Managing Complex Patches in Max

by Arne Eigenfeldt

One of the beauties of Max is its simplicity: the ability to quickly create a patch that does something artistically interesting. Part of this has to do with its visual programming style - patchcords allow us to see the relationship between graphic objects. However, unless you limit yourself to creating only straightforward patches, your patch can become a spaghetti-like series of connections that confound attempts at debugging.

Some users suggest that the potential of Max resides in its simplicity; in order to create truly complex patches, one must use procedural languages that Max can contain within it: Java, Javascript, ChuCK, RTCmix, et al. But I've been using Max for a long time, and I really feel comfortable working within its environment: I don't want to learn another language. Can't I use Max to create *Really Complex Patches*?

Of course, complexity can mean many different things. I've stared at single window FFT patchers for an hour, trying to understand them. But that type of patch will behave the same way (hopefully) every time. The complexity that I'm talking about are those patches that take ages to load, the ones with seemingly countless nested subpatchers that you can't even remember creating.

In the past few years, I've spent most of my Max time developing one major patch, and I've had to develop some procedures that deal with organizing complexity within Max itself.

A caveat: I am not trained as a programmer - I'm a composer who programs. The ideas that I'm suggesting are ones that I've found - in my sixteen years of programming specifically in Max - work well.

Programming 101

Some of these ideas are probably found in a first year programming course - but if you've taken a first year programming course, you probably don't need to read this, right?

And you've read the tutorials, right? For example, page 338, "Encapsulation", which deals with some issues of complex patches, and page 386 "Debugging", which deals with some basic debugging ideas.

Before we begin...
The tendency to take working patches and keep adding to them is one way to create complexity, perhaps inadvertently. Unfortunately, this can also lead to confusion, particularly if one simply adds more patch cords between processes.

For example, a given **gate** may control whether a process operates. As new situations arise, new processes are added via more patchcords to control this **gate**; while each addition may make perfect sense while you are adding it, a week later it may have become difficult to determine the relationship between processes that control the gate.

There are ways to solve this (outlined later), but it involves time spent cleaning up code; I don't think any programmers enjoy cleaning up code; most would rather solve a new problem. Therefore, it's best to avoid these problems before they happen.

In these cases, it's sometimes better to start over, salvaging parts of the existing code as modules. Instead of adding more rooms to your clubhouse, lay a better foundation and build your mansion from scratch.

Structured programming
One very useful traditional idea is the concept of structured programming. This notion suggests that "large problems are more easily solved if treated as collections of smaller problems" (O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, London, Academic Press, 1972). These "small problems" are handled in individual modules - in our case, they would be subpatchers or abstractions.

We tend to think of subpatchers as a method to clean up code (you gotta love the **Encapsulate** command!); however, *designing* your patch modularly - with subpatchers acting as modules - has many benefits. As outlined in in the Tutorials (p.339), these include:

1. smaller modules are easier to test;

2. modules can be reused. Once you have one that works, you don't have to keep rewriting it;

3. it is easier to see how a program works later on, especially if you need to debug it.

Subpatcher or Abstraction
If you find that you are using the same subpatcher over and over, then it is time to make that subpatcher an abstraction by saving it separately. When using abstractions, you are guaranteed of reusing working code; this becomes important when you edit the original abstraction - you know that *all* instances will be updated (rather than having to hunt around looking for copies of nested subpatchers).

On the other hand, handling data inside an abstraction is sometimes tricky during debugging. Naming data structures using changeable arguments (for example, giving a

coll the name #1_mycoll or a value object the name #1_myvalue) will clear them in all instances when you resave the original patcher. The use of an initialization routine will be discussed later.

Functional strength - defining a subpatcher
One tenet of structured programming has to do with how information flows between modules. Each module should be self contained, and have (ideally) one entry and one exit point; you shouldn't jump from the middle of one module to the middle of another. In Max, this means that each subpatcher should have a limited number of inlets and outlets.

Furthermore, each subpatcher should execute a clear task (and be named after that task): if your subpatcher does this, it would be considered to have *functional strength* (which is a *good* thing).

Does it take more than a few words to describe the purpose of a subpatcher? Does the name contain any conjunctions? If yes, the module is not "functionally bound", and needs to be broken up.

