Large-Scale Example

Brian Willkie

This supplemental website extra provides a larger example of applying OOP concepts to a score based composition than we had room to discuss in chapter 18. It makes use of the NRT_TimeFrame developed in the supplemental website extra, “HowTo Pass Arguments to a Stored Function.”

1.1 Material

Below is an excerpt from a String Quartet by the author. While it might not serve as an example of electronic music, a transcription of it does offer an opportunity to discuss ways of representing musical relationships with OOP.

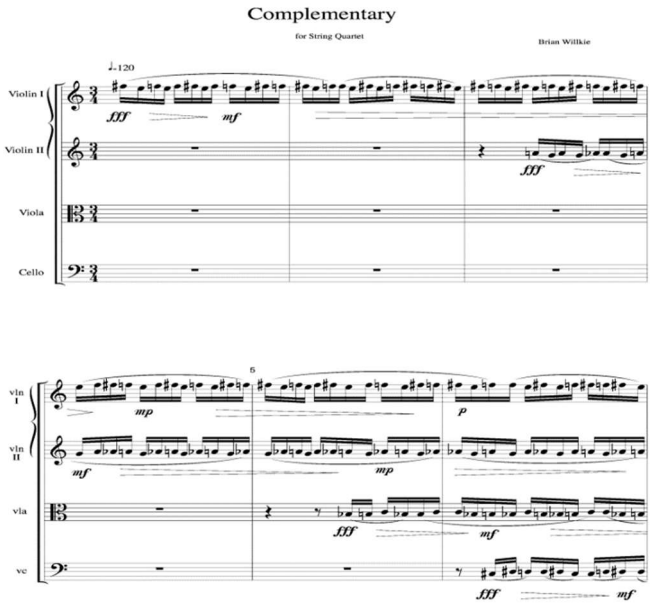

Figure 18.36

This section illustrates continuity on several levels: step-wise motion, narrow range (each voice uses only three pitches, each pitch only a half-step away from the next), a continuous stream of sixteenth notes, legato phrase markings, and smooth dynamic changes. Additionally, the dearth of any audible pattern of repetition adds to the sense of continuity by subverting any notion of ending. Despite all of this, there are hints of discontinuity, particularly the repeated notes and the abrupt entrance of each voice in *stretto*.

Already we have several things to represent. First we need some way to represent the three pitches, their various permutations and transpositions that form the pitch material of each part. Secondly, we need an instrument that allows us to articulate the beginning and ending of a phrase, but can also change pitch without re-articulating every note. In addition to the articulations, we’ll need a way to apply large-scale dynamic changes and to stagger the entrances.

Beyond modeling the relationships presented in the score, we also need to consider what information we want to provide manually, and what information we want the computer to infer. For example, we could manually specify the number of notes in the sequence and let the computer pick the pitches, or we could specify the pitches and ask the computer to count them for us. Whatever approach we choose, unless we specify every minute detail, we will have to provide at least two stages, data entry, and data processing. For this example, since we have already chosen the pitches, we'll transcribe them and let the computer figure out the timing. We will need a third stage as well to build a score from the processed data.

1.2 Data Entry

Figure 18.37 shows a very simple class, Complementary_Grundgestalt that helps manage the data we want to enter.

```
Complementary_Grundgestalt {

	var <a, <b, <c, <perm;

	*new {arg tonic = 76;
		^super.new.initGrundgestalt(tonic);
	}

	initGrundgestalt {arg tonic;
		a = PitchClass.new(tonic);
		b = PitchClass.new(a.keynum + 1);
		c = PitchClass.new(b.keynum + 1);

		perm = Dictionary.new;

		perm.put(\012, PitchCollection.new([a, b, c]));
		perm.put(\120, PitchCollection.new([b, c, a]));
		perm.put(\201, PitchCollection.new([c, a, b]));

		perm.put(\210, PitchCollection.new([c, b, a]));
		perm.put(\102, PitchCollection.new([b, a, c]));
		perm.put(\021, PitchCollection.new([a, c, b]));
	}

	at {arg key;
		^perm.at(key);
	}

	arrayAt {arg key;
		^this.at(key).pitchCollection;
	}

}
```

