

A Practical Guide to Patterns

H. James Harkins
SuperCollider Symposium 2009
Saturday, April 11, 2009

I. Why patterns?

In a word: *Abstraction*.

```
// Play a C major scale using Pbind.  
Pbind(  
  \degree, Pseries(0, 1, 8),  
  \dur, 0.25  
)play;
```

```
// Play a C major scale using a Task.  
Task({  
  #[60, 62, 64, 65, 67, 69, 71, 72].do { |note|  
    var synth = Synth.basicNew(\default, s);  
    s.sendBundle(s.latency, synth.newMsg(args: [freq: note.midicps, gate: 1, amp:  
0.1]));  
    thisThread.clock.sched(0.25 * 0.8, { s.sendBundle(s.latency,  
synth.releaseMsg); });  
    0.25.wait;  
  });  
}).play;
```

- The Task has to include lots of *implementation detail*. **Signal-to-noise ratio is lower.**
 - Signal = Compositional logic.
 - Noise = “Connective tissue,” including server messaging.
 - Is it immediately obvious that the MIDI note numbers are a major scale?
 - What percentage of the Task code is note-definition logic?
- The pattern *hides the connective tissue*, drawing attention to higher-level logic. **Signal-to-noise ratio is higher.**

I like patterns because the code is written in terms that are closer to musical thinking.

II. Streams

Patterns depend on Streams to do the work. A Stream is something that answers to *next* with a new value.

- Usually a Routine—can *yield* values in the middle and resume from that point.

```
~stream = Routine {  
  var x = 0;  
  loop {  
    x.yield; // each 'next' stops here  
    x = x + 1;  
  };  
};  
  
~stream.next;
```

next implies unidirectional access: no time travel!

Streams support lots of math and collection operations.

```
~stream2 = ~stream * 2;  
~stream2.next;
```

```
~stream2 = ~stream.reject(_ .odd);  
~stream2.next;
```

Patterns represent streams without imperative code

Like a cookie cutter: One pattern can make lots of independent streams.

- Pattern behavior is controlled by parameters.

A pattern by itself doesn't produce any values—must go through a Stream.

- The pattern makes a Stream in response to *asStream*.

```
p = Pseries(0, 1, inf);  
p.next;
```

```
q = p.asStream;  
q.next;
```

```
r = p.asStream;  
r.next;
```

III. Value patterns

Distinction between **value** and **event** patterns.

- **Value pattern:** Returns any kind of object, but usually numbers.
- **Event pattern:** Hold that thought... we'll get to it.

Basic generators

- **Generators**
 - Series: *Pseries*, *Pgeom*
 - Random: *Pwhite*, *Pexprand*, *Pbrown*, *Pgbrown*, *Pbeta*, *Pcauchy*, *Pgauss*, *Ppoisson*, *Pprob*
 - 3rd party: BhobUGens pack has some chaotic-function patterns.
 - Modulating parameters: Many patterns take patterns for parameters. (Neat trick with *Pseries*: Use a pattern for the *step* parameter and get linear motion.)

```
Pwhite(Pseries(0, 0.01, 100), 1.0, 100).asStream.all.plot;
```

```
p = Pbind(  
  \degree, Pseries(0, step: Prand(#[-1, 1], inf), length: inf).fold(-7, 9),  
  \dur, 0.125  
) .play;  
  
p.stop;
```

- **List patterns**
 - *Pseq, Prand, Pxrnd, Pwrand, Pshuf*
- What goes in the list?
 - A regular object (yielded as a return value).
 - A pattern or stream (which runs to completion before moving to the next list item).
- How does the list pattern know the difference?
 - It calls *embedInStream* on each item.
 - `1000.embedInStream` yields only 1000.
 - `Pwhite(1, 5, 3).embedInStream` yields 3 randomly chosen numbers.

```
Pseq([1000, Pwhite(1, 5, 3)], 2).asStream.all;
```

IV. Events

What's an Event? A set of named values with the ability to *play*.

```
(freq: 440, dur: 1).play;
```

play calls into the Event's *~play* function. The function comes from an *Event prototype*. A default Event prototype gets set up in the Event class and is always available.

