INSTITUTE OF SONOLOGY

Royal Conservatoire The Hague

Patterns

Programming and Music

http://www.bjarni-gunnarsson.net

Generative Processes

Deterministic

Deterministic procedures generate musical material by carrying out a fixed, rule-based compositional task that does not involve random selection.

The variables supplied to a deterministic procedure are called the seed data.

The seed data can be a set of pitches, a musical phrase, or some constraints that the procedures must satisfy.

Stochastic

Stochastic procedures, on the other hand, integrate random choice into the decision-making process.

A basic stochastic generator produces a **random** number and compares it to values stored in a probability table.

If the random number falls within a certain range of values in the probability table, the algorithm generates the event associated with that range.

By weighting the probability of certain events over others, one can guarantee an overall trend of possibilities.

SynthDefs

SynthDefs

A ***SynthDef*** is a ***client-side representation*** of a synthesis process running on the sound server.

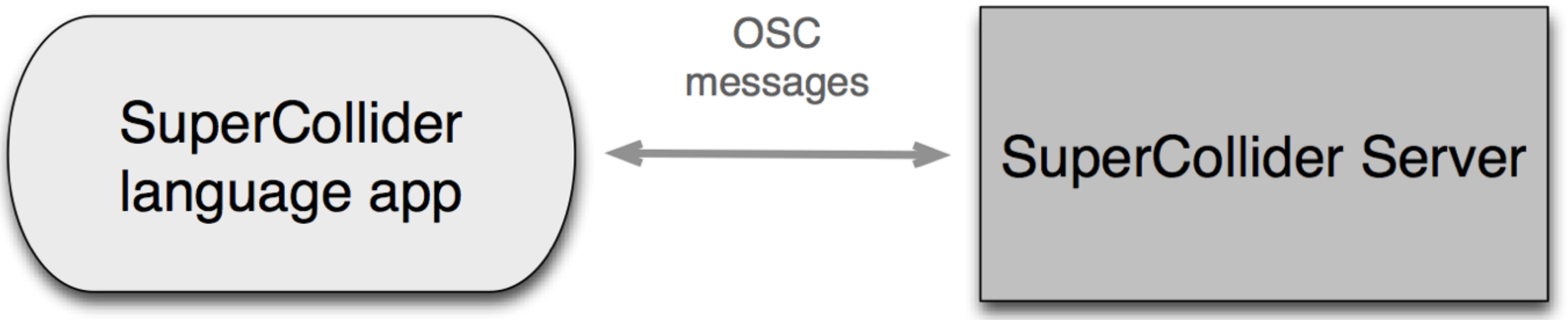
The definition has to be send to the ***server*** that in turn computes an ***audio graph*** used for synthesis.

Synthesis processes have two main rates: ***audio rate*** and ***control rate***.

