1. 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\textsuperscript{1} and ARP\textsuperscript{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.

2. The Patterns

The concise nature of Synthesizer is perhaps its greatest strength. As Alexander observes,

\begin{quote}
Even though the rules are simple, by the time you have twenty, perhaps fifty rules like this in your mind, it takes almost inhuman singleness of purpose to insist on them – not to let go of them.\textsuperscript{3}
\end{quote}

Therefore, Synthesizer is intended to allow developers to express software designs for a wide variety of good synthesizers, using a manageable number of patterns.

The remainder of this section presents each of the twelve patterns in the Synthesizer pattern language: Synth Module, Generator, Audio In, Oscillator, Sequencer, ADSR Module, Sampler, Processor, Shared History Processor, Patch, Noise Generator, and Synth Output. Each pattern describes the patterns that must be applied before it, and the other patterns with which it may be used, providing a generative language for designing digital modular synthesis software.

\textsuperscript{1} Named for its creator, Bob Moog.
\textsuperscript{2} Owner’s Manual, the ARP Electronic Music Synthesizer Series 2600, 1971, Tonus Inc.
\textsuperscript{3} Christopher Alexander, “The Timeless Way of Building”, 1979, Oxford University Press, pp. 222
**2.1. Synth Module**

**Problem Statement:** Sound synthesis requires that sounds be created, modified, combined, and output. Modular synthesis further stipulates that signals must be accepted from any device.

**Forces:** Different kinds of signal processing for digital synthesis may produce different sounds that can then be combined to produce many new sound effects. Modular synthesis stresses the ability to input any signal, regardless of the signal’s source or contents.

**Solution:** Therefore, represent each module (e.g., an ADSR\(^4\) module) in the synthesizer as a polymorphic instance of a class derived from an abstract base class. Supply three types of inputs: Audio-in, Control-in, and Gate-in; let the specific modules declare number of inputs, and outputs. These inputs may all be sampled at different rates\(^5\). Allow multiple signals to be mapped to a single input by mixing the signals down to one channel\(^6\).

**Implementation Notes:** For example, all modules in the Soft-Synth\(^7\) framework derive from an abstract base class called SoftSynth_Module. The SoftSynth_Module class declares the abstract methods: `int advance_sample()`, and `int get_sample(int output_channel)`. The `advance_sample` method advances

---

\(^4\) ADSR stands for Attack Decay Sustain Release. It is a module that generates a pulse with certain attack, decay, sustain, and release values. Section 2.6 describes ADSR in more detail.

\(^5\) A digital audio signal is represented as a series of values (samples). Standard CD encoding samples at 44.1kHz. Control signals may be clocked at one-tenth speed, and gated signals at one-hundredth.

\(^6\) This may be done in digital audio by adding each signal’s sample value.

\(^7\) Soft-Synth is a modular synthesis component framework being developed by the first author at Washington University.
the module to the next sample in the output stream. The `get_sample` method returns the sample value associated with the specified `output_channel`.

The `SoftSynth_Module` class also has three concrete methods:

```c
void set_audio (SoftSynth_Module m, int out_channel,
               int volume, int in_channel);

void set_control (SoftSynth_Module m, int out_channel,
                 int volume, int in_channel);

void set_gate (SoftSynth_Module m, int out_channel,
               int volume, int in_channel);
```

These methods will add a `SoftSynth_Module` to one of several lists. Each list will represent a single input signal (channel) consisting of any number of `SoftSynth_Modules` (mixed down to a single signal when the sample value is needed). The specific module will determine the number of audio and control signals.

**Other Patterns:** Consider applying Generator, Processor, Patch, Synth Output and the patterns that specialize them to implement the different kinds of modules in a synthesizer design.

### 2.2. Generator

**Guard:** The design decisions raised in the Synth Module pattern should be considered before applying Generator.

**Problem Statement:** All signals in Modular synthesis must have a source. At the root of synthesis, sounds must be generated without audio signal input.

**Forces:** Sounds can be generated from a variety of sources. Encapsulating these diverse sources within consistent interfaces allows flexibility in composing modules, which in turn results in software that can produce a greater range of sounds.