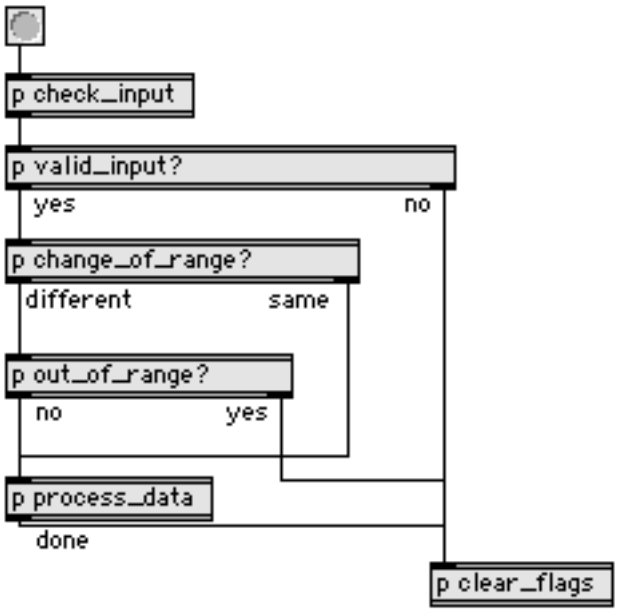
For example, consider the following:

A subpatcher doing too much

The function involves two separate concepts (checking the data, then sending it out). The following is not only cleaner and easier to debug, it also guarantees that data is not sent out until after it is checked.

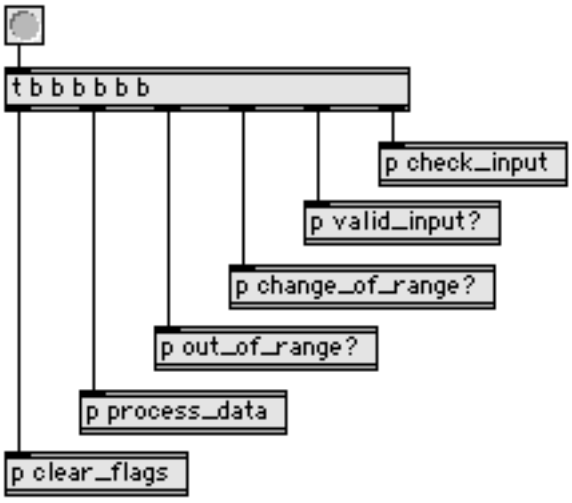
Breaking up functions

Controlling Program Flow
Following these few concepts, patches with much clearer program flow will result:



An example of structured programming within Max

Notice that each patcher has one inlet (entry point) and, at most, two outlets (exit points). This ensures that the process will be compete before moving to the next patcher.
Using a **trigger** object, which works very well when executing straightforward processes (storing data, for example), can potentially cause problems.



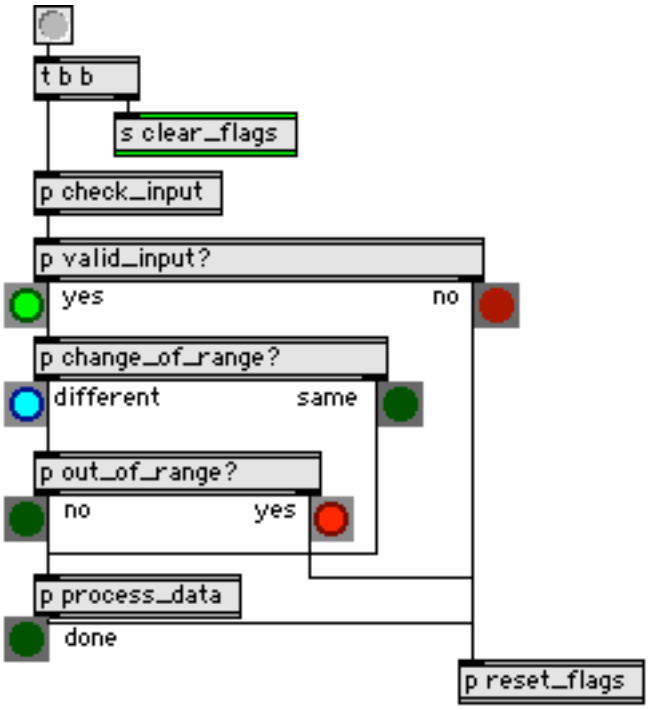
Questionable program flow that may lead to stack overflow

In this case, let's assume [process_data] involves some complex, time consuming calculations. In order to avoid stack overflow, you might put one or more **deferlow** objects inside this subpatcher. In that case, when does [clear_flags] get executed? Before or after deferred control returns to [process_data]?

