Synthesizer

A Pattern Language for Designing Digital Modular Synthesis Software

Thomas V. Judkins and Christopher D. Gill
Washington University, St. Louis, MO
November 15, 2000

Introduction

Synthesizer is a pattern language for designing digital synthesizers using modular synthesis in software to generate sound. Software developed according to this pattern language emulates the abilities of an analog synthesizer.

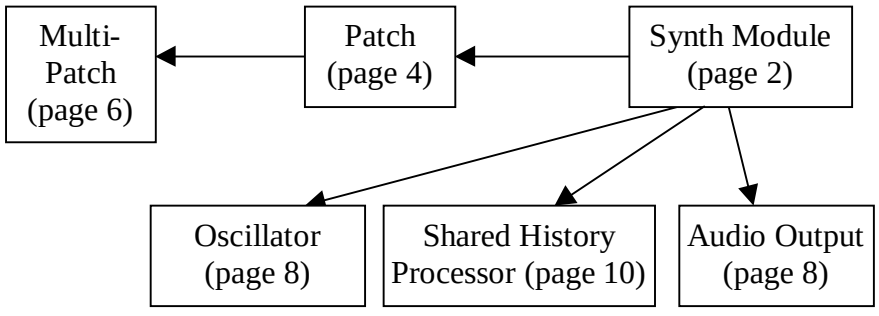
Modular synthesis is one of the oldest sound synthesis techniques. It was used in the earliest analog synthesizers, like the Moog [1] and ARP [2]. These machines introduced the oscillator-filter-amplifier paradigm, where sound generated by an oscillator is passed through a series of filters and amplifiers before being sent to a speaker. These first machines had physical modules through which electrical signals were passed. These modules can be emulated in software, and the Synthesizer pattern language captures the software design patterns embodied in this approach.

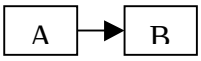
Context

A Digital audio signal is represented as a series of samples. The two factors determining the fidelity of the signal are the rate at which samples are produced, and the precision of each sample. High frequency signals change their values very quickly, and depend on high sample rates to retain fidelity, while low frequency signals may retain fidelity at lower sample rates. A sample is a numeric value: in this paper, a floating point number.

In this paper, each module will be represented by an object. We will use the term provider to describe a module providing a signal to another module, and the term requester to describe a module being provided with a signal. Most modules will both provide and request signals. Provider modules are said to input a signal to any one of a requester's channels

Pattern Map



Legend: consider applying A, then B 

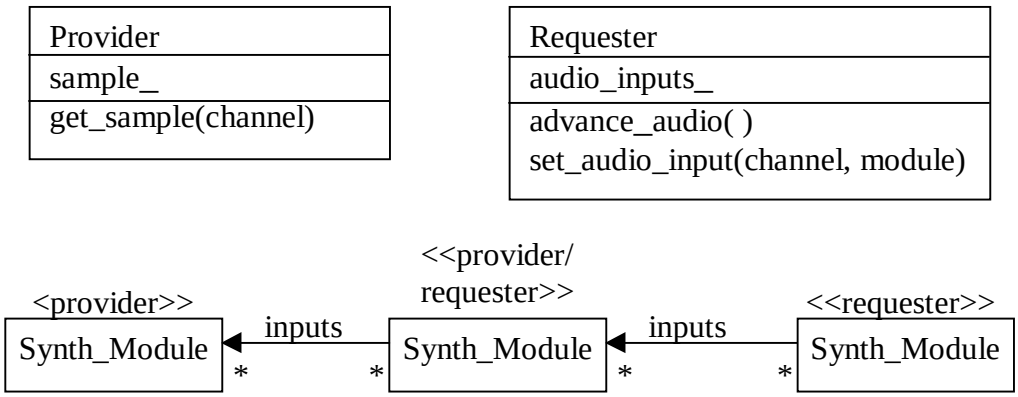
Synth Module

Problem:
Need to define a generic system of interaction between individual synthesizer modules.