**Solution:** Therefore, create an abstract class of generator modules, that all have the same property of expecting no audio inputs.

**Other Patterns:** Consider applying Audio In, Oscillator, Sequencer, Sampler, and ADSR Module to implement different kinds of Generators.
2.3. Audio In

Guard: The design decisions raised in the Synth Module and Generator patterns should be considered before applying Audio In.

Problem Statement: Sounds generated by various external devices need to be made available as inputs to the synthesizer.

Forces: Sounds from external devices must be treated consistently with each other and with other generated sounds, for greater flexibility in combining different sounds.

Solution: Therefore, provide a kind of Generator module, called an Audio In, for inputting an audio signal from either a file or from a hardware input line into the computer.

Notice that an Audio In does not actually generate sound; instead, it reads a signal from an external source. However, by incarnating that signal within the synthesizer framework, it is acting as a Generator. Thus, an Audio In should be considered a kind of Generator.

2.4. Oscillator

Guard: The design decisions raised in the Synth Module and Generator patterns should be considered before applying Oscillator.

Problem Statement: The most basic kind of sound generation begins with a waveform repeated regularly at a particular frequency. The frequency at which the waveform is repeated determines the pitch of the sound heard by the human ear. The shape of the waveform determines the timbre and intensity of the sound.

Forces: A repeating waveform may take on any one of an effectively infinite number of shapes. Providing a unifying abstraction that covers these myriad possibilities helps constrain this complexity for the synthesizer designer. Furthermore, for techniques such as frequency modulation, it is useful to abstract away the details of the waveform and modify the frequency itself in isolation. Finally, signals produced at audio frequencies are used to produce sounds, while signals at sub-audio frequencies are used as control inputs.

Solution: Therefore, provide a class that encapsulates the details of repetition, provides an interface for accessing and mutating the frequency of the signal, and defers details of waveform type. When creating an Oscillator, the user must specify the type of waveform and the frequency of repetition. This could mean including control inputs for frequency, frequency modulation, and amplitude.
The class (or its derived classes) must support two forms of oscillators: digital controlled oscillators (DCOs), and low-frequency oscillators (LFOs). DCOs generate waveforms at audio frequencies, thereby producing sound, whereas LFOs generate waveforms at sub-audio frequencies, producing a useful control signal for volume swells, vibratos, or the like. In hardware devices, discrete circuitry is used for the two forms, however, in a digital environment, the physical limitations are removed, and one oscillator may produce both sonic and sub-sonic frequencies.

Implementation Notes: The Soft-Synth framework provides two control input channels. The first determines the oscillation frequency (useful when controlling the frequency with a GUI slider, or MIDI). The second is responsible for frequency modulation: a positive control signal will increase the frequency, while a negative one will decrease it. Soft-Synth also includes classes responsible for encapsulating the waveform generating function, according to the GoF\textsuperscript{8} Strategy pattern.

The Strategy objects are passed a double-precision floating-point number representing the percentage of the wave completed, and return the next sample. There will be two types of Strategies. Strategies generating waveforms based on algorithms (i.e., sine-waves) will be stateless, suggesting they be implemented as GoF Flyweights to save space. Another type of Strategy class will hold a series of samples in a wave-table, and return the correct sample when queried. The differing characteristic for the sampled Strategy objects will be the means of determining the correct sample. Most current day synthesizers use only the second type of oscillator, emulating elementary waves through samples.

An alternative to using strategy objects for Oscillators is to create specialized child classes for all the types of Oscillators, according to the GoF Template Method pattern. However, this would not allow for an Oscillator to change its Strategy midway through its lifetime. Another alternative is to implement Oscillator using the GoF State pattern. This would, however, require repeated state checking, which is undesirable. Using the Strategy pattern therefore seems a natural choice. Meta-Synth and MaxDSP\textsuperscript{9} also appear to use Strategy with respect to Oscillators, because they both allow Oscillators to change their type during execution.