Figure 18.37

It encapsulates three instances of PitchClass[i] (variables a, b, and c), each one a half-step apart from the previous one, and a Dictionary of possible permutations (perm). The Dictionary key chosen for each permutation represents the order of the notes (\012 -> [a, b, c], \210 -> [c, b, a]). Beyond that, we simply provide a couple of access methods (at and arrayAt) for convenience. We can use this class to string together a list of PitchClass objects by specifying only the order.

```
	o = Complementary_Grundgestalt.new;
	[o.arrayAt(\201), o.arrayAt(\012), o.arrayAt(\021), o.arrayAt(\021)].flat;
```

Figure 18.38

Thus, once we have several measures' worth of notes, if we decide to change the reference pitch, we can do so easily without having to adjust them all manually. Likewise, we can easily build a relationship between the transposition of one voice and that of another.

```
o = Complementary_Grundgestalt.new;
p = Complementary_Grundgestalt.new(o.a.keynum - 9);
```

Figure 18.39

1.3 Sound Source

Now that we have a way to build a list of PitchClass objects, we'll need a way to parse that list and to generate the notes for our score. However we may want to consider how we want to generate sound, since that will tell us what parameters we'll need to provide as we build our score.

```
Complementary_Instruments {

    classvar <protoNotes;

    *initClass {
        StartUp.add({
            protoNotes = CtkProtoNotes.new(
                SynthDef.new(\NRT_shaper,{
                    arg outbus = 0, inbus = 0, bufnum = 0,
                    amp = 1, offSet = 0;
                    Out.ar(outbus,
                        Shaper.ar(bufnum,
                            In.ar(inbus, 1), amp, offSet));
                    }
                ),
                SynthDef.new(\NRT_sinosc,{
                    arg outbus = 0, freq = 622.542, phase = 0,
                    amp = 0, offSet = 0;
                    Out.ar(outbus,
                        SinOsc.ar(freq, phase, amp, offSet));
                    }
                )
            )
        });
    }
}
```

Figure 18.40

Figure 18.40 is an example of a singleton class. Singletons are classes for which there is only one instance. Since there is one and only one instance, all of the necessary data can be held in class variables and manipulated with class methods, consequently there are no instance variables. Also, we only need to initialize our class variable, protoNotes, once. By storing our SynthDefs in a singleton class, we keep them together in one place, and we don't have to clutter up our score file with SynthDefs or worry about loading them onto the Server (since CtkNoteObject takes care of that for us). Likewise they are available to any class that needs them.

We've opted to use a wave-shaping instrument for this example (\NRT_shaper), which requires an audio source. In this case we'll use a simple oscillator (\NRT_sinosc). Additionally, we'll create a separate singleton class for control-rate instruments:

```
Complementary_Controllers {

    classvar <protoNotes;

    *initClass {
        StartUp.add({
            protoNotes = CtkProtoNotes.new(
```

```
                SynthDef.new(\NRT_envgen,{
                    arg outbus = 0, gate = 1, levelScale = 1,
                    levelBias = 0, timeScale = 1, doneAction = 0;
                    Out.kr(outbus,(
                        ((EnvGen.kr(Control.names([\env])
                            .kr(Env.newClear(128)), gate,
                            timeScale:timeScale,doneAction:doneAction)
                            ) * levelScale) + levelBias)
                        );
                    }
                ),
                SynthDef.new(\NRT_cntlProduct,{
                    arg outbus = 0, left = 1, right = 1;
                    Out.kr(outbus, (left * right));
                    }
                )
            )
        });
    }
}
```

Figure 18.41

1.4 Data Processing

Since we are using an oscillator for our source, we can achieve a legato effect by changing the frequency. There is no need to articulate every note. We do however want to articulate the beginning and ending of each phrase, otherwise the repeated notes won't sound like repeated notes; they'll just sound like out-of-place eighth-notes in a world of perpetual sixteenth-note motion. Therefore, given an array of PitchClass objects, we need to identify the beginnings and endings of each phrase, and build one frequency envelope and one amplitude envelope for it.