Forces:

- Modules will need to pass samples to one another
- Number and type of provider modules not known in advance
- All modules must be synchronized to one common sample rate
- There is potential for feedback between modules
- Some signals will require a high sample rate, while others will not

Solution:



Therefore, allow modules to exchange information by keeping references to all provider modules within the requester module. Keep references in an expandable data structure such as a linked list, letting each data member correspond to one channel. Individual modules can then hold an arbitrary number of channels, querying each channel with a consistent method call. This allows individual modules to determine the use of each channel (i.e., channel 1: volume; channel 2: frequency; etc.).

A simple way to obtain synchronization would be to utilize the GoF Chain of Responsibility pattern [3], requesting a sample from last module in a chain, and allowing the request to trickle down the hierarchy. However, this would prevent feedback loops, which are necessary to certain effects such as delay. It also presents a problem when modules do not request samples from all their provider modules on every sample. For example, an oscillator produces waveforms at a certain frequency. If a request is only made every tenth sample, however, the oscillator will think only one sample has passed when there have actually been ten. The frequency produced by this oscillator will then be ten times slower than desired. Therefore, all modules must be notified exactly once during every sample.

Therefore, provide both a means of advancing the state of a module, and a means of requesting a sample from a module. All modules will be advanced on every sample, thus solving the issue of synchronization. The actual sample value will be computed only when the sample is requested of a module, allowing an oscillator that is only queried every ten samples to only compute one sample every ten samples, but still producing the correct frequency.

Thus, any module will go through two stages each period: the advancing state stage, and the requesting sample stage. Advancing the state of a requester module will trigger sample requests of that module's providers. This is similar to the Electrical Engineering concept of TTL flip-flops [4] in electronic hardware systems, which reads inputs on the rising edge of a clock. The Synth Modules, however, cannot all be clocked simultaneously, so a provider module may or may not have been advanced during the current sample. This produces a delay of a few samples when the requester modules are advanced before their provider modules.

When the provider modules are advanced before their sample is requested, however, there is no delay. This does not present a problem however, because we can control the order in which modules are advanced. Feedback loops therefore present the only delay because, due to their circular nature, it is impossible to advance every provider module before all of its requester modules. This small delay is acceptable, as there is naturally a small delay in analog feedback loops as well.

To reduce the number of requests made of modules, provide different methods for advancing state. This allows modules to check infrequently changing information (like user input) less frequently than audio information. For example, modules may be told to advance state 44,100 times per second, but may only be told to look at user information 200 times per second. An alternative to this is to let the individual modules keep track of the number of samples that have gone by, and only have them query certain in ports every so many samples. This, however, puts more of a burden on the individual modules, and since most modules will have to query at similar rates, one module may keep track of the rates instead of all modules. Putting rates in a central location also makes them easier to change. Now only one module needs to be altered instead of all of them.

Implementation Notes:
For example, the Soft-Synth framework implements Synth Module as a base class that all modules extend. The class allows for three types of in ports: audio in ports, to which all audio signals are passed, and are advanced on every iteration; control in ports, which are passed lower frequency control signals; and gate in ports, which are passed user generated data such as key presses, knob turns, or MIDI data. The base class provides concrete methods for setting the inputs to each type of port.

```
void set_audio  (int channel, Synth_Module &m, int m_out, double m_vol)
{
       audio_inputs_  ->insert(channel, m, m_out, m_vol);
}
```

This method adds a signal to the specified audio input channel. The input signal is specified by providing a Synth_Module, m, and m's output channel. The double m_vol adjusts the level of the signal received from m. When multiple inputs to a single channel[1] are specified, both signals are added together, and the sum becomes the input of that channel.

The base class also provides virtual methods for advancing each set of inputs. A control input is only queried when the `advance_control()` method is called, and the same is true of audio and gate inputs. There is also a method for retrieving sample values when a module is queried. This

[1] As we will see in the next section, a module may produce more than one output channel, so both m and m's output channel are needed here.

method computes and returns the sample value. In case a module is queried by two requesters, the method also stores the sample value to avoid repeat computation.

```
virtual void advance_audio() {};
virtual void advance_control() {};
virtual void advance_gate() {};
virtual double get_sample (int out_channel) {return sample_;};
```

The `get_sample (int channel)` method takes an integer parameter in case a module provides more than one output channel. The base class Synth_Module implements this method to return a number set in one of Synth_Module's constructors. This is important if a user wishes a constant value to be input to a module. The method is virtual so that child classes will override it.