Native Instrument’s Reaktor\textsuperscript{10} puts each type of oscillator in its own class, allowing a ‘change’ in waveform by multiplexing between multiple oscillator objects. It also provides Oscillators with a third control input for amplitude, which is useful when passing an envelope in response to a key press.

Other Patterns: Consider applying Patch with Oscillator to implement a Noise Generator or any other complex repeated waveform. Fourier has shown that any sound can be constructed by composing sinusoids of differing phases, amplitudes, and frequencies.

\textsuperscript{8} I.e., a design pattern described in the “Gang of Four” book: Gamma, Helm, Johnson, and Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, 1995, Addison-Wesley
\textsuperscript{9} Scholz, Carter “Opcode MAX”, April 1991, Keyboard Magazine
\textsuperscript{10} Reaktor User’s Guide, 1999, Native Instruments Software Synthesis
Consider applying the GoF patterns Template Method, Strategy and Flyweight to implement different kinds of Oscillators.

### 2.5. Sequencer

**Guard:** The design decisions raised in the Synth Module and Generator patterns should be considered before applying Sequencer.

**Problem Statement:** It is often useful to generate discrete levels sequentially without interpolation. These levels are useful in controlling the pitch of an Oscillator, or the level of an amplifier. In this way, simple rhythms or even melodies may be generated.

**Forces:** The rate at which each level is output is arbitrary, and may be changing. The number of levels (states) is also arbitrary, and subject to change. When the last level is played, the order repeats.

**Solution:** Therefore, provide a sequencer class similar to a finite state machine that allows any number of states. Let the first gate signal represent the clock. Every time the signal goes positive, advance to the next sample. Allow the user to determine the number of states, and the value of each state in the Sequencer’s constructor (perhaps through control signals). Also provide methods changing number of states and value of each state. These methods could be called from a user interface to provide real-time ‘hands on’ manipulation of the sequencer.

For example, feed the output of a sequencer to the first control input of an oscillator. The signal outputted from the sequencer now controls the pitch of the oscillator. Program the sequencer in increasing steps and you will get an Atari-reminiscent arpeggio.

**Other Patterns:** Consider applying Sequencer to Sampler or Oscillator. Both Sampler and ADSR Module are also clocked by a control signal.

### 2.6. ADSR Module

**Guard:** The design decisions raised in the Synth Module and Generator patterns should be considered before applying ADSR Module.

**Problem Statement:** A family of useful sounds relies on an envelope\(^{11}\) characterized by attack, decay, sustain, and release (ADSR) phases. The attack phase describes the period where the waveform signal rises from its initial value to its maximum amplitude. A piano has a relatively short attack period (controlled by how hard one strikes the key), whereas a gong has a long attack phase. The decay phase describes the abrupt falloff after the at-

---

\(^{11}\) An envelope is a shape that changes as a function of time. The term is used here, as in the common terminology of electronic music, to mean a non-repeated (and therefore not audible) waveform that is used as a control signal.
tack as a signal reaches equilibrium, both the time it takes to fall, and the amount in which it falls. The sustain phase corresponds to how well a note can be held. A violin has a high sustain rate—the signal will not degrade until the player stops bowing. A piano, on the other hand, has a lower sustain—a piano note will eventually die even if the player does not release the note. Finally, the release phase describes the period in which a note is released: when a piano key is lifted, or a cymbal is muffled.

**Forces:** Transitions between ADSR phases are controlled by a gated signal. The rising edge of the control signal corresponds to the start of the attack phase, and the falling edge to the end of the sustain phase. The algorithms for determining the slopes of each phase range from straight lines to complex logarithmic functions.

**Solution:** Therefore, provide an ADSR Module that produces a non-repeating envelope that represents the volume of a note according to the ADSR characteristics described above. Input the gated control signal through the first gate channel. The first four control channels can determine the duration of each phase. For example, consider using an ADSR Module to control an amplifier.

### 2.7. Sampler

**Guard:** The design decisions raised in the Synth Module and Generator patterns should be considered before applying Sampler.

**Problem Statement:** In sound synthesis there are many sounds that could be introduced not by using internal generators but by capturing them in some external manner (i.e., recording one note from Yoyo Ma’s cello) with the intent of reproducing that sound at whatever pitch is desired.

