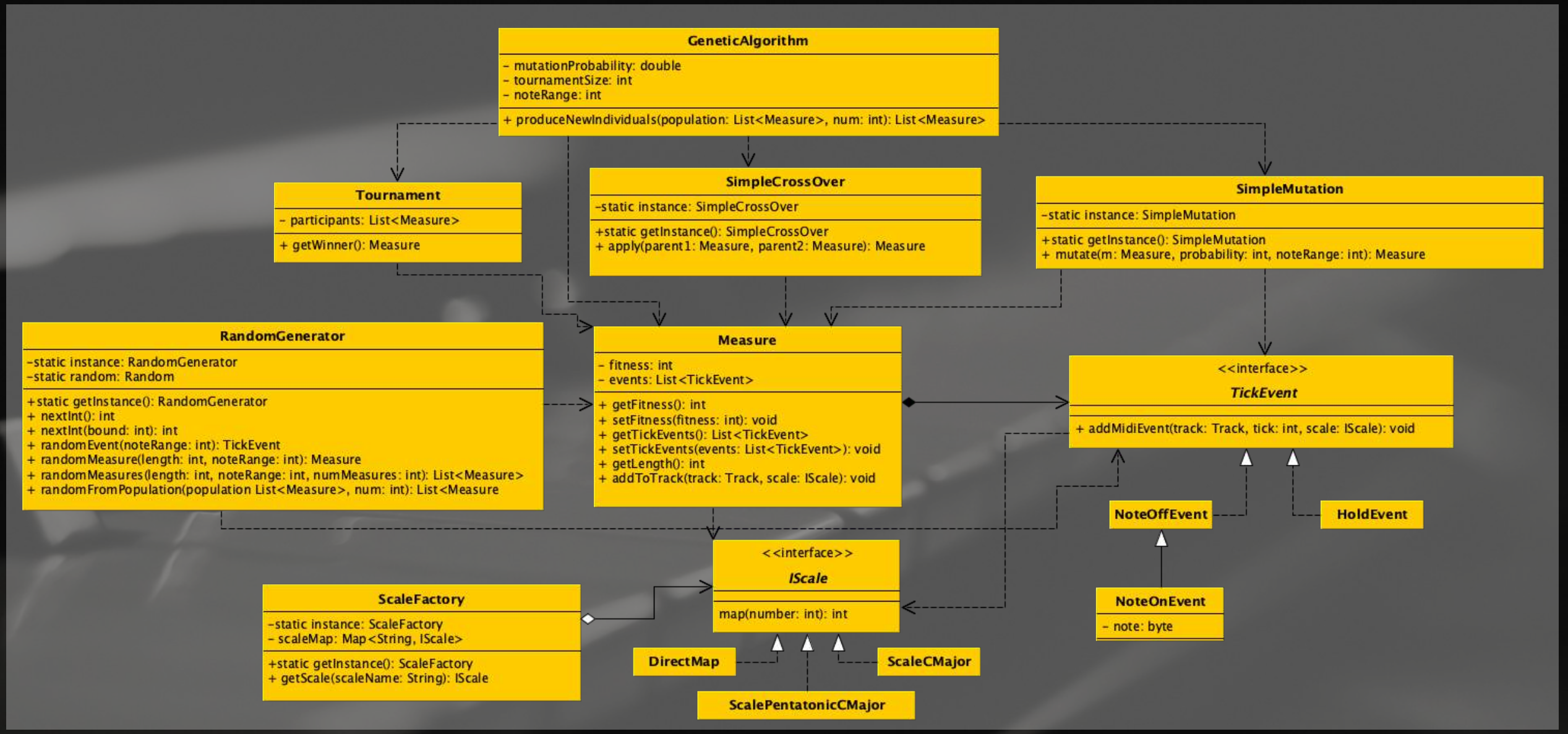
Fitness is assigned by the user

Playing a measure requires $16 * \frac{60\ seconds}{120\ PPM * 2\ PPQ} = 4\ seconds$

⇒ Population size is limited by the human factor, limiting the diversity of the produced music

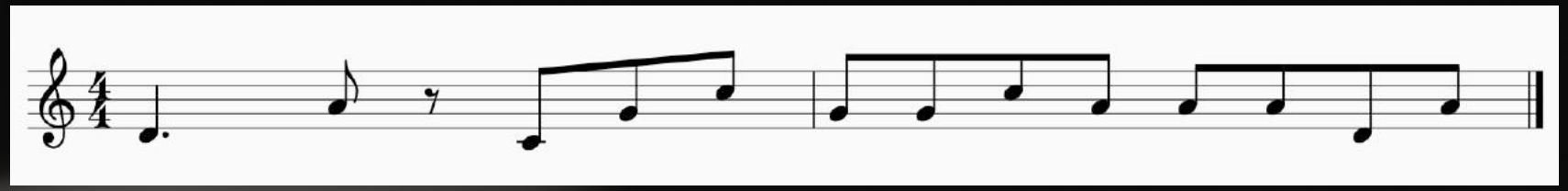
⇒ In order to avoid a quick convergence to very bad local maxima, a **certain amount of measures is randomly generated** from scratch in each new generation, refreshing the population

