INSTITUTE OF SONOLOGY

Royal Conservatoire The Hague

Patterns

Composing with Algorithms
http://www.bjarni-gunnarsson.net

Functions and Iteration

Functions

Functions contain code that can used later or elsewhere in a program.

The code creating a function is called the **definition** of the function

When a function runs its said to be **called** or **evaluated**.

Functions are enclosed in **curly brackets** { }

Argument declarations, if any, follow the open bracket.

Variable declarations follow argument declarations.

The **function body** follows the variables and specifies its behaviour.

Functions

A function call has the following form:

«function_name»(«arguments»)

Functions can **return values** or **cause side-effects**, by manipulating external values. It is usually a good idea to use the return behavior.

A function returns the value of the **last statement** it executes.

An empty function returns the value **nil**.

A function executes when it receives the **value** message.

Functions

In SuperCollider, functions follow a 3 step **lifecycle**:

1. They are **compiled**
2. Then **evaluated**
3. Their evaluation result is **returned**

The compilation process first parses the function code and then translates it to byte code stored in the computer memory.

The translated byte code is accessible in the language if needed for example in the case of optimization.

Arguments

Functions can have **arguments** that are set each time the function is called. These define the inputs of the function and will can be varied.

Function arguments come at the **beginning** of the function, before any variables are declared.

Function arguments are declared either with **vertical bars ||** or the **arg keyword**.

If the number of arguments is unknown one can use ... in front of a name that will compile all provided arguments to a list variable corresponding to that name.

Arguments

Function arguments can have default values so that one does not need to specify an argument unless it is needed.

Arguments which do not have default values will be set to **nil** if no value is passed for them.

Default values can be literals but also expressions.

```
{ 'I am a function' }.value

{ 1 + 1 }.value

f = {arg a,b; a.pow(b)}
f.value(4,2)

f = {|... numbers| numbers.sum }
f.value(1,2,3)

{ |rand = (10.rand)| "Number is" + rand }.value
```

Iteration

When a task or function has to be executed repeatedly, an **iteration** is applied.

An example of an iteration is a **loop**.

A loop is when a **sequence of statements** is specified once but may be carried out several times in succession with changing variables.

Iteration is often performed to a **condition** where it iterates until the condition is met.

Iteration coupled with **conditions** attribute to the control flow of a program.

Iteration

In SuperCollider **Iteration** can be executed in various ways such as using the following:

*** do** *(execute a number of time or iterate a collection)*
*** for** *(go from a start to a end count and execute a function)*
*** forBy** *(like for but has a variable step size)*
*** while** *(execute while a certain test condition fails)*
*** loop** *(a function method and loops that function)*
*** repeat** *(repeats an object call a number of times)*

Additionally the collection objects have special iteration methods.

Iteration

```
do ( [1,2], {|item, i| (item * 10 + i).postln } )

7.do ( { rrand(10, 100).postln } )

for (10, 50, { arg i; i.postln });

forBy (10, 100, 10,  { arg i; i.postln });

x = Prand([10, 12]).loop.asStream;

x.nextN(32);
```

SynthDefs

SynthDefs

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

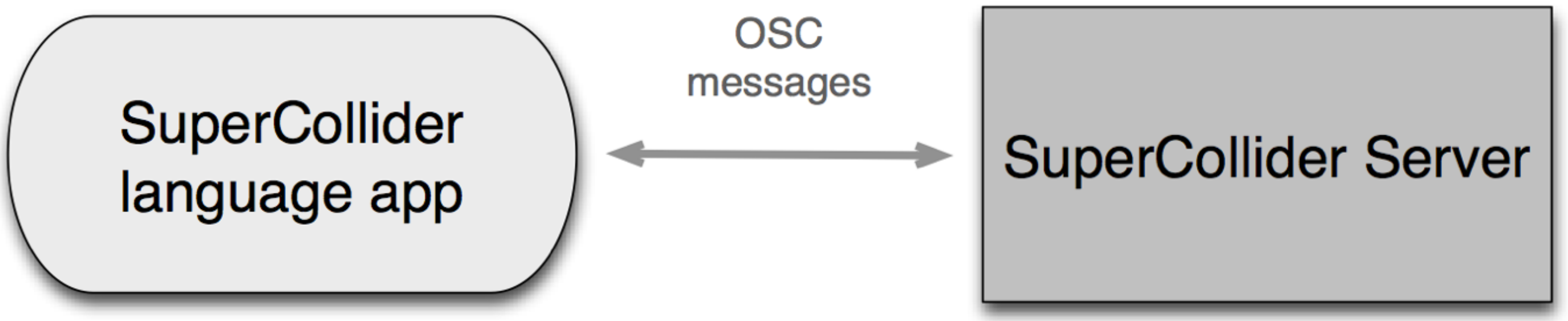
The **definition** has to be send to the **server** that in turn computes an **audio graph** used for synthesis.

