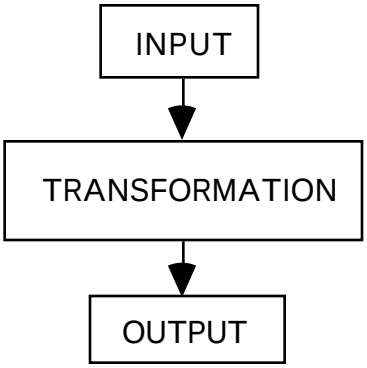
Solving Musical Problems with Max

Max gives us an awesome assortment of tools for accomplishing simple and complex tasks, and includes thorough instructions for how to use them. However, the beginning Max user often feels like someone who has a hammer but has never seen a house. You can sometimes learn from looking at patches built by experienced Max hands, but with a complex patch, the job can be like untangling a plate of spaghetti.

There are several publications that explain how to do various things (including this one, eventually) but sooner or later, you are going to try something on your own. This essay discusses how to solve new problems. I'm going to assume you've read the manual up to the objects section and looked at the tutorial patchers.

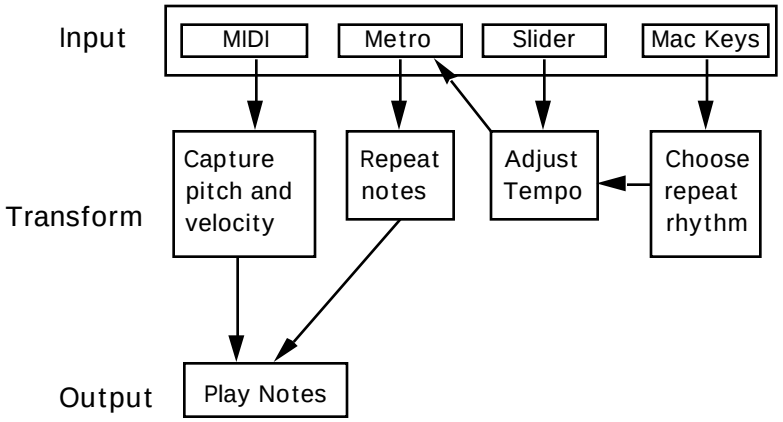
The Big Picture

A patcher usually has three parts: one that detects triggering events, one that produces the result, and stuff in the middle that connects the events to the results. Triggering events may come from the outside world, or may be generated by the computer's timing system. In this drawing, I lump them all together as "input".



Of course it won't be that simple, but you can see the point here. You are either gathering messages from the world, making changes in the messages, or creating output.

Let's try a real problem, say a repeating keyboard. We want the last note played on a MIDI keyboard to repeat at a steady rate. We want to choose the rhythm with the number keys on the Mac keyboard, and set the tempo with an on screen slider.



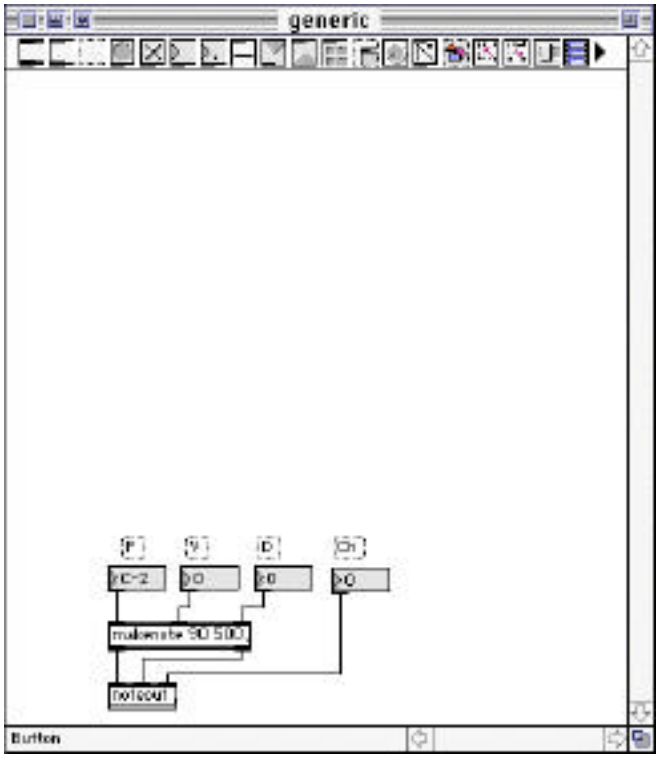
This drawing shows how the functions are broken down into smaller actions. If you can envision your patch in this manner (and draw a simple sketch) you are halfway to building it.