I prefer to avoid these types of questions, and instead guarantee the program flow in the earlier version.

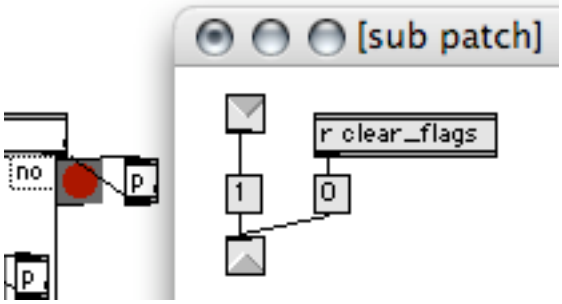
Debugging modular code

It is also much easier to trace program flow in modular code. Even after each subpatcher has been tested, it is necessary to follow how the program passes control amongst its modules. I use the **led** object to help me see this during debugging:



*Using the **led** object to trace program flow*

The output of each subpatcher is actually connected to a subpatcher, which is connected to the led:



*The subpatcher to control the **led**s*

This assumes that each subpatcher is passing bangs, which is something I always try to do. In this way, another tenet from structured programming is followed: *don't pass data between modules*. Instead, store the data; each subpatcher can *affect* data, but then pass control to the next subpatcher.

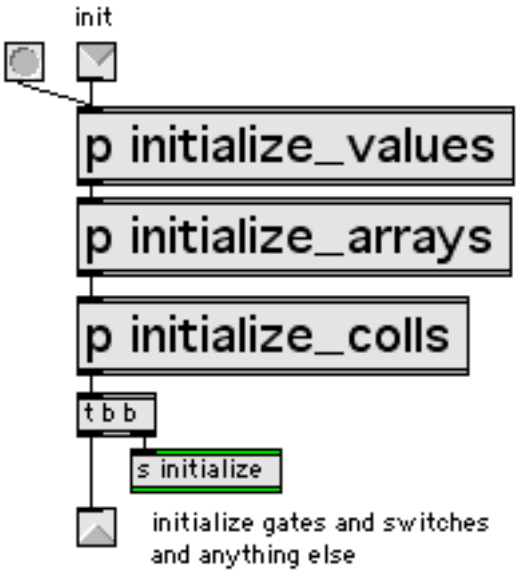
Some general hints and tips:

Here are a few ideas that I've found that I use a lot to help me organize large patches. These aren't just for large patches, but habits that I have found to be useful in all my Max projects:

Create an initialization routine

loadbang works great in basic patches, but can start getting problematic when used to initialize larger patches that involve calculations to initialize data, or cases when one process must be initialized before another.

Similar to the methods of structured programming, I guarantee the order in which initialization occurs:

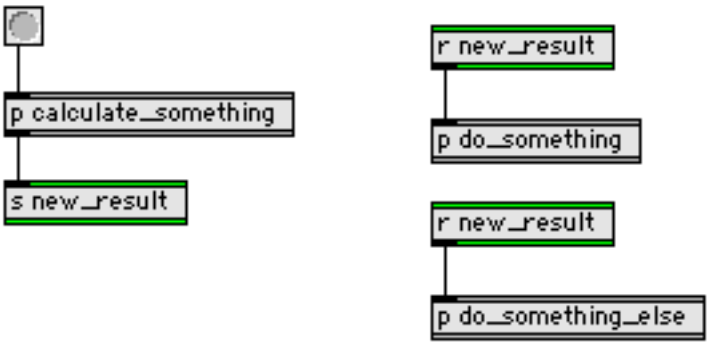


Making sure things initialize in order

Limit your use of Send / Receive

Initially a great way to get rid of extraneous patch cords, these objects can lead to other problems.

In the following patch, a message is sent to the named **receive** object "new_result". The potential problem is at the receiving end. If there are several **receive** objects with this name, which one will get the message first?

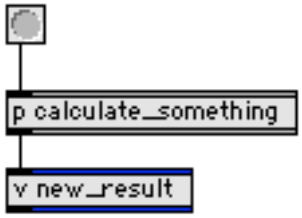


*A typical, if somewhat hazardous, use of **send/receive***

In this example, the lower **receive** will get the message first (since Max works not only right to left, but also bottom to top in ordering messages). But what if the **receive** objects are contained within several layers of subpatchers?