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

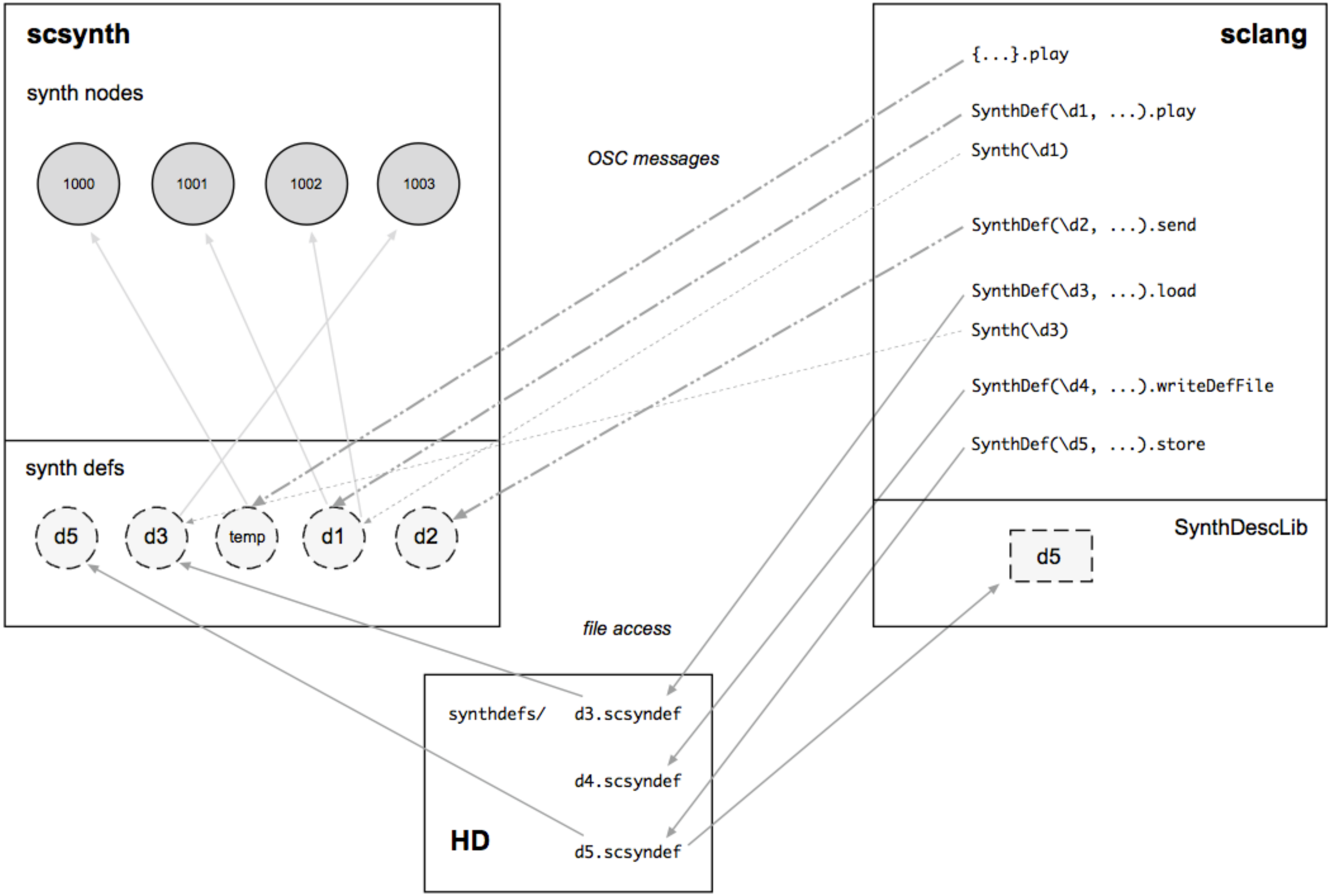
These are capable of more intense data processing than the SuperCollider language and the reason why a separation exists.