Last Things First

Start with a clear idea of what is supposed to happen. Will your patch play notes, show pictures, control a light board?

Now list everything that has to be known for the event to occur. For a note, that might be pitch, velocity, duration, channel.

Build the output section down at the bottom of the window. Make it work with some temporary controls such as number or message boxes. Stick labels on them, even if their function is obvious. Here's the simplest way I know to play a note:

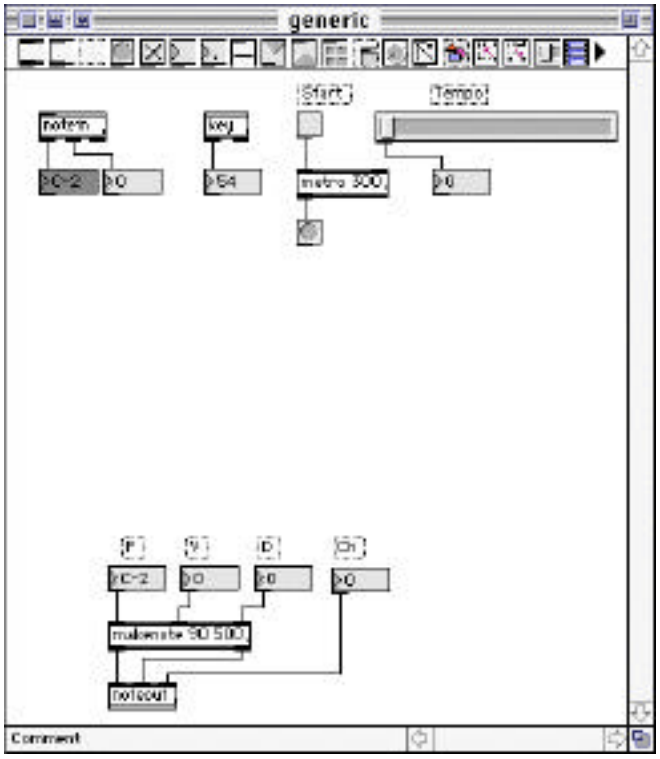


Define The Event That Will Trigger The Result

If the patch is played from MIDI or the Mac keyboard, connect a "-in" or key object to a number box so you can monitor the input. If you are using a metro to generate repeated actions, connect it to a button giving a visual indication of how fast things happen. (Provide ways to turn the metro on and off, and always set a default rate in the metro object.)

If you want user (mouse operated) controls, lay them out across the top of the window in some sensible order without connecting them to anything yet. Play with them and move them around til the layout seems right. Once the layout is set, label them.

Here are the controls called for in the example:

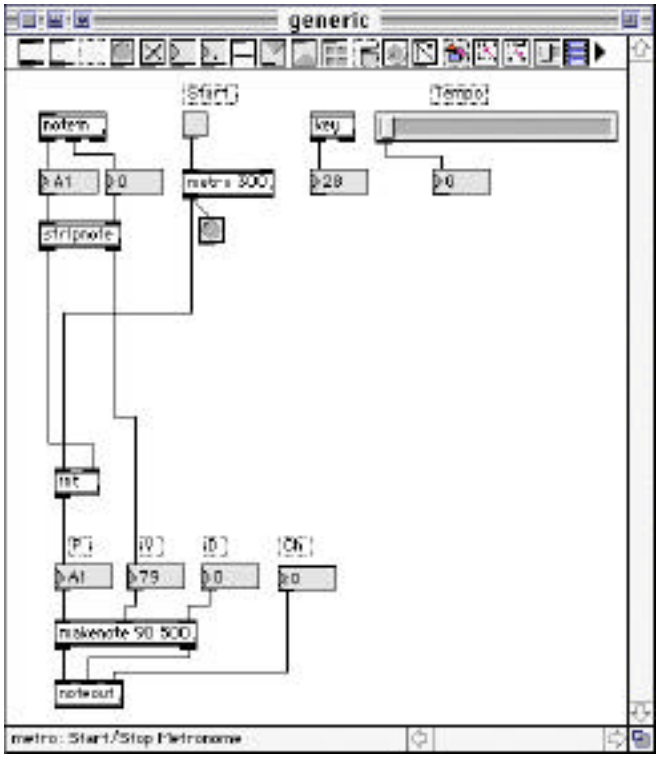


Transformations

You should begin to see the paths the messages will have to take. Of course, these paths will probably not be direct- the messages from the input section will need to be filtered, scaled, or processed in more complex ways. Add these, testing as you go.