This is further complicated if the subpatchers [do_something] and [do_something_else] operate upon the same data - the order of their operation may be important.

If [calculate_something] only operates upon data, and the result of the calculation is not time dependent (for example, the next note in a pre-composed melody that will be played later), the output should be *stored* for later use. The best object for this is **value**.

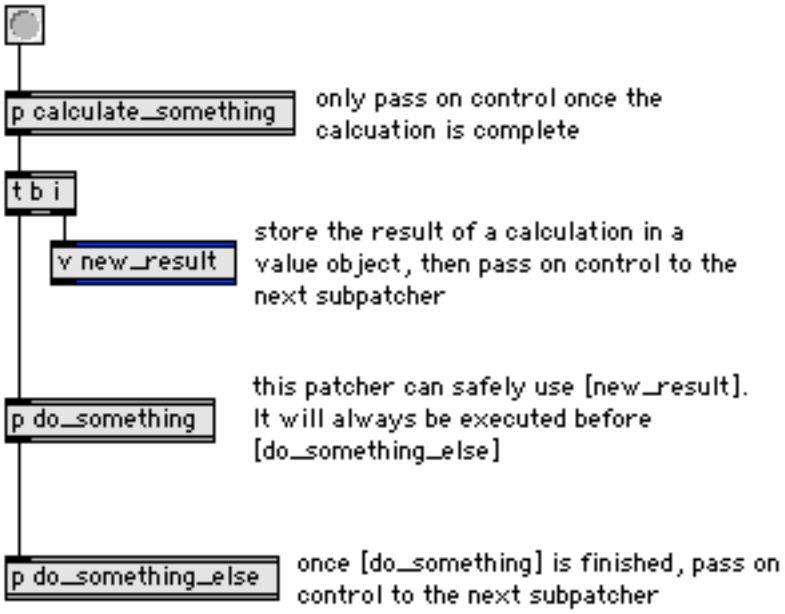


*Using **value** to store data*

Then, inside the other patchers [do_something] and [do_something_else], this data can be accessed when it is needed:

*Accessing **value** somewhere else*

As already mentioned, process order can be controlled so that data is stored *before* it is operated upon. For example, the following is one way to do so (although the storing of data in the **value** [new_result] could have been done inside [calculate_something]:

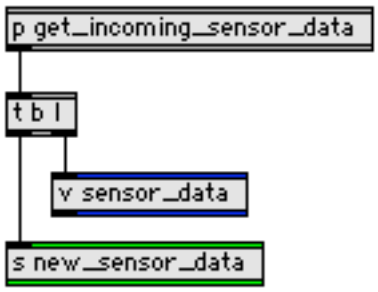


Storing data before operating upon it

send/receive vs. value

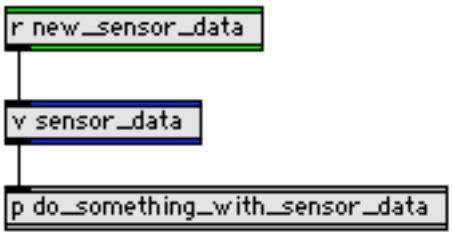
Value objects are great for storing data for later use, but they don't tell you when the data has changed. For example, incoming sensor data may be stored in one or more **value** objects, but the patcher to initiate action based upon this new data isn't directly notified.

Therefore, a **send**/**receive** pair is still necessary. Use such pairs when you are dealing with *time-based* data.



Storing incoming data, and notifying that new data has come in

And in the patcher that operates upon the data:



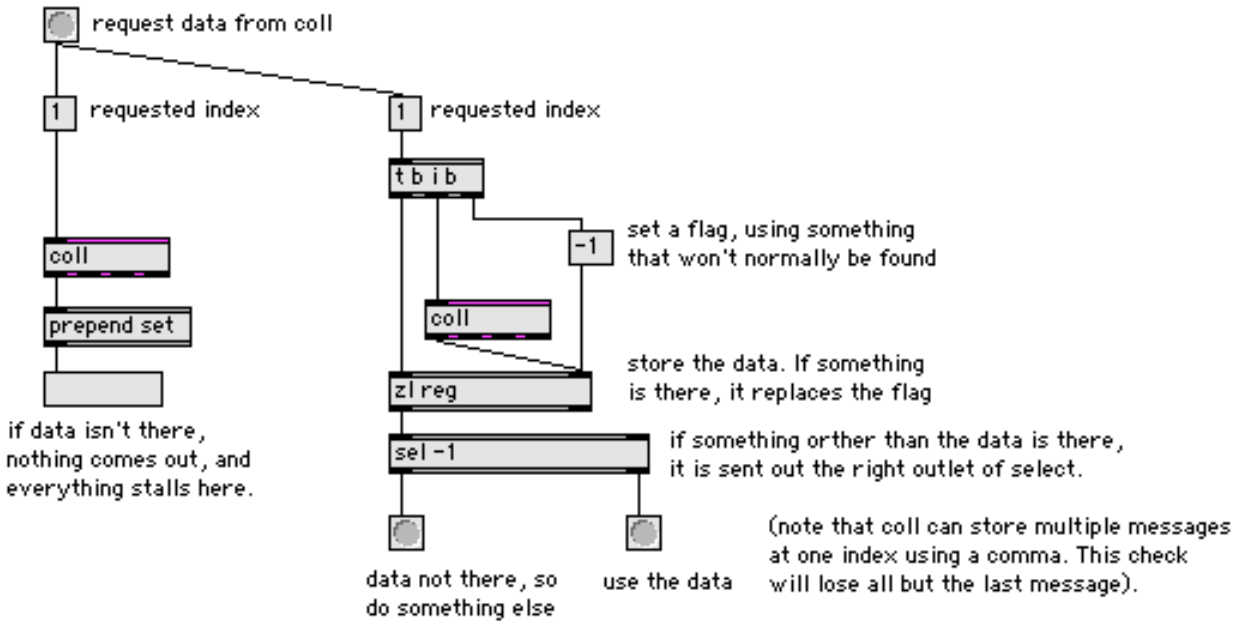
Accessing incoming data immediately after it has come in

Coll: pluses and minuses

Because larger patches often need to store lots of data, **coll** can initially be your friend. One of the best features of **coll** is the ability to quickly and easily *see* your data. Unfortunately, I have had some less than stellar experiences with **coll** when dealing with large sets of data required in a timely manner (in these cases, I *have* moved to Java to utilize the many data handling routines available within it).

However, I find **coll** to be sufficient when working with smaller sets of data. One of its quirks is that it can stop program flow if you request data that isn't there, since it won't return *anything* (unlike **table**, for example, which will always return *something*). See the left side of the example below.

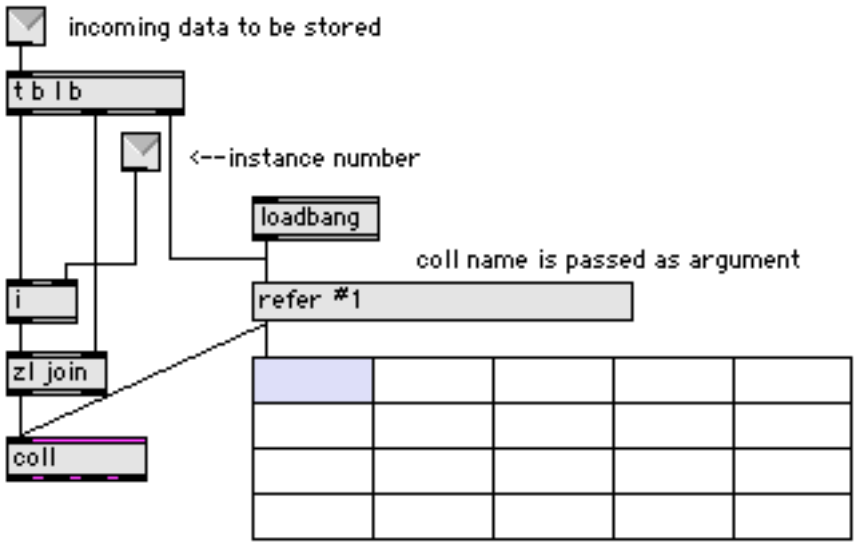
In these cases, use a flag to test whether data is in **coll**: see the right side of the example below.