The Soft-Synth framework also provides a class called Patch to advance state of Synth Modules.

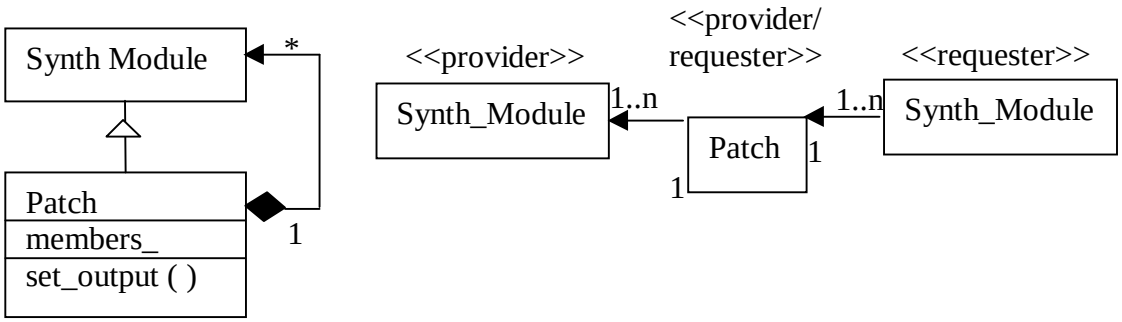
Patch

Problem:
Need to allow users to abstract away details of multiple modules and create compound modules.

Forces:

- Must synchronize internal modules with outside environment
- Must appear to be a single module to provide end users with a higher level of abstraction
- Modules outside the Patch must have a way of passing values to and from internal modules without breaking encapsulation
- Must be reusable and recallable

Solution:



Therefore, provide a Patch module to combine and synchronize other modules.

To address the issue of synchronization, Patch uses the Chain of Responsibility pattern [3]. Every time a Patch module is advanced, that Patch module advances each module contained within it. This assures that the states of all modules contained within a Patch object are advanced exactly once per sample. It also assures that Patches are advanced once every time the external environment is advanced. This invariant could break down when Patch objects circularly contain one another (e.g., Patch A contains Patch B, and Patch B contains Patch A). However, this would produce an infinite string of modules connected to one another, and thus should not be allowed.

To allow signals to be passed into a Patch object, have external modules referenced by the Patch module, so internal modules can query the Patch object to receive signals. The Patch module will

then access the modules outside the encapsulation barrier and return the value to the internal module. External modules will access internal modules in a similar way: internal modules will register as outputs with the Patch object, and when an external access request is made of a Patch object, it will compute and return the value of the internal module(s) corresponding to the output.

Finally, the Patch objects must be able to be reused arbitrarily; for example, a user may wish to use a program in response to a keyboard. Every different note played must be represented by its own set of oscillators, filters, etc. Several copies of a Patch object must exist in order to handle multiple notes. Also, programs in traditional analog synthesizers were difficult to recall (or remember), so having a way of recalling a Patch's settings would be beneficial.

Therefore, provide a way of externalizing the state of a Patch object in a file. Every time the Patch needs to be duplicated or recalled, only the name of the file representing the Patch's state must be referenced.

Implementation Notes:
In the Soft-Synth framework, the Patch class extends Synth_Module, thereby providing an interface identical to any other module. This allows users to think of patch as a single object; inputs can be assigned to just as any other module, and any module can request a sample from a Patch object.