Since every phrase needs a frequency envelope and an amplitude envelope, we can encapsulate them into a class.

```
Complementary_Intro_Segment {

    var atomicDur, attackDur, releaseDur, shortenBy,
        attackLevel, sustainLevel, releaseLevel;
    var pitchColl, <frequencyEnv, <amplitudeEnv;

    *new {arg pitchClasses, atomicDur, attackDur = 0.1,
        releaseDur = 0.1, shortenBy = 0.0, attackLevel = 0,
        sustainLevel = 1, releaseLevel = 0;
        ^super.newCopyArgs(atomicDur, attackDur, releaseDur,
            shortenBy, attackLevel, sustainLevel, releaseLevel
        ).initCompIntroSegment(pitchClasses);
    }

    initCompIntroSegment {arg pcArray;
        pitchColl = PitchCollection.new(pcArray.unbubble);
        //N.B. Env's require that their levels Array contain one more element
        // than their times Array. Since we want an Env with the number
        // of times equal to the number of elements in our pitchColl,
        // we'll prepend a duplicate of the first pitch in our frequencyEnv
        frequencyEnv = Env.new(
            Array.with(pcArray.at(0).keynum.midicps) ++
                Array.fill(pcArray.size, {arg i;
                    pcArray.at(i).keynum.midicps;
                }),
                Array.fill(pcArray.size, {arg i; atomicDur}), \step
        );
        amplitudeEnv = Env.new(
            [attackLevel, sustainLevel, sustainLevel, releaseLevel],
            [attackDur, this.sustainDur, releaseDur],
            [-4, 1, 4]
        );
```

```
    }

    sustainDur {
        ^((this.size * atomicDur)
        - (attackDur + releaseDur + shortenBy));
    }

    size {
        ^pitchColl.pitchCollection.size;
    }

    duration {
        ^this.size * atomicDur;
    }
}
```

Figure 18.42

Now when we parse our array of PitchClass objects and identify a phrase, we can simply pass it into an instance of our Complementary_Intro_Segment class and let it create the necessary Envs for us.

So now we need a way to parse our PitchClass array and determine the beginning and ending of each phrase. To do this we need to step through the array, and where we find repeated pitches, create a new Complementary_Intro_Segment with all the PitchClass objects that precede the repeated pitch, then start looking for the next repeated pitch or the end of the array. We can encapsulate this into a single class.

```
Complementary_Intro_Data {

    var pitchClasses, <atomicDur, <segments;

    *new {arg pcArray, noteDur;
        ^super.new.initCompIntroData(pcArray, noteDur);
    }

    initCompIntroData {arg pcArray, noteDur;
        pitchClasses = pcArray; // or use PitchCollection
        atomicDur = noteDur;
        segments = Array.newClear(0);
        this.processPitchCollIntoSegments;
    }

    //loop through the pitchClasses to see if there are any repeated notes
    // if there are, split pitchClasses into segments at those points
    processPitchCollIntoSegments {
            var subSet;
            subSet = Array.newClear(0);

            (pitchClasses.size).do({arg ndx;
                if((subSet.size == 0), {
                    //if subSet is empty we're starting a new phrase
                    subSet = subSet.add(pitchClasses.at(ndx));
                }, {
                    //subSet isn't empty so we may be continuing a phrase
                    if(subSet.at(subSet.size - 1) == pitchClasses.at(ndx), {
                        //we've found a repeated note, so end the current segment,
                        // and start a new one
                        segments = segments.add(Complementary_Intro_Segment(subSet, atomicDur));
                        subSet = Array.with(pitchClasses.at(ndx));
                }, {
                    //This pitchClass is a continuation of the sequence...
                    subSet = subSet.add(pitchClasses.at(ndx));
                });
            });
        });
        //be sure to add the last subSet to segments
        segments = segments.add(Complementary_Intro_Segment(subSet, atomicDur));
```

```
    }

    size {
        var size = 0;
        segments.do({arg seg; size = size + seg.size;});
        ^size;
    }

    duration {
        ^this.size * atomicDur;
    }
}
```