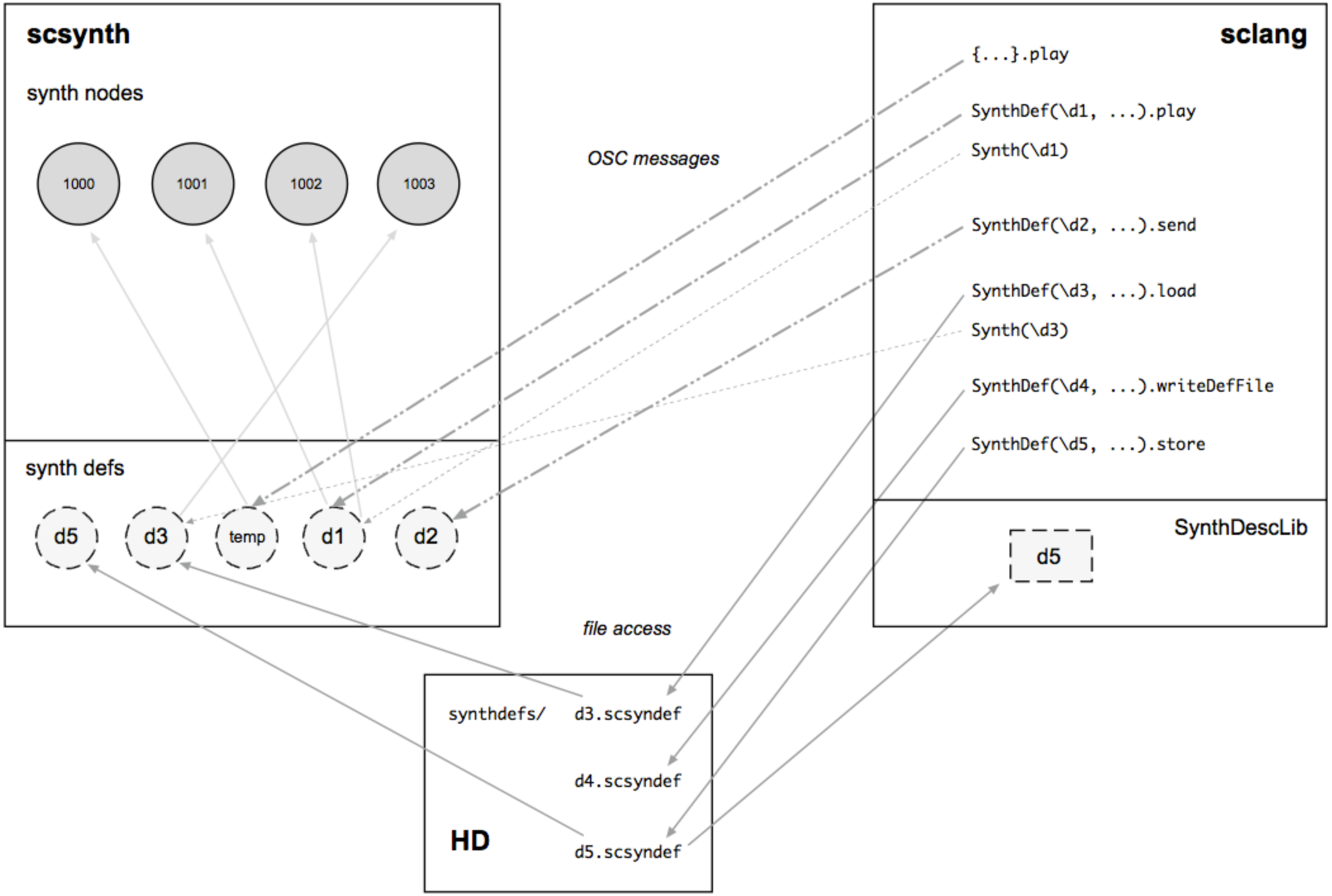
These are capable of more intense data processing than the SuperCollider language and the reason why a separation exists.

The SuperCollider language is used to create the definitions that contains ***arguments*** that can be changed at a later point.