The class also provides a method `add_member`, which adds a module to the Patch. To ensure synchronization, the method only adds the module if it is not already contained in the Patch. Users must ensure that modules passed to this method are not contained in any other Patch object.

```
int Patch::add_member(Synth_Module &m){
	if (members_.is_member (m)){	// trying to add member twice
		return 0;
	}else{
		members_.add (m);
		return 1;
	}
}
```

When a Patch's advance state call is made, it calls the same method on each member. The `advance_audio` method is shown below; `advance_gate` and `advance_control` are similar.

```
void Patch::advance_audio(){
	for(Synth_Module *temp=members_; 0!=temp; temp=temp->get_next_module())
	{
		temp->get_module().advance_audio();
	}
}
```

A method to set the output of a Patch is also included. The `set_output` method takes an internal module, and adds it to a list of outputs.

```
int Patch::set_output(int patch_out, Synth_Module &m, int m_out, double m_vol){
	if (is_member (m)){
		outputs_->insert(patch_out, m, m_out, m_vol);
		return 1;
	}else{
		return 0;
	}
}
```

A problem arises, however, in the fact that there will now be two sets of modules requesting samples from a Patch object: internal modules requesting inputs, and external modules requesting outputs. There is, however, only one `get_sample` method. This problem was solved by holding negative channel numbers for all internal modules requesting input signals. An alternative to this would have been to check whether the requester module was a member of the Patch. This, however, would have been time-consuming, and would have required a reference to the calling object to be passed to each `get_sample` call. So instead, a negative parameter returns input to internal modules, while a positive parameter would returns output to external modules. A call from an internal module with a positive parameter should not happen, but would not break any abstraction barriers, as the output modules are already accessible by the module.

```
double Patch::get_sample(int out_channel)
{
	if (out_channel > 0){
		return outputs_->mix(out_channel);
	}else if (out_channel < 0){
		return audio_inputs_->mix(-out_channel);
	}
	return 0;
}
```

The program Reaktor creates objects called ‘macros’ that are similar to Synthesizer’s Patch objects [5].

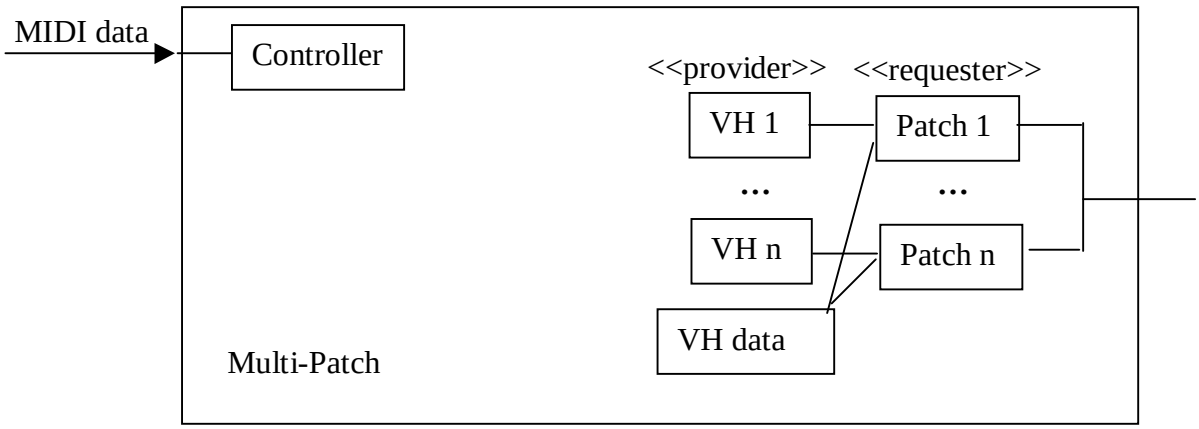
Multi-Patch

Problem:
Users will eventually wish to control their programs by passing it MIDI information.

Forces:

- MIDI presents data as periodic events, but modules expect digital audio signals
- Each Patch can represent one voice or note, but the maximum number of voices is not set or known
- Notes must sound with little delay after a note is pressed
- Many different types of MIDI data may be used to represent operations

Solution:



Therefore, provide an object called a Multi-Patch that handles interaction between user and program. Separate notes that are being played simultaneously must produce separate frequencies, so each note must be represented by a separate set of modules. Therefore, in every Multi-Patch, include a number of identical Patch objects so that each of these may play a separate note.