- This is the "connective tissue."

```
Event.default[\play].postcs;
```

```
// Override normal ~play function; the Event does something else now
(freq: 440, dur: 1, play: { ~freq.debug("Event's frequency") }).play;
```

Event types

The default event uses *event types* to take different actions. These are some important ones. See [\[PG 08 Event Types and Parameters\]](#) for more.

- **note:** Plays a note (or notes), and schedules the release if the SynthDef is gated.
- **midi:** Send to a MIDIOut object, using *midicmd* for the MIDI message type.
- **on:** Starts a Synth but does not release it. Companion types:
 - **off:** Release a Synth created by *on*.
 - **kill:** Forcibly stop a Synth created by *on*.
 - **set:** Change controls in a Synth created by *on*.
 - **Off, kill** and **set** types need the node ID(s) in *~id*. The *on* type has a *callback* function so the caller can get the ID. Time doesn't permit a full explanation, but here's an example of how to use it. (*Penvir* sets up an *Environment* space to hold data global to the pattern it contains, and the callback function saves the node ID(s) into that Environment. A cleanup function makes sure the Synth is removed from the server upon stop.)

```
p = Penvir(), Pseq([
  Pfuncn({ (type: \on, dur: 0.25,
    callback: inEnvir { levent| ~id = event[\id] },
    addToCleanup: inEnvir { (type: \off, id: ~id).play } })
}, 1),
Pbind(
  \type, \set,
  \degree, Pwhite(0, 7, inf),
  \dur, 0.25
```

```
)
], 1)).play;

p.stop;
```

Most event types have some standard parameters. (You can ignore parameters not relevant to the SynthDef used in the Event.)

- **instrument:** Which SynthDef to use.
- **group, out:** Target group and output bus. (SynthDef should use *out* as an argument.)
 - **addAction:** Normally 0 (*\addToHead*).
- **amp:** Conventionally used for amplitude scaling. Useful only if the SynthDef has an 'amp' argument.
- **Timing**
 - **dur:** Duration until next event. Modified by *stretch*.
 - **legato:** Sustain time expressed as a factor of *dur*.
 - **stretch:** Multiplier for all time parameters.
 - **sustain:** Real number of beats to hold the note.

$$\text{sustain} = \text{dur} * \text{legato} * \text{stretch}$$
 - **delta:** Real number of beats until the next event.

$$\text{delta} = \text{dur} * \text{stretch}$$
 - **lag:** Delay the server message by this many seconds.
 - **strum:** When playing a chord, stagger the note onsets by this many beats.

```
p = Pbind(
  \note, [-3, 0, 2, 6, 7],
  \strum, 0.2,
  \dur, 1.0
).trace.play;

p.stop;
```

- **Pitch** (basically a chain from abstract units toward physical units)
 - **degree:** Scale degrees (scale given as an array).
 - **note:** Equal-tempered units of any size; arbitrary octave.
 - **midinote:** 12-tone ET units (60 = middle C).
 - **freq:** Hz.
 - Modifiers at each stage for transposition and stretching.

SynthDesc, SynthDescLib

The Event needs to know which of its values to include in server messages.

SynthDesc is a *SynthDef Descriptor* that knows the SynthDef's controls.

- It has a *msgFunc* to extract control values from the current Event.

```
SynthDescLib.global[\default].msgFunc.postcs;
```

When playing synths with patterns, *store the SynthDefs into the global SynthDescLib*.

- **.store:** Write the .scsyndef file to disk, and read the SynthDesc from it.
- **.memStore:** Send the SynthDef (no disk file) and create the SynthDesc in memory.

V. Event patterns

Event patterns' job is to put named values into the result events.