**Forces:** Artists often want to reverse these externally captured sounds, loop them, shift their pitches, or repeat them.

**Solution:** Therefore, provide a Sampler module that, like a wave-table Oscillator, also reads input from a file. The Sampler plays this file (the sample) at a certain pitch every time an event is triggered. The Sampler’s first gate signal represents the gate. Every time the gate goes positive, the sample will play. The first control input determines at what pitch the sample will play. The sample may easily be reversed, looped or repeated by manipulating the order in which samples are played.

**Other Patterns:** Consider using Oscillator to control a Sampler’s pitch.

---

12 Pitch, not frequency: the sample is played at a higher than original pitch by playing the sample faster. An octave higher would mean playing every other sample from the Wavetable. An octave lower would mean playing every sample twice.
2.8. Processor

Guard: The design decisions raised in the Synth Module pattern should be considered before applying Processor.

Problem Statement: Once audio signals are generated, there must be a way to alter them. This allows audio signals to be adjusted to get desired sounds.

Forces: Sounds can be modified in a number of ways. There are many different algorithms, and families of algorithms, many taking different number of control and audio inputs. A consistent interface to these various algorithms is needed, so that the algorithms can be interchanged.

Solution: Therefore, provide an abstract Processor class. Each processor will encapsulate a unique algorithm and adhere to a consistent interface. These processors will be interchangeable.

Implementation Notes: A natural way to implement processors is to encapsulate each algorithm in a Strategy object. Any of these Strategy objects can then be included in the Processor pattern. The Processor object can provide accessor methods to the input signals so the algorithm can recover any information it needs. This allows any number of inputs to be used by the algorithm. The Processor can then pass a reference to itself to the algorithm every time a sample needs processing.

An alternative to encapsulating each algorithm in a Strategy object is to have each algorithm contained in its own class. This approach makes the algorithm explicitly linked to the Module. We instead prefer to separate the algorithm as much as possible from the design of the pattern, in this way, allowing the synthesizer and the algorithms behind the synthesizer to be developed separately.

Another way to pass the input signals to the algorithm is to explicitly pass them as parameters in the call to the algorithm. One problem with this approach is that the Processor does not know how many input signals the algorithm is expecting. Another problem is that the algorithm may not need all the information for every sample it processes. Allowing the algorithm to request information only as it needs it assures no unnecessary information is passed.

The Functor idiom described by Coplien\textsuperscript{13} can be very useful when implementing the Processor pattern. When Processors are implemented as Functors (objects that behave as functions, or as Coplien points out, functions that behave as objects), a natural applicative style of programming emerges that is well suited to modular synthesis in software. Processors are applied to audio signals to produce modified signals.

\textsuperscript{13} Coplien, “Advanced C++: Programming Styles and Idioms”, 1992, Addison-Wesley, pp. 165-178
Coplien also gives an example using Functors to implement filters for electrical circuits. That example provides a good model of how to implement various kinds of Processors (including filters) for audio signals.

**Other Patterns:** Consider applying the GoF Template Method or Strategy patterns to encapsulate the algorithms. Consider also applying the GoF Singleton or Flyweight patterns to implement stateless algorithms.

### 2.9. Shared History Processor

**Guard:** The design decisions raised in the Synth Module and Processor patterns should be considered before applying Shared History Processor.

**Problem Statement:** Not all effects may be performed on a signal by considering a single sample. Multiple samples must be considered in some cases to determine the resulting frequency of a signal.

**Forces:** When several Processors are modifying the same signal (as in an EQ band), it is wasteful to have each Processor contain a separate copy of the signals passed.

**Solution:** Therefore, provide a Shared History Processor class to allow sharing of previous samples among many Shared History Processor modules, thus conserving storage space and processing time.