There are two possible ways to assign notes to Patch objects. The first is to create a new Patch object every time a new note is pressed. The second is to create a pool of Patches, and assign each note to an existing object. The first approach allows for an expandable number of notes that at all times matches the load applied by the user. This creation of new Patches can be time consuming, which could lead to a noticeable delay between the time the note is pressed to the time the note is heard. It also may consume memory or processor load to a point where the program will not run properly.

The second approach limits the number of notes that can be played to a set number, and the Multi-Patch must begin to drop notes when that limit is exceeded.[2] The Patch objects are already created, however, so no additional delay is incurred between the pressing and sounding of a note. Also, delay time will not degrade because of an increasing number of Patch objects in use. The main drawback to this approach is that notes must be cut off prematurely when the limit is exceeded.

Both are viable solutions, but we chose the latter because of the reduced delay. This issue is similar to the use of Thread Pool vs. Thread-Per-Request [6]. Thus, when more notes are played than the synthesizer can produce (in our case, the number of Patch objects in use), one note must be replaced. The criteria used to decide which voice to interrupt is called a voice replacement scheme. These schemes may be random, or based on criteria such as lowest pitch, least recently played, or most recently played. Whatever strategy is chosen, this Pattern will decide the replaced note before it is needed, and keep track of it in case it is needed. This may take additional computing time, but there will again be no additional delay when the note is struck.

Another problem arises in converting the MIDI information sent to the Multi-Patch into a stream of samples that can be understood by the Patches. The first part of this problem is the fact that MIDI represents notes by numbers, 0-127, and our objects must use a frequency instead of these numbers. Therefore, include a lookup table inside our Multi-Patches that will convert these MIDI note numbers into their frequency equivalents. This not only removes this task from individual modules, but also allows for different tuning schemes to be utilized. This approach does, however, eliminate the ability to manipulate the MIDI data being passed into the program.

The second part of this problem resides in the fact that MIDI data are not persistent events. In other words, when a note is played, a signal is not held high until the note is released, instead, a packet is sent informing of a note being pressed, then another packet is sent when the note is released. Therefore, provide a voice handler class that will hold values such as note pressed and frequency. This voice handler will then be referenced by the individual Patch objects like any other module would be. Each Patch will have one voice handler object which passes signals relevant to only that one Patch module, as well as one voice handler that will present signals relevant to all Patch objects.

[2] This is in fact no different from traditional instruments such as the guitar, which limits the musician to six notes. The piano is a bit less limiting with eighty-eight possible notes.

There are, however, many different types of MIDI packets that may be sent. So, instead of implementing a channel for every possible type of packet, only implement a channel for the common messages, and then allow the user to program separate messages over specific channels.

Audio Output

Problem:
Audio signals must eventually be passed out of the program in one form or another.

Forces:
- Users may wish to send output to a file or soundcard
- Soundcards usually have multiple output channels (left and right)
- Most systems will have interrupts, which will slow sown the processing of some samples, but samples still must be produced at constant rate

Solution:
Therefore, contain all a program's Synth Modules within an Audio Output object. The object will be responsible for transferring data from the modules to the destination (a soundcard, file, or other program). The single object will also handle multiple signals to handle stereo or multi-channel outputs.

What's more, the object will handle advancing modules in a program. It will hold a buffer to prevent lost samples due to interrupts. It will also be capable of reducing the sample rate in case of processor overload.

Implementation Notes:
This portion was implemented in assembly language. The code basically creates two buffers in memory from which the sound card reads. The object writes samples to the first buffer while the sound card is reading from the second. It continues to write samples until the buffer is full. When the sound card is finished reading the second buffer, the Audio Output object repeats the process on the second buffer.