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

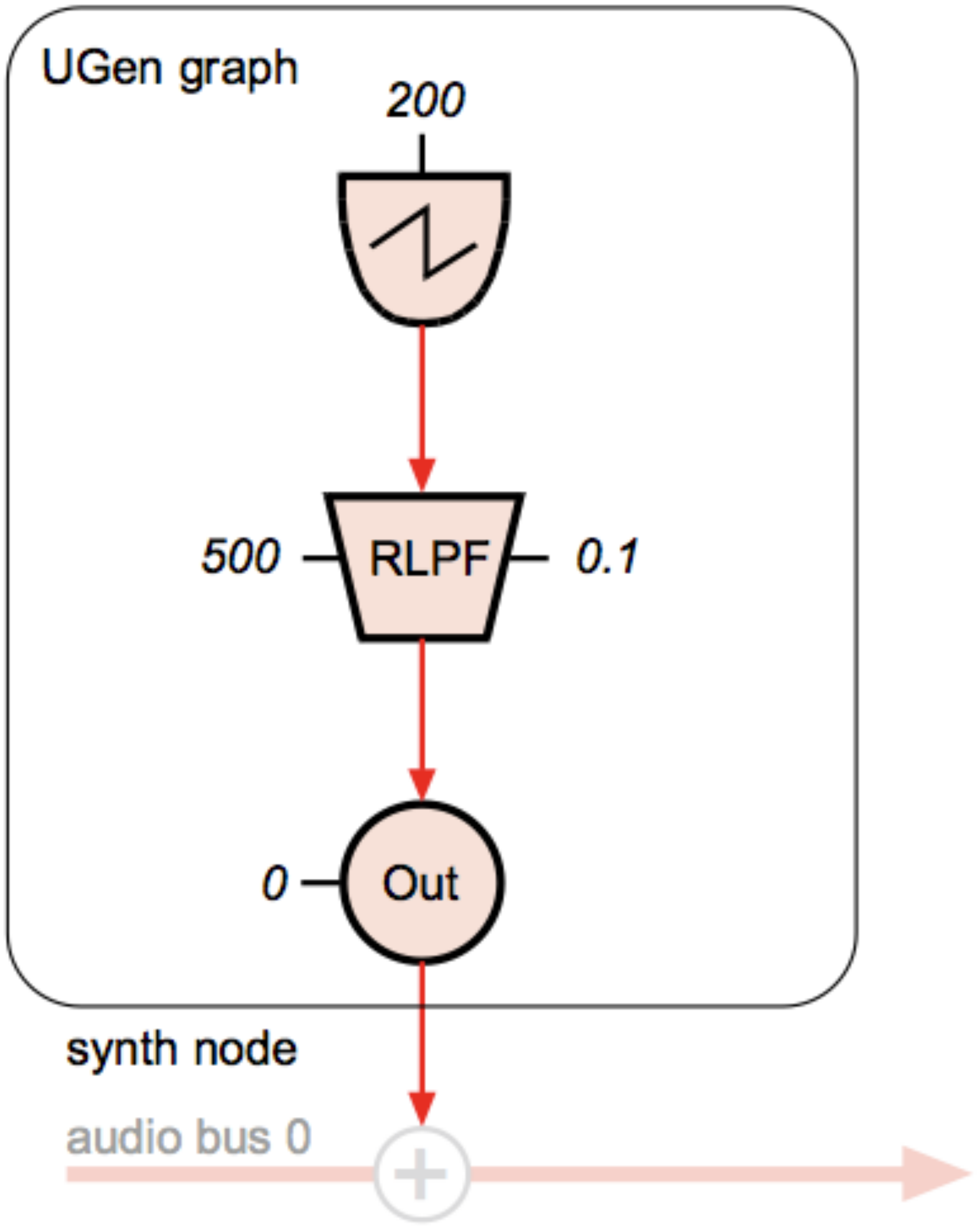
SynthDefs


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

Arguments for the synth (inputs)

Name of synth

```
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 behaviour 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 behaviour** to a synthesis or sound process parameter.

SuperCollider

Among the d**esign 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 **behaviour** 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 behaviour 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 **behaviour**

Streams execute the behaviour

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 **behaviour** 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

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 then build up.

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)

```
(
    Ndef(\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;
)


(
    Ndef(\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;
}
```

Exercises

```
(
    Ndef(\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(PMOsc.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

1. Create a Pbind sequence that consists of 4 different **durations** and 4 different **frequencies**

2. Create a Pbind sequence where **duration**, **frequency** and **amplitudes** are **randomly** determined

3. Create a Pbind sequence where frequency **goes up** in time while amplitude **goes down**

4. Create a sequence of at least 2 different Pbinds that are different in some way