**Implementation Notes:** One way to share samples among Shared History Processors is to maintain a pointer to a shared reference counted table within each Shared History Processor, and provide a subscription method so that other Shared History Processors for that same signal can gain access to the table. When the first Shared History Processor is added to the audio signal, it creates a table, assumes ownership of the table, and sets the table’s reference count to one. Whenever the audio signal advances, the current table owner updates the table. When any subsequent Shared History Processor is added to the signal, it registers with any other existing Shared History Processor for that signal, receives a pointer to the reference counted table, and increments the table’s reference count. When a Shared History Processor is removed from the signal, it decrements the table’s reference count. If the count reaches zero, the Shared History Processor recognizes that is the last one associated with the signal, and destroys the table. If the count does not reach zero after decrement, but the Shared History Processor being removed is the table’s owner, it passes ownership to any one of remaining Shared History Processors. In this way, a shared table can be maintained independent of the order in which Shared History Processors are added or removed.
2.10. Patch

**Guard:** The design decisions raised in the Synth Module pattern should be considered before applying Patch.

**Problem Statement:** Multiple modules are often required to produce desired sound. A way of consolidating these multiple modules into a single logical module greatly aids in the application of modular synthesis.

**Forces:** It is necessary to provide an abstraction barrier between modules inside the logical module and modules outside the logical module. Modules inside a logical module will rely on the output of modules both inside and outside the logical module for both audio and control inputs. The modules inside the logical module must also have a way of passing their output back to the exterior modules. Modules outside the logical module must be shielded from the complexity of details of the internal modules and their interconnections.

**Solution:** Therefore, provide a Patch module that behaves as an intermediary between modules inside the Patch and modules outside the Patch.

**Implementation Notes:** Several GoF patterns, notably Façade, Mediator, Composite, Chain of Responsibility, Memento and Command, offer natural ways to implement the Patch pattern.

Allowing modules outside the patch to access modules inside the patch directly, or vice versa, can result in unwanted interdependencies between these modules. Users of a patch would have to know about the modules inside of the patch, and where to connect the various signals. This would violate the purpose of the Patch pattern, to provide a single logical module that encapsulates such details. Applying the Façade pattern to a patch creates an abstraction barrier to such interdependencies. By applying the Façade pattern, we allow the user to encapsulate many modules into one ‘black box’ that provides the desired behavior. The details of how each such black box works are hidden, as they are not necessary to understand the overall behavior of the patch. Once these black boxes are created, one can also reuse them.

Because a patch may require multiple inputs and possibly produce multiple outputs, applying the Mediator pattern is useful when implementing Patch. Applying the Mediator pattern to a patch allows behavioral as well as structural decoupling between modules that provide inputs to the patch, modules inside the patch, and modules for which the patch produces outputs.

Applying the Composite pattern allows a Patch to hold references to other constituent modules, some of which may in turn also be Patch modules. This approach is the one used in Soft-Synth. Reaktor also appears to use this pattern: it allows the user to create several ‘Instruments’ in its GUI that are essentially sets of components grouped together in a window. The user can also view individual components that make up the ‘Instru-
Each module in a patch must be advanced once for each sample the program generates. A natural way to do this is using the Chain of Responsibility pattern. The problem with this is making sure a module is not advanced more than once. The easiest way to avoid this problem is to realize that each module is immediately contained by exactly one composite Patch object (albeit patches can be nested arbitrarily deeply). Therefore, if modules are advanced by the Patch that immediately contains them, each will only be advanced once per sample. Thus, the use of the Composite pattern in recursively nested Patches makes this use of Chain of Responsibility possible. Patches must be careful not to create any race conditions between modules. To assure that no module is stepped through more than once for each sample, a Patch may not produce feedback loops with modules outside the Patch.

By saving the state of a patch using the Memento pattern, we can use any patch previously created in any new patch we are working on. We can create any specific patch once, externalize its state, and then reuse it in any patch we make later. Applying Memento provides great reusability of patches. Not only can a user repeatedly call up the exact same patch with ease (this is a problem in hardware analog synthesizers), but it is also easy to reuse any patch within another.

Applying the Command pattern can also be useful when implementing Patch. Commands to load and store patches can be combined with a simple command line interface with commands including output \texttt{filename}, and load \texttt{filename}, to give the user control over loading and storing patches. In Soft-Synth, Patch files start with an \texttt{int n} representing the number of modules that are used by that patch. The next \texttt{n} lines contain descriptions of all \texttt{n} modules. The rest of the patch file contains the signal routing instructions containing a number corresponding to the source, and destination module, as well as the volume, and type of signal (control, gate, or audio; channel number).