- **Pbind**, **Pmono**, **PmonoArtic**
- You can also write Events or Event patterns into list patterns (**Pseq** etc.).

```
Pbind(  
  \degree, Pseries(0, 1, 8),  
  \dur, 0.25  
) .play;
```

Pbind etc. work by associating names with value patterns.

- The C major scale example associates:
 - **\degree** with **Pseries(0, 1, 8)**
 - **\dur** with 0.25
- (**Pbind** automatically makes streams for the value patterns, internally.)

Every event, **Pbind** loops through the associations *in order*, gets a value from each stream, and puts them in the event.

- **First event:**
 - **\degree** gets 0 from **Pseries**
 - **\dur** gets 0.25
 - Result event is (**'degree': 0, 'dur': 0.25**)
- **Second event:**
 - **\degree** gets 1 from **Pseries**
 - **\dur** gets 0.25
 - Result event is (**'degree': 1, 'dur': 0.25**)

```
p = Pbind(  
  \degree, Pseries(0, 1, 8),  
  \dur, 0.25  
) .asStream;  
  
p.next(());
```

When does **Pbind** stop?

- When any of its value streams stops (returns nil).

What happens when you play an event pattern?

- Pattern → **EventStreamPlayer**
 - The **EventStreamPlayer** gets new events from the pattern, and plays them.
 - Reschedules after *delta* beats to do the next one.
 - Stops when there are no more events (or *delta* is nil).

VI. Operations on Patterns

Math operators

Patterns respond to most math operators lazily.

- Math on a pattern *makes another pattern* that will do the operation when asked, later.
- The new pattern can be used anywhere another pattern could.
 - The operands should yield values compatible with math operators.

```
(Pwhite(1, 5, inf) * 2).dump;
```

```
Instance of Pbinop { (03083B60, gc=9C, fmt=00, flg=00, set=02)
  instance variables [4]
    operator : Symbol '*'
    a : instance of Pwhite (03083B20, size=3, set=2)
    b : Integer 2
    adverb : nil
}
```

Very useful: `.x` operator adverb. Here, it transposes (+) a major-7th chord to a random root. Pwhite to the left is the root, evaluated once per *complete run* through the Pseq.

```
(
  p = Pbind(
    \midnote, Pwhite(48, 72, inf) +.x Pseq(#[0, 4, 7, 11], 1),
    \dur, 0.125
  ).play;
)

p.stop;
```

Pkey: Calculations within Pbind

Pbind evaluates each named stream *in order*. So, you can look back to previously calculated values by the **Pkey** event-lookup pattern.

```
// something simple - the higher the note, the shorter the length
(
  p = Pbind(
    \degree, Pseq([Pseries(-7, 1, 14), Pseries(7, -1, 14)]), inf),
    \dur, 0.25,
    // \degree is EARLIER in the Pbind
    \legato, Pkey(\degree).linexp(-7, 7, 2.0, 0.05)
  ).play;
)

p.stop;
```

Can Pkey be used inside other patterns?

- Yes, should be OK in general.

```
p = Pbind(
  \lowBound, Pseries(0.0, 0.01, 100),
  \highBound, Pgeom(1.0, 1.05, 100),
  \random, Pwhite(Pkey(\lowBound), Pkey(\highBound), inf)
);

p.collect(_.random).asStream.nextN(100, {}).plot;
```

Changing Pbind child patterns on the fly

The value patterns in a Pbind are *fixed in stone* once **asStream** makes the stream. How can you replace them while it's playing, then?