Figrure 18.43

1.5 Building the Score

Now that we have a way to enter the data using our Complementary_Grundgestalt class and process it using Complementary_Intro_Data and Complementary_Intro_Segment classes, we need one last class to connect our processed data to our instruments in order to build notes that we can add to our score.

```
Complementary_Intro_Part {

    var data, weights, server, group, cntlGroup, procGroup, srcGroup, globalAmp, chebyshev,
        sine, cntlProduct, <score;

    *new {arg starttime = 0.0, introData, globalAmpEnv,
        addAction = 0, target = 1, server, weights;
        ^super.new.initCompIntroPart(starttime, introData, globalAmpEnv,
            addAction, target, server, weights);
    }

    //given a Complementary_Intro_Data and amplitude envelope, build a score
    // for one part of the Complementary Quartet
    initCompIntroPart {arg starttime, introData, globalAmpEnv,
        addAction = 0, target = 1, server, weights;
        var localStart;

        data = introData;
        weights = weights ?? {[1, 0, 1, 1, 0, 1]};
        server = server ?? {Server.default};

        group = CtkGroup.new(starttime, data.duration,
                addAction: addAction, target: target, server: server);

        cntlGroup = CtkGroup.new(starttime, data.duration,
                    addAction: \head, target: group, server: server);

        procGroup = CtkGroup.new(starttime, data.duration,
                    addAction: \tail, target: group, server: server);

        srcGroup = CtkGroup.new(starttime, data.duration,
                    addAction: \before, target: procGroup, server: server);


        chebyshev = Complementary_WaveShaper.new(starttime, data.duration, weights,
                    addAction: \head, target: procGroup, server: server);

        //create a source to feed to the chebyshev wave-shaper instrument...
        sine = Complementary_Instruments.protoNotes[\NRT_sinosc]
                   .note(starttime, (data.duration), \head, srcGroup, server);
        sine.outbus_(chebyshev.inbus);

        //apply dynamics globally...
        globalAmp = Complementary_EnvGen.new(starttime, data.duration, globalAmpEnv,
                    addAction: \head, target: cntlGroup, server: server);

        //Mix the global amplitude envelope with the local amplitude envelope
        // once the local env is built in the segments loop below
```

```
        cntlProduct = Complementary_ControlProduct.new(starttime, data.duration,
                      right: globalAmp.outbus, addAction: \head, target: cntlGroup,
                      server: server);

        score = CtkScore.new(chebyshev.waveTable, group, cntlGroup, procGroup, srcGroup,
                globalAmp.note, cntlProduct.note, chebyshev.note, sine);

        //now loop through the segments in data and add an EnvGen each for
        // sine.amp and sine.freq
        data.segments.size.do({arg ndx;
            var localAmpEnvGen, localFreqEnvGen, segment;

            segment = data.segments.at(ndx);

            //first we need a starttime (localStart) that is local to each
            // segment
            if((ndx == 0),
                {localStart = starttime;},
                {localStart = localStart + data.segments.at(ndx - 1).duration;}
            );

            localAmpEnvGen = Complementary_EnvGen.new(localStart, segment.amplitudeEnv.times.sum,
                             segment.amplitudeEnv, addAction: \head, target: cntlGroup,
                             server: server);

            //map the local amplitude env to the global cntlProduct's left parameter
            // set the time for this mapping relative to cntlProduct's starttime
            cntlProduct.left_(localAmpEnvGen.outbus, localStart - cntlProduct.starttime);

            localFreqEnvGen = Complementary_EnvGen.new(localStart, segment.duration,
                              segment.frequencyEnv, addAction: \tail, target: cntlGroup,
                              server: server);

            //map the results of the global cntlProduct to
            // this segemnt's amplitude, relative to sine's starttime
            sine.amp_(cntlProduct.outbus, localStart - sine.starttime);
            sine.freq_(localFreqEnvGen.outbus, localStart - sine.starttime);

            score.add(localAmpEnvGen.note, localFreqEnvGen.note;);
        });
    }
}
```

Figure 18.44

1.6 Inside the Complementary_Intro_Part

The main goal of this class is to create a wave-shaping instrument (the instance variable chebyshev) that plays for the duration of our example, plus the requisite envelopes for the frequency and amplitude that play for the duration of each segment. Additionally we need a global amplitude envelope for the dynamics, a source to feed to our wave-shaper, and a few supporting objects like CtkGroup, CtkControl, CtkBuffer, and CtkAudio. Finally we need to add all of these to a CtkScore.