SynthDefs

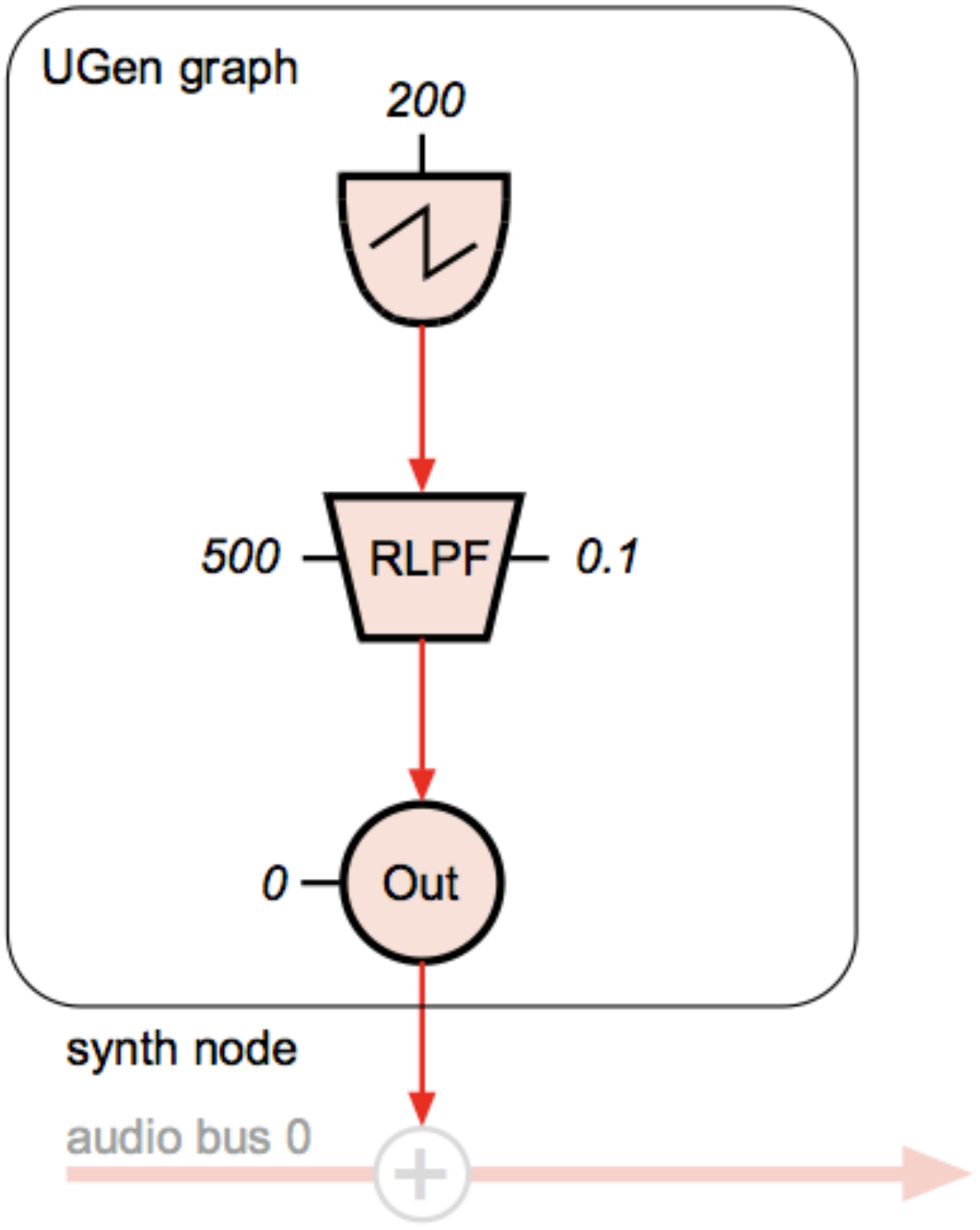

SuperCollider language app
OSC messages
SuperCollider Server

SynthDefs



scsynth
synth nodes
1000
1001
1002
1003
synth defs
d5
d3
temp
d1
d2
OSC messages
sclang
{...}.play
SynthDef(\d1, ...).play
Synth(\d1)
SynthDef(\d2, ...).send
SynthDef(\d3, ...).load
Synth(\d3)
SynthDef(\d4, ...).writeDefFile
SynthDef(\d5, ...).store
SynthDescLib
d5
file access
synthdefs/
d3.scsyndef
d4.scsyndef
d5.scsyndef
HD

SynthDefs



Synth Nodes and UGen graphs

```
(
SynthDef("simple", {
var sig;
sig = Saw.ar(200);
sig = RLPF.ar(sig, 500, 0.1);
Out.ar(0, sig);
}).play;
)
```

SynthDefs

Name of synth

Arguments for the synth (inputs)

```
SynthDef(\mySynth, {
    |amp=0.1, freq=440, pan=0|
    var env, sig;
    env = EnvGen.kr(Env.perc, doneAction:2);
    sig = SinOsc.ar(freq) * env;
    sig = Pan2.ar(sig, pan, amp);
    Out.ar(0, sig);
}).add;
```

Variable for the synth (storage boxes)

Ugens that generate and modify signals

Final signal is sent to an output bus

SynthDefs need to be added to the synthesis server

Patterns

Algorithmic Composition

Composing music, sounds and behavior using algorithms and computer programs.

Possible methods for doing this with:

* *Randomness*

* *Stochastic processes*

* *Selection principles*

* *Markov chains*

* *Random walks*

* *Grammars*

* *Genetic algorithms*

All these can be realized using the ***Pattern library*** in SuperCollider.

Approaches with Patterns

Two opposite poles for composition are the ***top-down*** approach and the ***bottom-up*** approach.

A top-down model is often specified with the assistance of ***"black boxes"*** these can later be edited, replaced or upgraded without destroying the whole model or process.

Patterns in SuperCollider specify behavior instead of details. This means they are ideal to experiment with top-down approaches and methods that ***combine patterns*** or ***blocks of patterns***.

Composing with Parameters

A ***parameter*** is one of the variables that controls the outcome of a system.

Attributes of a process are converted to values representing its state where its properties and variability control the value settings.

Parametrical thinking enables ***limits***, ***boundaries***, ***parameter spaces*** and ***mapping*** from one to the other.

Parameter mappings include ***one-to-many***, ***many-to-one*** and ***many-to-many***.

Patterns in SuperCollider relate generative behavior to a synthesis or sound process ***parameter***.

Representations

How the different possibilities of composing with computers are made available to users is a problem of ***representation***.

The ***possible operations*** on sounds, notes, lower or higher-level structures is that what enables meaningful interaction.