- Use a *proxy*: an object that stands in for a pattern.
- The **PatternProxy** used in Pbind remains the same object.
- The pattern to which the proxy refers (its *source*) can change anytime.

```
(
~degree = PatternProxy(Pn(Pseries(0, 1, 8), inf));
~dur = PatternProxy(Pn(0.25, inf));

p = Pbind(
  \degree, ~degree,
  \dur, ~dur
).play;
)

~degree.source = (Pexprand(1, 8, inf) - 1).round;
~dur.source = Pwrand(#[0.25, 0.5, 0.75], #[0.5, 0.3, 0.2], inf);

p.stop;
```

Pdefn and **Pdef** are global repositories of (respectively) **PatternProxies** and **EventPatternProxies** that allow easier access, without using variables.

```
(
Pdefn(\degree, Pn(Pseries(0, 1, 8), inf));
Pdefn(\dur, Pn(0.25, inf));

Pdef(\flexible, Pbind(
  \degree, Pdefn(\degree),
  \dur, Pdefn(\dur)
)).play;
)

Pdefn(\degree, (Pexprand(1, 8, inf) - 1).round);
Pdefn(\dur, Pwrand(#[0.25, 0.5, 0.75], #[0.5, 0.3, 0.2], inf));
Pdef(\flexible).stop;
```

VII. Filter patterns

A filter pattern modifies the resulting stream in some way.

There are a lot of them! See the *Practical Guide* for the ones we don't discuss today.

Repetition and Constraint Filters

Repetition filters repeat a whole pattern (**Pn**), or individual values/events (**Pstutter**, **Pclutch**).

```

// play repeated notes with a different rhythmic value per new pitch
// using Pstutter
p = Pbind(
    // making 'n' a separate stream so that degree and dur can share it
    \n, Pwhite(3, 10, inf),
    \degree, Pstutter(Pkey(\n), Pwhite(-4, 11, inf)),
    \dur, Pstutter(Pkey(\n), Pwhite(0.1, 0.4, inf)),
    \legato, 0.3
).play;

p.stop;

```

```

// using Pclutch
// the rule is, when degree changes, dur should change also
// if Pdiff returns 0, degree has not changed
// and Pclutch prevents a new Pwhite value from coming through
p = Pbind(
    \degree, Pstutter(Pwhite(3, 10, inf), Pwhite(-4, 11, inf)),
    \dur, Pclutch(Pwhite(0.1, 0.4, inf), Pdiff(Pkey(\degree)).abs > 0),
    \legato, 0.3
).play;

p.stop;

```

Constraint filters stop a pattern early. One code block can create a pattern for general behavior, and another can control how long it runs.

- Can go by number of items (**Pfin**), sum of numbers (**Pconst**) or total duration (**Pfindur**).

```

// Two variants on the same thing
// Use Pconst or Pfindur to create 4-beat segments with randomized rhythm
// Pconst and Pfindur both can ensure the total rhythm doesn't go above 4.0

```

```

p = Pn(Pbind(
    // always a low C on the downbeat
    \degree, Pseq([-7, Pwhite(0, 11, inf)], 1),
    \dur, Pconst(4), Pwhite(1, 4, inf) * 0.25
), inf).play;

```

```
p.stop;
```

```

p = Pn(Pfindur(4), Pbind(
    \degree, Pseq([-7, Pwhite(0, 11, inf)], 1),
    \dur, Pwhite(1, 4, inf) * 0.25
)), inf).play;

```

```
p.stop;
```

Merging patterns (“Pattern composition”)

Modularize pattern construction

- One block generates rhythm pattern, another generates pitches or other parameters
- Pattern composition combines them into one result event stream

- **Pchain**, or composition operator <>
- Patterns are chained from right to left, like mathematical notation $f \cdot g = f(g(x))$

```
~rhythm = Pbind(
  \dur, Pwrand(#[0.125, 0.25, 0.5], #[0.3, 0.5, 0.2], inf),
  \legato, Pwrand(#[0.1, 0.6, 1.01], #[0.1, 0.3, 0.6], inf)
);
~melody = Pbind(
  \degree, Pwhite(-4, 11, inf)
);

// Calculates ~rhythm first, then overlays ~melody on top
p = Pchain(~melody, ~rhythm).play;
p.stop;
```