As with most of the classes we've presented in this chapter, the initialization method handles most of the work.

1.6.1 Supporting Objects

First our initialization method declares a variable, localStart, that we'll use later. Then it sets some initial values for data, weights (i.e. weights for the chebyshev polynomials), and server. Next our initialization method creates four CtkGroups we'll use to ensure that everything executes in the correct order.

```
        group = CtkGroup.new(starttime, data.duration,
                addAction: addAction, target: target, server: server);

        cntlGroup = CtkGroup.new(starttime, data.duration,
                    addAction: \head, target: group, server: server);

        procGroup = CtkGroup.new(starttime, data.duration,
                    addAction: \tail, target: group, server: server);

        srcGroup = CtkGroup.new(starttime, data.duration,
                    addAction: \before, target: procGroup, server: server);
```

Figure 18.45

1.6.2 Global Audio Objects

With our groups in place, we can create our wave-shaper and oscillator. With our waveshaper, we've taken the opportunity to encapsulate the CtkBuffer, the CtkAudio busses, the call to our CtkNoteObject, and the subsequent CtkNote into a single class, which helps tidy up our initialization method. As the class is fairly strait forward, we'll include the code but skip over a detailed discussion.

```
Complementary_WaveShaper {

    var <note, <waveTable;

    *new {arg starttime = 0.0, duration, weights, inbus, outbus, addAction, target, server;
        ^super.new.initCompWaveShaper(starttime, duration, weights, inbus, outbus,
            addAction, target, server);
    }

    initCompWaveShaper {arg starttime, duration, weights, inbus, outbus,
        addAction, target, server;
        weights = weights ?? {[1, 0, 1, 1, 0, 1]};

        waveTable = CtkBuffer.new(size: 512, server: server).load(sync: true);
        waveTable.cheby(0.0, 1, 1, 1, weights.unbubble);

        inbus  = inbus ?? {CtkAudio.new(numChans: 1, server: server)};
        outbus = outbus ?? {CtkAudio.new(bus: 0, numChans: 1, server: server)};

        //Process Instrument, apply wave shape distortion to an input
        note = Complementary_Instruments.protoNotes[\NRT_shaper].note(starttime,
               duration, addAction, target, server)
                   .bufnum_(waveTable.bufnum)
                   .outbus_(outbus)
                   .inbus_(inbus);
    }

    inbus {
        ^note.inbus;
    }

    outbus {
        ^note.outbus;
    }

    inbus_ {arg ctkAudio, time = 0.0;
        note.inbus_(ctkAudio, time);
    }

    outbus_ {arg ctkAudio, time = 0.0;
        note.outbus_(ctkAudio, time);
    }
}
```

Figure 18.46

1.6.3 Global Control Objects

Next we create an instance of Complementary_EnvGen, a class we created to encapsulate an EnvGen with an Env and CtkControl.

```
globalAmp = Complementary_EnvGen.new(starttime, data.duration, globalAmpEnv,
            addAction: \head, target: cntlGroup, server: server);
```

Figure 18.47

CtkControl.env provides the same functionality, but with a smaller array allocation for its myEnv Control (CtkControl.env sets aside 16 slots for its input Env whereas Complementary_EnvGen's SynthDef, \NRT_envgen, sets aside 128). While the implementation included in CtkControl would suffice for our global amplitude envelope, we'll need the larger array for our frequency envelope later. As with Complementary_WaveShaper, the implementation is fairly strait forward and is presented here without discussion.

```
Complementary_EnvGen {

    var <note;

    *new {arg starttime = 0.0, duration, env, control,
        addAction = 1, target = 0, server;
        ^super.new.initCEG(starttime, duration, env, control, addAction, target, server);
    }

    initCEG {arg starttime = 0.0, duration, envelope, control, addAction = 1, target = 0, server;
        server = server ?? {Server.default};
        control = control ?? {CtkControl.new(1, envelope[0], starttime, server: server)};
        note = Complementary_Controllers.protoNotes[\NRT_envgen]
                    .note(starttime, duration, addAction, target, server)
                    .env_(envelope).outbus_(control);
    }

    env {
        ^note.env;
    }

    env_ {arg newEnv, time = 0.0;
        note.env_(newEnv, time);
    }

    outbus {
        ^note.outbus;
    }

    outbus_ {arg ctkCntl, time = 0.0;
        note.outbus_(ctkCntl);
    }
}
```