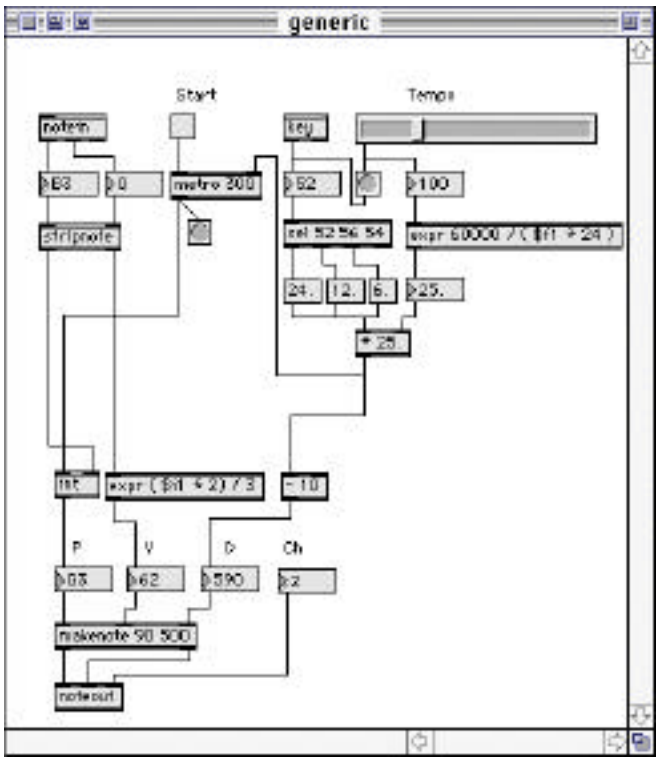
The most common transformation on MIDI data is filtering. (MIDI almost always tells us more that we need to know.) The first step of filtering is to choose an appropriate input object: in this case notein ignores program changes, controls and a lot of other stuff. Even notein can give too much data. Here we only want the note and velocity of the note on, so we add a stripnote.

We also usually need to store information: here the pitch and velocity of the input note are needed for the repeating notes. A number box is a kind of memory, and it will do for the velocity, but I am using the int object to remember the pitch because I don't want to echo the original input. (That's the kind of thing you notice as you test parts of the patch. You also notice ways to keep the patch neat. You can see a slight reorganization of objects to keep the connection from the metro to the int short.)



This is enough to give the basic function- it plays repeated notes. After trying it a bit, I decided I wanted the repeated notes to be quieter than the original, so I added a scaling box to the velocity path.

The next step is to add the tempo controls, which set the metro speed and makenote duration parameters. I discuss this in some detail in the essay on Max & Rhythm. It suffices here to say I'm using some math objects to calculate the metro speed from the slider setting and selected key presses.



Debugging

Now it's ready for testing and debugging. Debugging is one of those arts that can't be explained. The manual has a good chapter on it, and shows how to use the various debugging features built into Max, but it doesn't really convey the knack. The second best way to learn is to watch someone with experience debug a patch. (The best way to learn is to debug your own patches.)

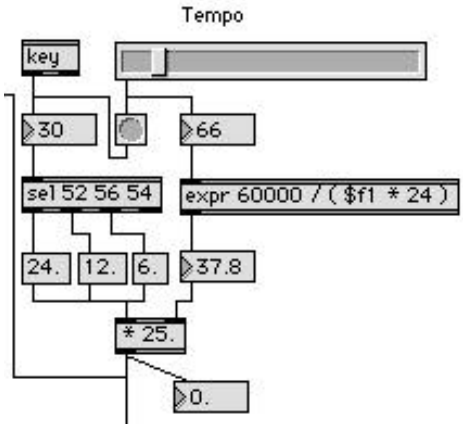
Actually, we have been testing as we went along, using the number boxes to monitor the input and generate trial values for the output section, so we are pretty sure each part of the patch works. For thorough testing, we need to try all combinations of inputs: high and low notes with the metro on and off, at each note value, and both extremes as well as the middle of the tempo.