Possible to swap melodies and rhythms independently using **EventPatternProxy**.

```
~rhythm = EventPatternProxy(Pbind(
  \dur, Pwrand(#[0.125, 0.25, 0.5], #[0.3, 0.5, 0.2], inf),
  \legato, Pwrand(#[0.1, 0.6, 1.01], #[0.1, 0.3, 0.6], inf)
));

~melody = EventPatternProxy(Pbind(
  \degree, Pwhite(-4, 11, inf)
));

p = Pchain(~melody, ~rhythm).play;

~melody.source = PmonoArtic(\default, \degree, Pseries(4, Prand(#[-1, 1], inf), inf).fold(-4, 11));

~melody.source = Pbind(\degree, Pseries(4, Pwrand(#[-2, -1, 1, 2], #[0.3, 0.2, 0.2, 0.3], inf), inf).fold(-4, 11));

p.stop;
```

Running patterns in parallel

The easiest way to run patterns in parallel is to play them separately. The clock handles parallelism.

Sometimes you need to have a single pattern object that runs several event streams concurrently.

- **Ppar**: Takes a preset list of patterns.
- **Ptpar**: Takes a preset list of [time, pattern, time, pattern...]. Each pattern is delayed by its time. (Example later)
- **Pspawner**: Uses a Routine-style function to spawn parallel patterns dynamically. Ron will talk about this.

Effects, bus and group management

Isolate patterns' activity on separate buses, possibly groups to control order of execution, and run effect synths after the source synths.

- **Pgroup**: Run the enclosed pattern in its own group, on its own audio bus.
- **Pfxb**: Run the pattern and an effect synth in its own group, on its own audio bus.
 - **Pfx**: Run the pattern and an effect synth with a given group and bus.
 - **Pfxb(Pfx(...))** chains effects.

```

// Demonstrates how Pfxb isolates signals on different buses
// The fx synth is a simple volume control here
// but it could be more complex

(
SynthDef(\volumeCtl, { lout, amp = 1, gate = 1
  var sig = In.ar(out, 2) * amp;
  sig = sig * EnvGen.kr(Env#[1, 1, 0], #[1, 1], -3, releaseNode: 1), gate, doneAction:
2);
  ReplaceOut.ar(out, sig)
}).memStore;

~vbus1 = Bus.control(s, 1).set(0.5);
~vbus2 = Bus.control(s, 1).set(0.5);

~window = GUI.window.new("mixers", Rect(10, 100, 320, 60));
~window.view.decorator = FlowLayout(~window.view.bounds, 2@2);
EZSlider(~window, 310@20, "low part", \amp, { lez| ~vbus1.set(ez.value) }, 0.5);
~window.view.decorator.nextLine;
EZSlider(~window, 310@20, "high part", \amp, { lez| ~vbus2.set(ez.value) }, 0.5);
~window.front.onClose_({ ~vbus1.free; ~vbus2.free });
)

(
p = Ppar([
  Pfxb(Pbind(
    \degree, Pseq([0, 7, 4, 3, 9, 5, 1, 4], inf),
    \octave, 4,
    \dur, 0.5
  ), \volumeCtl, \amp, ~vbus1.asMap), // map to control bus here
  Pfxb(Pbind(
    \degree, Pwhite(0, 11, inf),
    \dur, 0.25
  ), \volumeCtl, \amp, ~vbus2.asMap) // ... and here
]).play;
)

p.stop;

```

VIII. Embedding imperative code into patterns

What about jobs for which there isn't a pre-written pattern?