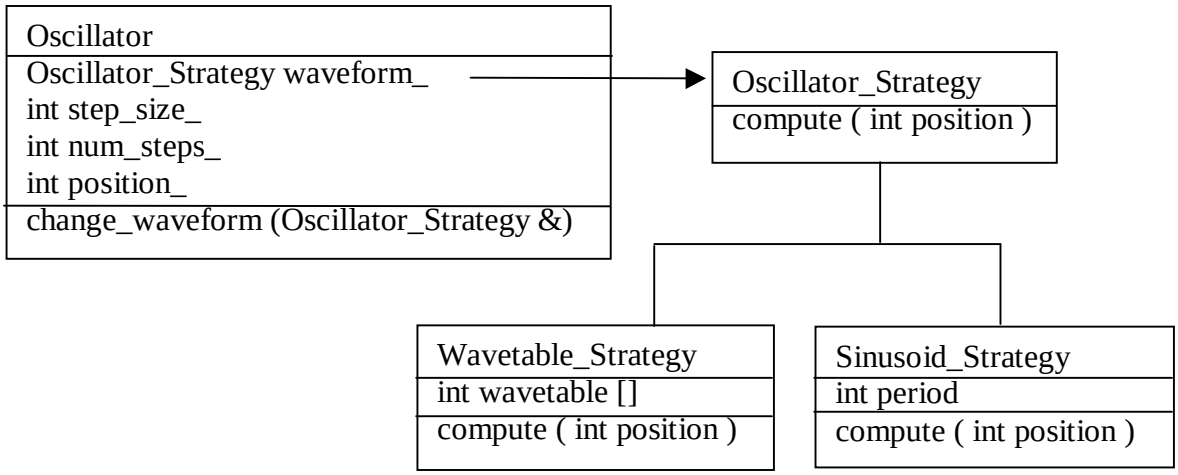
Oscillator

Problem:
The most common way of producing signals is through general waveforms repeated at some frequency.

Forces:
- Many different wave forms will be needed (either generated mathematically or by table lookup), but all will be generated similarly
- Wave forms may be switched arbitrarily, and might be different in form
- Will be used differently LFO[3] vs. Oscillator
- Needs to be fast / won't be accessed every sample

[3]Low Frequency Oscillator (LFO): These objects are identical to regular oscillators, except for the fact that they generate signals which are several octaves lower, and thus need not be processed at such a high sample rate.

Solution:



Therefore, abstract away the actual shape of the wave forms by encapsulating the shapes in a separate Strategy object. The Oscillator class then can focus solely on frequency and progression through a wave form. When actual samples are needed, the Oscillator object may provide it's Strategy object with the progression through the current wave to retrieve actual sample values. This allows frequency, amplitude, and waveform to be changed at any time, even in the middle of a waveform.

Also, the strategies used by the Oscillators may be quite abundant. They will, however, all be stateless objects. Therefore, it is natural to implement them all as flyweights. This reduces the amount of memory used by the strategies, and makes it easier to

Method calls to the Strategy object will be needed when a `get_sample` request is made of the oscillator. This allows oscillators being used only as control inputs (low frequency oscillators) to use less computation time than a regular oscillator.

Implementation Notes:
The Soft-Synth framework implements the Oscillator as a sub-class of the Synth-Module class. It implements both Amplitude and Frequency Modulation. The class takes two channels of gate inputs and two channels of control inputs. Oscillators take no audio input signals.

The first gate input specifies the frequency of oscillation in Hertz. This value will most likely be set by some form of user input, and thus will not change very often. The second gate input is a wave-sync that will reset the oscillator to the beginning of it's waveform when the signal is non-zero. This is useful when using an Oscillator as an LFO, and resetting the waveform in response to a key press.

```
void Oscillator::advance_gate(){
       frequency_ = gate_inputs_->mix(1);
       if (gate_inputs_->mix(2))
       {
              position_ = 0;
       }
       advance_control();
}
```

The first control input specifies frequency modulation, where a positive value will increase the frequency, and a negative one will decrease it. The second control input specifies amplitude modulation. It is simply a constant that the output signal will be multiplied by. The variable `step_size_` is then calculated based on the value gotten from the FM input. This variable corresponds to the decimal to be added to `position_` every time the Oscillator is advanced.

```
void Oscillator::advance_control ()
{
	double FM_freq = control_inputs_->mix(1);
	double temp_freq;
	if (FM_freq > 0)
	{
		temp_freq = ((FM_freq / MAX_VALUE) + 1)* frequency_;
	}
	else
	{
		temp_freq = ((FM_freq / MAX_VALUE) + 1)* frequency_;
	}
	step_size_ = temp_freq / (SAMPLE_RATE);
	amplitude_ = control_inputs_->mix(2) / MAX_VALUE;
	advance_audio();
}
```

Another approach taken by the Reaktor [5] program is to provide separate modules for both LFO's and Oscillators. Also differentiating between waveforms generated by mathematical formula and waveform stored in wavetables. Reaktor also allows switching between different waveforms by multiplexing between several existing oscillators, in effect shutting off processing for unused oscillators.