Figure 18.48

At this point, we do face a small obstacle. We want to map two amplitude envelopes to our oscillator, which has only one parameter for amplitude (amp). One solution would be to add another parameter to our oscillator for the global amplitude. However this solution does not scale well, i.e. if we later decide that we need even more layers of control, we'll have to create a new oscillator SynthDef with the requisite number of amplitude parameters for each layer. Instead, we've decided to create a SynthDef that outputs the product of two control signals. As with Complementary_EnvGen, we've included the code but have skipped over a detailed explanation.

```
Complementary_ControlProduct {

    var <note;

    *new {arg starttime = 0.0, duration, left, right, outbus,
        addAction = 1, target = 0, server;
        ^super.new.initCEG(starttime, duration, left, right, outbus,
                addAction, target, server);
    }

    initCEG {arg starttime = 0.0, duration, left, right, outbus,
        addAction = 1, target = 0, server;
        server = server ?? {Server.default};

        left = left ?? {CtkControl.new(1, 1, starttime, server: server)};
        right = right ?? {CtkControl.new(1, 1, starttime, server: server)};
        outbus = outbus ?? {CtkControl.new(1, 1, starttime, server: server)};

        note = Complementary_Controllers.protoNotes[\NRT_cntlProduct]
                    .note(starttime, duration, addAction, target, server)
                        .left_(left)
                        .right_(right)
                        .outbus_(outbus);
    }

    left {
        ^note.left;
    }

    right {
        ^note.right;
    }

    starttime {
        ^note.starttime;
    }

    outbus {
        ^note.outbus;
    }

    left_ {arg ctkCntl, time = 0.0;
        note.left_(ctkCntl, time);
    }

    right_ {arg ctkCntl, time = 0.0;
        note.right_(ctkCntl, time);
    }
    outbus_ {arg ctkCntl, time = 0.0;
        note.outbus_(ctkCntl, time);
    }
}
```

Figure 18.49

Now, every time we want to add another layer, we can create a new instance of Complementary_ControlProduct and map the left and right inputs accordingly.

```
        cntlProduct = Complementary_ControlProduct.new(starttime, data.duration,
                      right: globalAmp.outbus, addAction: \head, target: cntlGroup,
                      server: server);
```

Figure 18.50

Note that at this point we map the globalAmp.outbus to the right parameter (choice of sides makes no difference here, we could have chosen the left side just as easily). We'll map the local amplitude EnvGen for each segment separately below.

1.6.4 Collecting Global Objects into a CtkScore

The next line in our initialization method creates a CtkScore and adds all of our global events to it.

```
score = CtkScore.new(chebyshev.waveTable, group, cntlGroup, procGroup, srcGroup,
        globalAmp.note, cntlProduct.note, chebyshev.note, sine);
```

Figure 18.51

1.6.5 Local Objects

The only thing left is to handle the local events.

```
data.segments.size.do({arg ndx;
    var localAmpEnvGen, localFreqEnvGen, segment;

    segment = data.segments.at(ndx);

    //first we need a starttime (localStart) that is local to each
    // segment
    if((ndx == 0),
        {localStart = starttime;},
        {localStart = localStart + data.segments.at(ndx - 1).duration;}
    );

    localAmpEnvGen = Complementary_EnvGen.new(localStart, segment.amplitudeEnv.times.sum,
                     segment.amplitudeEnv, addAction: \head, target: cntlGroup,
                     server: server);

    //map the local amplitude env to the global cntlProduct's left parameter
    // set the time for this mapping relative to cntlProduct's starttime
    cntlProduct.left_(localAmpEnvGen.outbus, localStart - cntlProduct.starttime);

    localFreqEnvGen = Complementary_EnvGen.new(localStart, segment.duration,
                      segment.frequencyEnv, addAction: \tail, target: cntlGroup,
                      server: server);

    //map the results of the global cntlProduct to
    // this segemnt's amplitude, relative to sine's starttime
    sine.amp_(cntlProduct.outbus, localStart - sine.starttime);
    sine.freq_(localFreqEnvGen.outbus, localStart - sine.starttime);

    score.add(localAmpEnvGen.note, localFreqEnvGen.note;);
});
```