The precise definition of ***materials***, ***possible operations*** that ***transform*** the material and ***relationship*** between them is perhaps part of what is to be composed.

Patterns in SuperCollider allow a composer to define how to ***interact*** with his materials by defining their ***behavior*** and focus on ***abstract representations*** instead of detailed procedures.

SuperCollider

Among the design goals of SuperCollider:

* To realize ***sound processes*** that are different every time they are played.

* To write pieces in a way that describes ***a range of possibilities*** rather than a fixed entity.

Patterns allow us to compose music by describing possibilities and realize processes that are unique each time they run.

Patterns

Patterns describe calculations without an ***explicit definition*** of every step of a musical process.

Patterns represent a ***higher-level view*** of a computational task.

Patterns allow a composer to focus on ***parameters, relationships*** and ***behavior*** of musical materials instead of having to focus on detailed implementation.

Patterns allow to define what should happen instead of how exactly it happens.

Why Patterns?

The code is shorter and more clean.

Patterns are ***tested*** and their behavior works as specified opposed to a custom user process which could contain bugs.

Patterns are specified in a ***well defined format*** and are ***easy to read***.

There are over 150 different patterns covering many complex use cases and they can also be ***extended***.

Patterns allow us to focus on ***abstract but meaningful attributes*** instead of implementation details.

Patterns in SuperCollider

Patterns and Streams

Patterns can be seen as ***templates***

Patterns define ***behavior***

Streams ***execute*** the behavior

Patterns are ***stateless***, their definition does not change over time

A stream is what keeps track of ***where we are*** in the pattern's temporal evaluation

Patterns and Streams

A pattern does not have any knowledge of a ***current state*** so it can not proceed in time (calling next) or go backwards in time.

Invoking the methods **'*asStream*'** creates a stream specified by to a pattern. This stream can be advanced by calling ***´next´***.

A pattern specification can result in ***multiple instances*** of a stream.

To ***transform*** a pattern to a stream we use the .asStream message.

Streams

Routine and Task are ***subclasses*** of Stream.

A stream uses a ***lazy evaluation*** where an expression is only evaluated if the result is required.

Using a stream means obtaining ***one value at a time*** with a lazy evaluation using the ***.next*** message.

A stream sequence can be ***finite but also infinite***.

Thinking in patterns

Patterns can be seen as a different way of ***expressing musical thought***, describing behavior rather than precise orders

A focus on the representation means that the implementation could be done in another class library or even another programming language

The execution of precise and accurate parameters is avoided and instead the focus is on ***higher-level behavior***

Describing musical procedures using patterns requires much ***exercise and thinking*** since it differs from traditional approaches

A potential first step is to ***modularize*** events and ***build up*** from elementary blocks

List Patterns

Pseq(list, repeats, offset): Goes through a list linearly
Pser(list, repeats, offset): Play through the list as many times as needed, a 'repeats' number of times

- Lists with randomness -

Prand(list, repeats): Choose items from the list randomly
Pxrand(list, repeats): Choose randomly, but without repetition
Pshuf(list, repeats): Shuffle the list in random order
Pwrand(list, weights, repeats): Choose randomly, weighted probabilities
Pwalk(list, stepPattern, directionPattern, startPos): A random walk

Stochastic patterns

Pwhite(lo, hi, length): Random numbers with equal distribution
Pexprand(lo, hi, length): Random numbers with an exponential distribution, favoring lower numbers
Pbrown(lo, hi, step, length): Brownian motion, a value adds a random step to the previous value
Pbeta(lo, hi, prob1, prob2, length): Beta distribution, where prob1 = α and prob2 = β.
Pcauchy(mean, spread, length): Cauchy distribution.
Pgauss(mean, dev, length): Guassian distribution.
Ppoisson(mean, length): Poisson distribution.

Repetition and Constraint patterns

Pseries(start, step, length): Arithmetic series, successively adding 'step' to the starting value, returning a total of 'length' items.
Pgeom(start, grow, length): Geometric series, successively multiplying the current value by 'grow'.
Pseg(levels, durs, curves, repeats): Similar to Pstep, but interpolates to the next value.
Pkey(key): Read the 'key' in the input event, making previously-calculated values available for other streams.
Pfunc(nextFunc, resetFunc): The next value is the return value from evaluating nextFunc.
Proutine(routineFunc): Use the routineFunc in a routine.

Parallelizing event patterns

Ppar(list, repeats): Start each of the event patterns in the 'list' at the same time.

Ptpar(list, repeats): Start patterns with offset.

Pgpar(list, repeats): Like Ppar, but it creates a separate group for each subpattern.