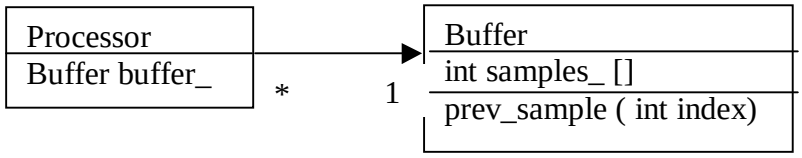
Shared History Processor

Problem:
Modules may have to keep track of previous samples.

Forces:

- Sample values must be recorded every time the sample is advanced
- Two modules may be keeping track of the same samples
- Number of samples to be kept track of may vary

Solution:



Therefore, provide a buffer class that will keep track of samples, and interface with processing modules that need these values. The buffer class will act as a requester module, requesting samples from any other module, but will define a separate interface for passing recorded values to the processor. This buffer will be used by an arbitrary number of processing modules, thus eliminating the need for each processor module to keep track of the same sample values.

For example, reverberation is produced by applying multiple delay effects to an audio signal. Individual delay modules will receive a sample, and then play it back a fixed period later. Were each delay unit to keep track of it's own internal buffer, data would be repeated in each delay unit. However, implementing the delay units as Shared History Processors allows the buffer to be kept in a shared external buffer, thus reducing memory use.

The buffer must, however, keep track of the samples that will be needed by the processors. This could be problematic because, as in the previous example, it is unlikely that all delay units will be set to the same delay length; some may request a sample recorded 100 samples earlier, while another may request one recorded 1000 samples earlier.

One possible solution is to have each processor specify to the buffer the number of samples it will need it to keep track of. This, however, would require that the processor not only know how many samples it needs, but also inform the buffer of that number every time it changes. A better solution is to let the buffer decide how many samples to track based on its use by the processors. Thus, every time a processor requests a sample that is no longer held in the buffer, the buffer can increase the number of samples it holds.

Another problem raised by creating an external buffer is the question of which processor owns it. In the case of internal buffers, each processor would own it's own buffer, however, with an external buffer, processors cannot automatically destroy their buffers. For this reason, it is easiest to allow the Patch object to claim responsibility for both buffer destruction, and advancing the state of the buffer.

Acknowledgements

The authors are grateful to our pattern shepherd, Norm Kerth, for guiding us in the months leading up to the PLoP 2000 conference. Norm has provided many valuable comments, questions, and suggestions that have shaped the evolution of both this paper and the pattern language it describes.

We also wish to thank David L. Levine, Director of the Center for Distributed Object Computing, in the Department of Computer Science at Washington University in St. Louis, for his ongoing support of this work.

Finally, the first author would like to thank Rich O'Donnell, Professor of Music at Washington University, and lead percussionist for the St. Louis Symphony Orchestra, for introducing him to sound synthesis.

References

[1] Electronic Musician magazine, August 2000, vol.16, no. 8
[2] Owner's Manual, the ARP Electronic Music Synthesizer Series 2600, 1971, Tonus Inc.
[3] Gamma, Helm, Johnson, and Vlissides, "Design Patterns", 1995, Addison Wesley
[4] Breeding, "Digital Design Fundamentals", 2nd. Ed., 1992, Prentice-Hall, NJ, pp. 217-223.
[5] Benutzerhandbuch, Reaktor Series version 2.3, 1997-2000 Alle Rechte vorbehalten, Berlin
[6] Douglas C. Schmidt, "Evaluating Architectures for Multi-threaded CORBA Object Request Brokers", Communications of the ACM, Vol. 41, No. 10, October 1998.