It turns out there is a bug. The tempo slider doesn't seem to work any more. The number box changes, but the repeats continue at the same rate. Furthermore, when I look at the Max window, I see this:

```
expr:
divide by zero detected
expr:
divide by zero detected
expr:
divide by zero detected
```

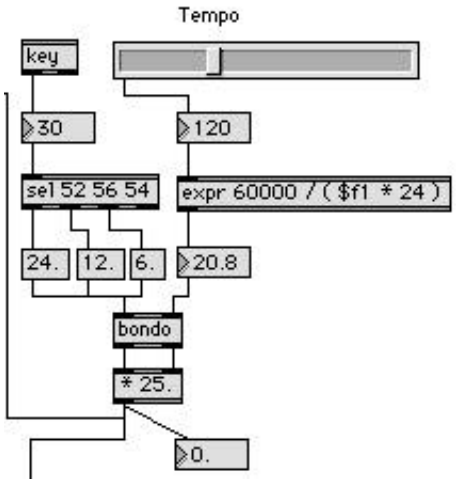
Looking at what's in the expr, it's no surprise how this can happen. When the slider hits zero, there is certainly an attempt to divide by zero. An easy fix is to change the slider range with an offset.

That gets rid of the error messages, but the slider still has no effect on what I hear. My next effort is to watch what happens as I change the slider. I do this by tacking number boxes on every object in the path from the slider to the metro:



Well, there's already plenty of number boxes, so only one more, on the * object will do me any good. It tells me the value doesn't change when I move the slider. (I knew that. Debugging can be the most frustrating....) The value should change, because while the value from the expr is going into the right inlet of the *, changing the slider is also banging the number of the last pressed key through the sel ... oops.

Looking at the value out of the key object, I realize it won't make it through select. Must have hit some other key. A different approach is needed, so I try bondo:



I test this and it goes bonkers, the synth playing bursts of machine gun fire. The diagnostic number box flickers as I move the slider, but the value stays zero. Bondo has

nothing to send. I can fix this by hitting a key, but this will be the situation every time the patch is opened. (You must always save and reopen a patch during the testing process. To be really thorough, you should quit and restart Max too.) The fix is a loadbang to initialize bondo.

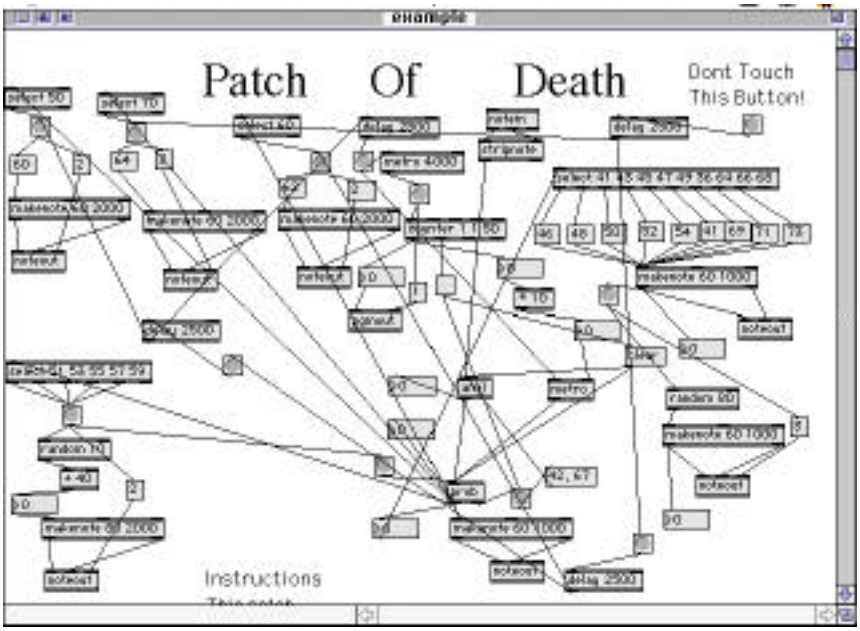
Notice that it took longer to test and debug the patch than it took to design it. This was a typical debugging session, and it was honest. I actually made that mistake in this example and found it as I was writing these words.