*Using flags to know when data isn't available in **coll***

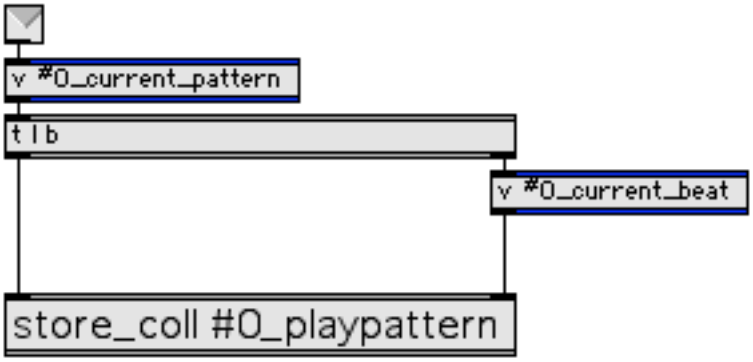
Create abstractions to handle data between instances
One very powerful feature of Max is the ability to use *instances*: abstractions that use arguments to maintain individual data within them (for example, #1_mycoll within the original abstraction.

I use a single abstraction to handle all storage of **coll** data, and another to handle all **value** data. In these cases, a single **coll** or **value** will hold the data for all instances, making it much easier to handle data, especially during debugging and testing.



*My **store_coll** abstraction*

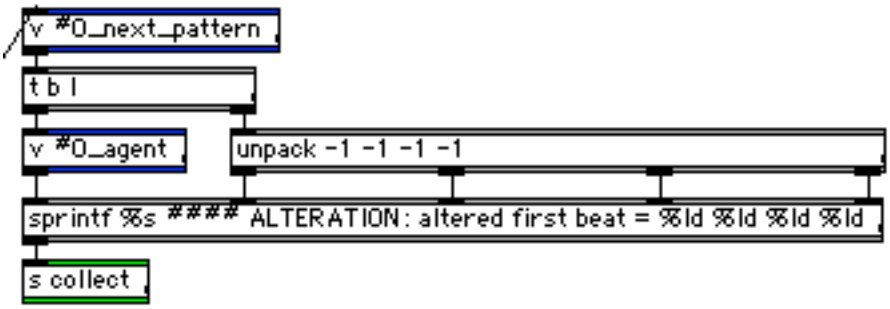
The usage of **store_coll** in an abstraction is shown below. In this case, the **coll** [play-pattern] will hold the patterns for all beats of an instance; this particular subpatcher will move the current pattern into [playpattern] at the current beat offset:



***store_coll** in action*

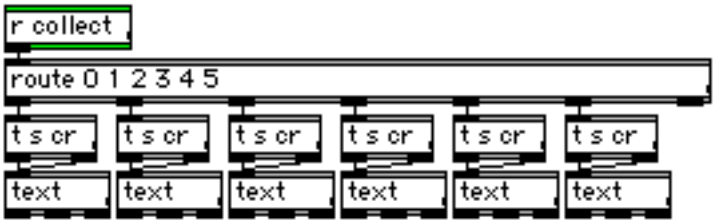
Use the text object to track program flow
The earlier method of tracking program flow using **led** objects works fine for debugging a single subpatcher, but when everything is put together and several instances of an large abstraction are working, this is ineffective. I often want to know what the data in a **value** or **coll** is when entering and exiting a given subpatcher. Printing it to the Max window is equally ineffective, since all instances would print (and cause an overflow of information).

In these cases, I use the **text** object to collect this information, one for each instance. In the example below, I want to know what [next_pattern] is at a certain point of the patch. The **value** is unpacked (unpack is initialized with negative numbers in case [next_pattern] was not initialized, which I would then notice), and combined into a longer message using **sprintf**, and finally sent to a **receive** object named [collect]:



Collecting the state of a data object at a certain time

The main patcher contains six instances of a large abstraction in which the above example is contained. The instance number - named agent in the above example - is stripped off, and routed to individual **text** objects.



*Collecting information from instances in separate **text** objects*

Summing up
Like many people, I use Max because I love its simplicity and its speed. But in order to create complex patches that are created over many months (and even years), a disciplined programming style is required, following some of the basics of traditional structured programming.

Arne Eigenfeldt has been using Max since 1990 when George Lewis showed him a very early version. His Max pieces have been performed throughout Europe and North America, and he teaches Max/MSP at Simon Fraser University (Canada). He is currently pursuing research in creating intelligent software tools using Max, and can be reached at arne_e@sfu.ca.