\textbf{Other Patterns:} Consider applying the Façade, Mediator, Composite, Chain of Responsibility, Memento, and Command patterns to implement Patches. When applying the Chain of Responsibility pattern to implement Patches, be aware of the potential for race conditions.

\section*{2.11. Noise Generator}

\textbf{Guard:} The design decisions raised in the Synth Module and Generator patterns should considered before applying Noise Generator.

\textbf{Problem Statement:} Effectively random waveforms are highly useful for generating a wide range of sound effects. They can be used directly as sound waveforms to produce a “white noise” effect, or as control signals to other modules.
**Forces:** Pseudo-random number generation is available in many programming environments. These pseudo-random numbers can be used as inputs to various combinations of modules to produce randomized waveforms.

**Solution:** Therefore, provide a Noise Generator module that produces a randomized waveform as its output.

**Implementation Notes:** A simple way to implement a noise generator is to create an Oscillator Strategy object that outputs pseudo-random numbers obtained directly from the programming environment. A randomized timing signal could be used to re-seed the pseudo-random number sequence at appropriate intervals.

Another way to design a Noise Generator is to record a pseudo-random number sequence in a wave table. A Sequencer could then feed the signal to an Oscillator, creating random frequencies. The length of the pseudo-random sequence would need to be sufficient to ensure that over an appropriately long time period, the repeated sequence still appears random.

Yet another way to generate a randomized signal is to use a pseudo-random number sequence to randomly set the frequency of one or more oscillators to random values. Combining outputs of several oscillators using a Patch can produce a richly convolved signal. This would result in a Noise Generator that is a compound rather than elementary object.

**Other Patterns:** Consider applying one or more of Sequencer, Sampler, Oscillator and Patch to implement a Noise Generator.

### 2.12. Synth Output

**Guard:** The design decisions raised in the Synth Module and Patch patterns should be considered before applying Synth Output.

**Problem Statement:** Modules produce signals that must be played on a sound card or other device to become actual sounds.

**Forces:** Encapsulating the details of a device within a module reduces coupling between modules, improving flexibility and promoting reuse. Different signals may require coordination prior to delivery to the device. Furthermore, a natural point of concurrency control for device access is needed if synthesizer modules are operating in separate threads of execution.

**Solution:** Therefore, provide one or more classes derived from a Synth Output base class to encapsulate the details of different output devices. A Synth Output module could take audio inputs for every physical output, thus allowing discrete stereo, or separate audio signals to be passed out of the computer.
**Implementation Notes:** A Synth Output sends the generated audio signals to a sound card, and thus may be implemented as a GoF Singleton if there is only a single card. Often, a Synth Output is only used within the program’s main function, but there is a chance that the user may decide to create a Synth Output within one of her patches.

Multi-threaded programs should not allow two Synth Outputs with the same output destination to be in use at any one time, because of a potential race condition of output signals at the sound card or output file. One solution is to apply the Double-Checked Locking optimization pattern to a Singleton Synth Output accessed by all threads. Another approach is to direct output from each Synth Output to a separate file, and to employ a Synth Output Singleton that accesses the appropriate instance from Thread-Specific Storage, for each thread.

Applying the GoF Command pattern can also be useful when implementing Synth Output. In this approach, an output command tells the Synth Output to output its signal to a given audio file. Adding an output command to the command line interface described in Section 2.10 gives the user control over output redirection.

**Other Patterns:** Consider applying the Singleton, Thread-Specific Storage, Double-Checked Locking, and Command patterns to implement Synth Output.

### 3. Applicability

This pattern language is very useful for sound production in a low-cost environment. Until recently (with the possible exception of the Commodore 64), computers were not fast enough to handle the task of modular synthesis, and the only machines able to accomplish this were expensive samplers and synthesizers such as the Korg Wavestation. The recent trend, however, is to replace these hardware devices by virtual ones, such as Native Instrument’s Reaktor, or Steinberg’s Cubasis VST. These modular synthesis programs are more affordable if not more reliable. Both the Synthesizer pattern language, and Soft-Synth framework could prove extremely useful if current trends persist.