Figure 18.52

In this loop, we want to create EnvGens for amplitude and frequency for each segment and map them to our global wave-shaper source, sine. Two details to note are that we map the left parameter of our global cntlProduct to the outbus of our local amplitude EnvGen (localAmpEnvGen). The other detail to note is that all of our local assignments (cntlProduct.outbus_, sine.amp_, and sine.freq_) include a time parameter that specifies when the assignment should occur relative to the receiver (i.e. relative to cntlProduct.starttime or sine.starttime respectively). The last thing to do for our Complementary_Intro_Part class is to add the local events to the score.

1.7 Putting It to Use

Figure 18.53 shows how to combine the classes we've built for one part from the quartet.

```
s = Server.local
s.boot;
(
~baseDur = 0.125;

~time = NRT_TimeFrame.new(starttime: {arg total, curr, dur; ((total - curr) * dur);});

////////////////////////////////////////
//                                    //
//       E N T E R  T H E  D A T A    //
//                                    //
////////////////////////////////////////

//FIRST VOICE
o = Complementary_Grundgestalt.new;

k = [
[o.arrayAt(\201), o.arrayAt(\012), o.arrayAt(\021), o.arrayAt(\021)],
[o.arrayAt(\201), o.arrayAt(\102), o.arrayAt(\012), o.arrayAt(\120)],
[o.arrayAt(\210), o.arrayAt(\201), o.arrayAt(\021), o.arrayAt(\021)],
[o.arrayAt(\012), o.arrayAt(\102), o.arrayAt(\102), o.arrayAt(\021)],

[o.arrayAt(\201), o.arrayAt(\012), o.arrayAt(\021), o.arrayAt(\021)],
[o.arrayAt(\201), o.arrayAt(\102), o.arrayAt(\012), o.arrayAt(\120)],
[o.arrayAt(\210), o.arrayAt(\201), o.arrayAt(\021), o.arrayAt(\021)],
[o.arrayAt(\012), o.arrayAt(\102), o.arrayAt(\102), o.arrayAt(\021)],

[o.arrayAt(\201), o.arrayAt(\102), o.arrayAt(\012), o.arrayAt(\120)],
[o.arrayAt(\012), o.arrayAt(\102), o.arrayAt(\102), o.arrayAt(\021)],
[o.arrayAt(\201), o.arrayAt(\012), o.a, o.b]
].flat;

//SECOND VOICE
p = Complementary_Grundgestalt.new(o.a.keynum - 9);

l = [
[p.c, p.a,],
[p.arrayAt(\201), p.arrayAt(\102), p.arrayAt(\021), p.arrayAt(\201)],
[p.arrayAt(\201), p.arrayAt(\201), p.arrayAt(\012), p.arrayAt(\021)],
[p.arrayAt(\201), p.arrayAt(\201), p.arrayAt(\102), p.arrayAt(\021)],

[p.arrayAt(\012), p.arrayAt(\012), p.arrayAt(\120), p.arrayAt(\120)],
[p.arrayAt(\201), p.arrayAt(\102), p.arrayAt(\021), p.arrayAt(\201)],
[p.arrayAt(\201), p.arrayAt(\201), p.arrayAt(\102), p.arrayAt(\021)],
[p.arrayAt(\012), p.arrayAt(\012), p.arrayAt(\021), p.arrayAt(\201)],

[p.arrayAt(\201), p.arrayAt(\201), p.arrayAt(\012), p.arrayAt(\021)],
[p.c, p.a]
].flat;

//THIRD VOICE
q = Complementary_Grundgestalt.new(p.a.keynum - 9);

m = [
[q.arrayAt(\012), q.arrayAt(\012)],
[q.arrayAt(\201), q.arrayAt(\021), q.arrayAt(\012), q.arrayAt(\201)],
[q.arrayAt(\201), q.arrayAt(\201), q.arrayAt(\021), q.arrayAt(\012)],
[q.arrayAt(\021), q.arrayAt(\201), q.arrayAt(\201), q.arrayAt(\021)],
[q.arrayAt(\201), q.arrayAt(\021), q.arrayAt(\012), q.arrayAt(\201)],

[q.arrayAt(\021), q.arrayAt(\012), q.arrayAt(\201), q.arrayAt(\021)],
[q.arrayAt(\120), q.arrayAt(\201), q.a, q.b]
].flat;

//FOURTH VOICE
r = Complementary_Grundgestalt.new(q.a.keynum - 9);

n = [
[r.arrayAt(\021), r.arrayAt(\120), r.arrayAt(\012), r.arrayAt(\201)],
[r.arrayAt(\201), r.arrayAt(\012), r.arrayAt(\201), r.arrayAt(\012)],
[r.arrayAt(\021), r.arrayAt(\201), r.arrayAt(\102), r.arrayAt(\012)],
```

```