Pspawner(routineFunc): Function is used to make a routine where a Spawner object gets passed into this routine.

Pspawn(pattern, spawnProtoEvent): Uses a pattern to control the Spawner object instead of a routine function.

Pbind

Pbind is a way to give ***names to values*** coming out of the types of patterns.

When one asks a Pbind stream for its next value, the result is an object called an ***Event***. Like a Dictionary, an event is a set of **"*key-value pairs*"**

A Pbind's stream generates ***Events***.

The Event class provides a default ***event prototype*** that includes powerful options to create and manipulate objects on the server.

An Event can also be ***extended*** and use custom configurations.

Pbind

Arguments for the synth

The synth to play

```
Pbind(
   \instrument, "mySynth",
   \freq, Pwhite(100, 1000),
   \amp, Pseq([0.1, 0.2], inf),
   \dur, 0.1
).play
```

Values for the arguments (can be patterns)

Exercises

```
(
    NF(\iop, {|freq=78, mul=1.0, add=0.0|
        var noise = LFNoise1.ar(0.001).range(freq, freq + (freq * 0.1));
        var osc = SinOsc.ar([noise, noise * 1.04, noise * 1.02, noise * 1.08],0,0.2);
        var out = DFM1.ar(osc,freq*4,SinOsc.kr(0.01).range(0.92,1.05),1,0,0.005,0.7);
        HPF.ar(out, 40)
    }).play;
)


(
    NF(\dsc, {|freq = 1080|
        HPF.ar(
            BBandStop.ar(Saw.ar(LFNoise1.ar([19,12]).range(freq,freq*2), 0.2).excess(
            SinOsc.ar( [freq + 6, freq + 4, freq + 2, freq + 8])),
            LFNoise1.ar([12,14,10]).range(100,900),
            SinOsc.ar(20).range(9,11)
        ), 80)
        ;
    }).play;
)
```

```
var <>pindex, <>cindex;

initialize {
    if(pindex.isNil, { pindex = 1000 });
    if(cindex.isNil, { cindex = 2000 });
}

clearProcessSlots {
    pindex = 1000;
    (this.pindex - 1000).do{|i| this[this.pindex+i] = nil; }
}

clearOrInit {|clear=true|
    if(clear == true, { this.clearProcessSlots() }, { this.initialize() });
}

transform {|process, index|
    if(index.isNil && pindex.isNil, {
        this.initialize();
    });

    pindex = pindex + 1;
    this[pindex] = \filter -> process;
}

control {|process, index|
    var i = index;

    if(i.isNil, {
        this.initialize();
        cindex = cindex + 1;
        i = cindex;
    });

    this[i] = \pset -> process;
}
```

```
(
    NF(\depfm, {|freqMin=5, freqMax=20, mul=20, add=80, rate=0.5, modFreq=2100, index=0.3, amp=0.2|
        var trig, seq, freq;
        trig = Dust.kr(rate);
        seq = Dwhite(freqMin, freqMax, inf).midicps;
        freq = Demand.kr(trig, 0, seq);
        HPF.ar(FMOsc.ar(LFCub.kr([freq, freq/2, freq/3, freq/4], 0, mul, add),
        LFNoise1.ar(0.3).range(modFreq,modFreq*2), index) * amp, 50)
    }).play;
)
```

Exercises

Exercise 1: Basic Sequence
Create a Pbind that plays the MIDI notes 60, 64, 67, 72 in sequence with a duration of 0.2 seconds between each note. Use \instrument \sine.

Exercise 2: Random Pitches
Create a Pbind that plays random MIDI notes between 48 and 72 using Pwhite.
Set the duration to 0.15.

Exercise 3: Amplitude Pattern
Create a Pbind that plays MIDI note 60 repeatedly, but with an amplitude pattern that alternates between 0.5 and 0.2. Use Pseq for the amplitude.

Exercise 4: Duration Pattern
Create a Pbind with a fixed pitch (60) but varying durations using Pseq.
Use the durations: [0.25, 0.25, 0.5, 0.25].

Exercise 5: Scale Degrees
Create a Pbind that plays scale degrees 0, 2, 4, 5, 7 using \degree.
Set the scale to Scale.minor and duration to 0.3.

Exercises

6. Implement a stochastic pattern where low pitches have a longer duration than high ones.

7. Implement a process that interpolates between two groups of pitches where the interpolation duration is variable

8. Implement a process based on two or more layers where moments occur with all layers playing at the same time and others with only a single layer playing.

9. Implement a process with various layers where each appears and later disappears gradually.