Soft-Synth allows an arbitrary number of Modules to feed a single input. Whenever an input sample is needed, a Module will scale down all samples on the same channel by different amounts, and add them together. Neither MaxDSP nor Reaktor allow the user to designate more than one input signal to a single port. This means the user must create mixer modules to serve as an intermediary between the devices the user wants to connect.

---


Two examples help demonstrate some useful applications of this pattern language. To create a vibrato effect, one could use a Low Frequency sine wave Oscillator (LFO) to control a filter, as shown in Figure 1.

![Figure 1: Vibrato Module](image)

Any audio signal received by the filter would then have a vibrato added. The vibrato module could also become a composite Patch that would function as a Processor.

To create a random melody, the user could feed the output from a Noise Generator into a Sample-and-hold module. The output would be a random level. This random level would then be used as the control signal to an audible frequency digital controlled oscillator (DCO), as shown in Figure 2. The frequency would therefore randomly change at a set interval.

![Figure 2: Random Melody Module](image)

The random melody module would function much like a generator. Furthermore, the output from the random melody module could be passed to the vibrato module input to produce yet another, random vibrato melody module.

Modular synthesis is particularly useful when one wishes to modify an existing signal. This is useful during live performances, when a musician wishes to digitally modify a sound he or she is generating. MaxDSP is geared toward this sort of signal manipulation.
Modular synthesis is also very useful in generating sound, either constant, through the use of oscillators and filters/amplifiers, or in response to an event, through the use of triggers. Reaktor is more geared toward this latter use. It involves the use of MIDI signals to trigger its synthesizer capabilities. Thus the entire patches in the end wind up working much as the individual components do: they either generate sound, or modify it in some way. The next logical step in developing this framework is to involve MIDI applications in its design. This would mean developing additional patterns to deal with such issues as polyphony.

4. Consequences

Even though the ideas behind modular synthesis have been around for more than fifty years, it is still the most widely used synthesis technique in electronic synthesizers. These principles are constant whether modular synthesis is implemented in an analog or digital environment. This framework merely attempts to translate it into a digital software form.

One major consequence of using modular synthesis on a computer is the requirement for expensive Digital to Analog Converters (DACs) to preserve full quality of sound. These expensive sound cards are essential to inputting and outputting audio signals to/from the computer. They usually have at least four mono inputs, or two stereo inputs, and can generate 24-bit samples at 96kHz. The drawbacks become most evident when a user wishes to input more signals than are available on the sound card. With physical synthesizers this is not an issue because each module had its own input and output jack.

Another consequence is more subjective. Many musicians enjoy the ‘hands on’ manipulation of the dials and pots on a physical synthesizer. Although this manipulation can be emulated by an on-screen GUI, many musicians prefer physical interaction with their instruments. This drawback can be solved in part by inputting signals from such physical controls to the program, for example via a USB port, or MIDI cable.

The last consequence in using a general-purpose computer for modular synthesis is the possibility of an occasional ‘hiccup’ encountered during execution, which can cause several samples to be skipped. These ‘hicups’ are greatly reduced when using a very fast computer. However, they are still a problem no matter how infrequent. Physical synthesizers and their real-time operating systems are specifically designed to avoid these hiccups; in general, a Pentium running Windows is not. One increasingly affordable solution is to run the software using a real-time operating system such as LynxOS16 or QNX Neutrino17 on a general-purpose CPU. Availability of device drivers for various sound cards may also make running a variant of an open-source operating system such as Linux18 an attractive option.

16 http://www.lynx.com/products/realtimesos.html
18 http://www.linux.org
5. 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 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 comments on this paper, and his ongoing support of this work. We also would like to thank Jeff Parsons, Research Associate in the Center for Distributed Object Computing, for his comments on this paper and for pointing out the relevance of Coplien’s Functor idiom discussion to the Processor and Shared History Processor patterns.
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.