- **Pfunc**: Each 'next' call gets the given function's return value.
- **Pfuncn**: Like Pfunc, but do it only 'n' times.

```

// index 12-tone matrix by Pfunc
~row = ((~row = (0..11).scramble) - ~row.first) % 12;
~inversion = ~row.neg % 12;
~matrix = ~inversion.collect { |root| (~row + root) % 12 };

p = Pbind(
  \instrument, \default,
  \index, Pn(Pseries(0, 1, 12), inf),
  \rowtype, Pstutter(Pwhite(8, 12, inf), Pwhite(0, 3, inf)),

```

```

\note, Pswitch1([
    // normal
    Pfunc { |ev| ~matrix[0][ev[\index]] },
    // inversion
    Pfunc { |ev| ~matrix[ev[\index]][0] },
    // retrograde
    Pfunc { |ev| ~matrix[0][11 - ev[\index]] },
    // retrograde inversion
    Pfunc { |ev| ~matrix[11 - ev[\index]][0] },
], Pkey(\rowtype)),
\dur, 0.25
).play;

p.stop;

```

- **Proutine:** Write a custom routine that returns values by `.embedInStream`.
 - `embedInStream` allows patterns to be embedded too, like subroutines.
 - Special handling for *inval*: `inval = return.embedInStream(inval)`

```

// Two-dimensional random walk, with GUI
(
~row = ((~row = (0..11).scramble) - ~row.first) % 12;
~inversion = ~row.neg % 12;
~matrix = ~inversion.collect { |root| (~row + root) % 12 };

~window = ResizeFlowWindow("magic square", Window.screenBounds);
~window.view.background_(Color.gray(0.8));
~textBounds = Rect(0, 0, 36, 36);
~textBk = Color.gray(0.3);
~textC = Color.white;
~textHL = Color.new255(49, 106, 126);
~font = Font("Helvetica", 18);
~texts = ~matrix.collect({ |row|
    var    guiRow = row.collect({ |cell|
        StaticText(~window, ~textBounds)
            .string_("#["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A",
"A#", "B"])[cell])
            .font_(~font)
            .background_(~textBk)
            .stringColor_(~textC)
            .align_(\center)
    });
~window.startRow;
guiRow
});
~window.recursiveResize.front;
)

(
p = Pbind(
    // how many steps to go in this direction?
    \n, Pwhite(4, 10, inf),
    \xStep, Pstutter(Pkey(\n), Pwrand(#[0, -1, 1], #[0.5, 0.25, 0.25], inf)),
    // nonzero only if xStep is 0

```

```

\yStep, Pstutter(Pkey(\n), Pif(Pkey(\xStep).abs > 0, 0, Prand(#[-1, 1], inf))),
\note, Proutine{ linEvent
  var    x = 12.rand, y = 12.rand, oldEvent,
         x1, y1;
  loop {
    oldEvent = inEvent;
    { ~texts[y][x].background = ~textHL }.defer(s.latency);
    inEvent = ~matrix[y][x].embedInStream(inEvent);
    x1 = x; y1 = y;
    { ~texts[y1][x1].background = ~textBk }.defer(s.latency);
    x = (x + oldEvent[\xStep]) % 12;
    y = (y + oldEvent[\yStep]) % 12;
  }
},
\noteDiff, Pdiff(Pkey(\note)),
\octave, Paccum(2, 8, step: Pif(Pkey(\noteDiff).abs > 6,
  Pkey(\noteDiff).sign.neg,
  0
), length: inf, start: 5),
\dur, 0.125
).play;
)

p.stop;

```

- **Plazy**: Like **Pfunc**, except **Pfunc** yields the function's return value directly. **Plazy** embeds the return instead. Use **Plazy** to break up pattern logic into manageable sizes.
 - The rhythm example in [\[PG Cookbook07 Rhythmic Variations\]](#) uses **Plazy** to compute a whole bar of rhythm at once, then play it out in time.

IX. Conclusions

Patterns are a different way of writing: *describing behavior* rather than always giving orders.

- One plus: Possible to reimplement patterns in other languages—same benefit as the Object-Oriented Programming distinction between *interface* and *implementation*.

“Connective tissue” doesn't have to be stated repeatedly: *Event types* add convenience.

It takes some time to get used to thinking non-imperatively.

- Start with simple things and grow into more complicated cases.