UML


GeneticAlgorithm
- mutationProbability: double
- tournamentSize: int
- noteRange: int
+ produceNewIndividuals(population: List<Measure>, num: int): List<Measure>
Tournament
- participants: List<Measure>
+ getWinner(): Measure
SimpleCrossOver
-static instance: SimpleCrossOver
+static getInstance(): SimpleCrossOver
+ apply(parent1: Measure, parent2: Measure): Measure
SimpleMutation
-static instance: SimpleMutation
+static getInstance(): SimpleMutation
+ mutate(m: Measure, probability: int, noteRange: int): Measure
RandomGenerator
-static instance: RandomGenerator
-static random: Random
+static getInstance(): RandomGenerator
+ nextInt(): int
+ nextInt(bound: int): int
+ randomEvent(noteRange: int): TickEvent
+ randomMeasure(length: int, noteRange: int): Measure
+ randomMeasures(length: int, noteRange: int, numMeasures: int): List<Measure>
+ randomFromPopulation(population List<Measure>, num: int): List<Measure
Measure
- fitness: int
- events: List<TickEvent>
+ getFitness(): int
+ setFitness(fitness: int): void
+ getTickEvents(): List<TickEvent>
+ setTickEvents(events: List<TickEvent>): void
+ getLength(): int
+ addToTrack(track: Track, scale: IScale): void
<<interface>>
TickEvent
+ addMidiEvent(track: Track, tick: int, scale: IScale): void
NoteOffEvent
HoldEvent
NoteOnEvent
- note: byte
<<interface>>
IScale
map(number: int): int
ScaleFactory
-static instance: ScaleFactory
- scaleMap: Map<String, IScale>
+static getInstance(): ScaleFactory
+ getScale(scaleName: String): IScale
DirectMap
ScaleCMajor
ScalePentatonicCMajor

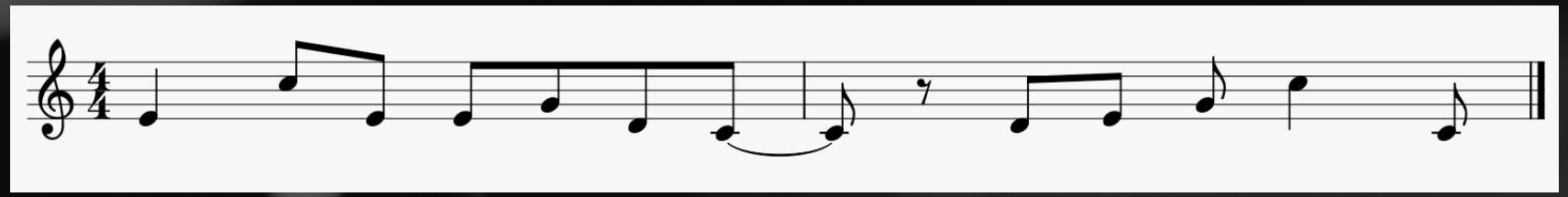
Results (Pentatonic C Major)

Gen 1



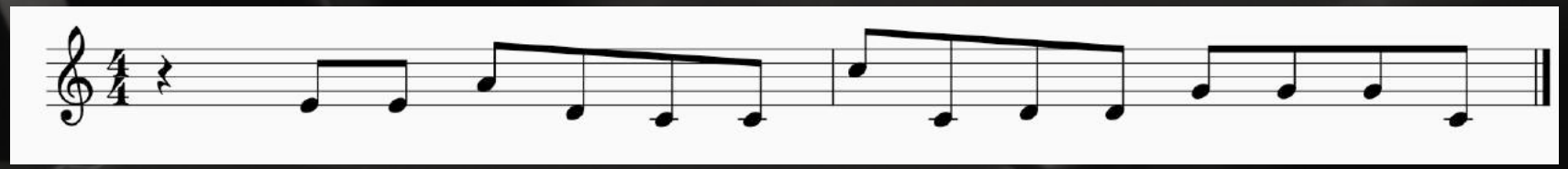
Random

Gen 5



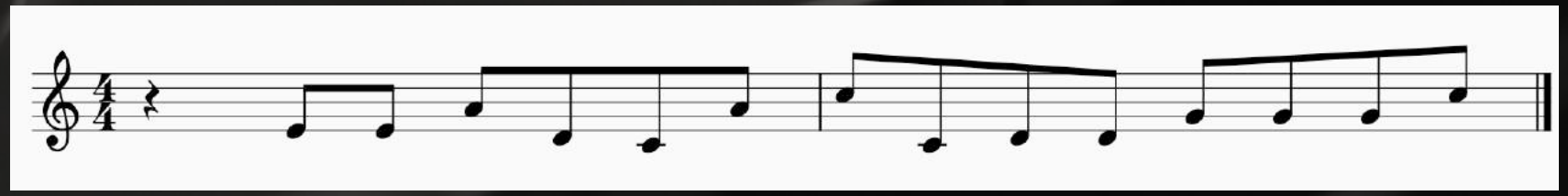
Learned low-pitched conclusion

Gen 7



Better note sequences

Gen 8



Learned high-pitched conclusion

References

- Biles, John. (1994). GenJam: A Genetic Algorithm for Generating Jazz Solos.
- Biles, John. (2013). Straight-Ahead Jazz with GenJam: A Quick Demonstration.

Thank you!