[r.arrayAt(\021), r.arrayAt(\120), r.arrayAt(\012), r.arrayAt(\210)],
[r.arrayAt(\021), r.arrayAt(\201), r.arrayAt(\102), r.arrayAt(\012)],
[r.arrayAt(\021), r.arrayAt(\201)]
].flat;

////////////////////////////////////////
//                                    //
//     P R O C E S S  T H E  D A T A  //
//                                    //
////////////////////////////////////////


//FIRST VOICE
t = Complementary_Intro_Data.new(k, ~baseDur);

//SECOND VOICE
u = Complementary_Intro_Data.new(l, ~baseDur);

//THIRD VOICE
v = Complementary_Intro_Data.new(m, ~baseDur);

//FOURTH VOICE
w = Complementary_Intro_Data.new(n, ~baseDur);

////////////////////////////////////////
//                                    //
//      B U I L D  T H E  S C O R E   //
//                                    //
////////////////////////////////////////

// BUILD  THE  PARTS //
~firstVoice  = Complementary_Intro_Part.new(~time.starttime(k.size, k.size, ~baseDur), t,
Env.new([0, 0.707, 0.0125], [0.01, k.size * ~baseDur], [1, -4]), server: s);

~secondVoice = Complementary_Intro_Part.new(~time.starttime(k.size, l.size, ~baseDur), u,
Env.new([0, 0.707, 0.0125], [0.01, l.size * ~baseDur], [1, -4]), server: s);

~thirdVoice  = Complementary_Intro_Part.new(~time.starttime(k.size, m.size, ~baseDur), v,
Env.new([0, 0.707, 0.0125], [0.01, m.size * ~baseDur], [1, -4]), server: s);

~fourthVoice = Complementary_Intro_Part.new(~time.starttime(k.size, n.size, ~baseDur), w,
Env.new([0, 0.707, 0.0125], [0.01, n.size * ~baseDur], [1, -4]), server: s);

// ADD  THE  PARTS  TO  A  SCORE //
//  take advantage of the fact that we can add CtkScores to a CtkScore


CtkScore.new(~firstVoice.score, ~secondVoice.score, ~thirdVoice.score, ~fourthVoice.score).play;
)
```

Figure 18.54

Since the only rests in this example occur before the entrance of each voice, we can subtract the number of notes for a given voice from the total number of notes for the section. That will tell us the number of notes for the rest. If we multiply that by our baseDur, the result tells us the amount of time after the first voice starts until the given voice enters.

1.8 Conclusion

There are a number of important aspects to this example. First, to understand music in general, we try to establish the relationships between its constituent parts. We try to identify groupings of pitches, rhythms, etc. that are recurrent, and give those groupings names. This process is analogous to modular

programming, where we try to identify recurrent lines of code, collect them together as a function, a method, or a class, and give them a name.

Over several centuries, we have developed a sophisticated written language for music. It makes great strides at capturing the relative nature of music through abstract concepts like a whole note, a music staff, a scale, a tempo. Computer programming makes it possible to model these relationships, indeed any relationship you can imagine. With a little bit of consideration, you can build code that allows you to work with these relationships, rather than low-level, specific values. In this way, you can build gestures, sections, and entire pieces that offer you many more opportunities to edit, change, and shape, even after you've written the notes.

[i] **PitchClass** is included in the Ctk **Quark**, along with a number of other useful classes such as **PitchCollection** and **PitchInterval**. See the respective help files for more information.