Now the patch can be refined. It might be nice if it didn't repeat forever, and we don't need all those number boxes any more. Each change requires retesting, even of parts that don't seem related to the change.

Neatness Counts

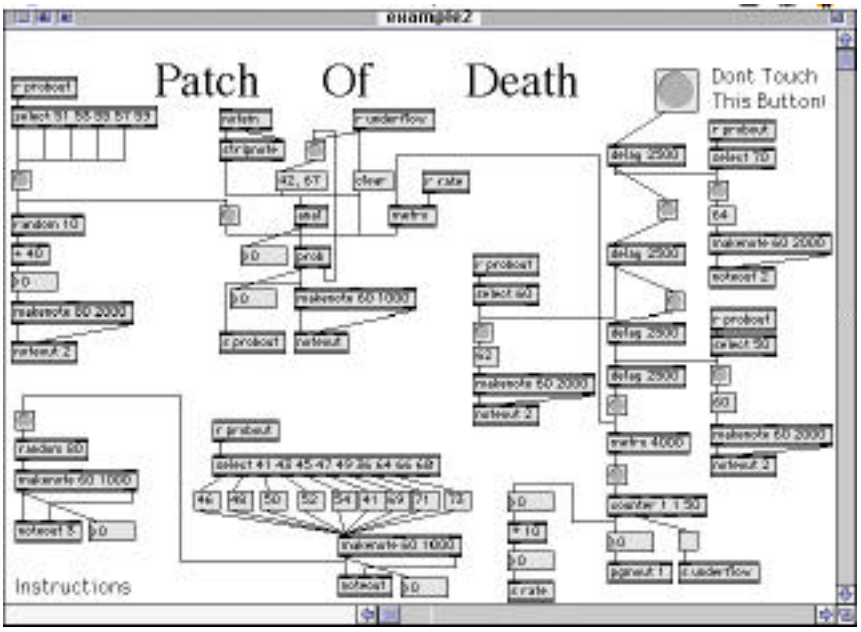
Teachers don't nag students about keeping work neat just because sloppiness is easy to criticize, we're trying to share a principle we've learned the hard way: now matter how clear something seems as you are working on it, a few weeks later you haven't a clue what you were doing.

Here is an example built by a beginning student. It isn't particularly bad, in fact I choose it because it shows how fundamentally good work can be difficult to understand when it's not clearly organized.



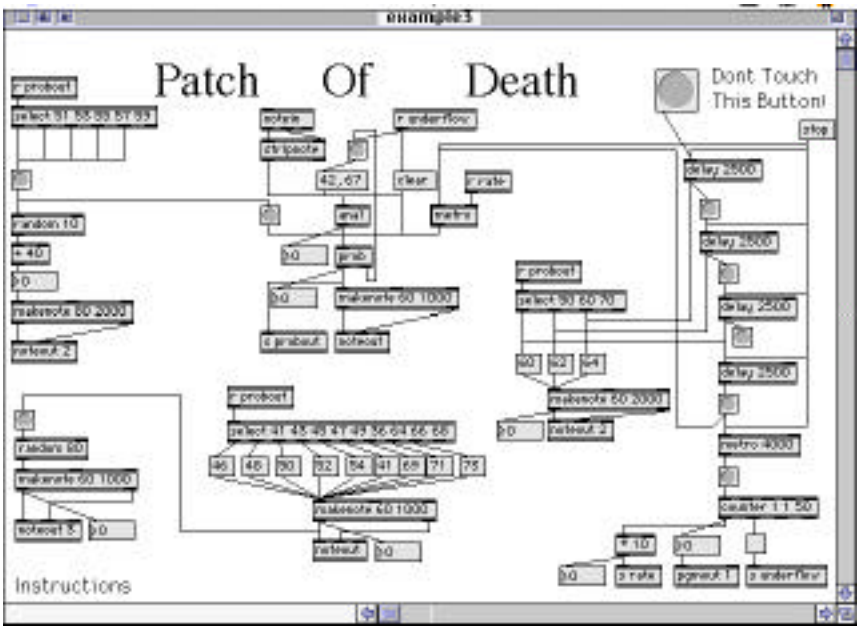
The patch does some interesting things, but how would you make it run a bit faster, and how could you get it to stop?

Here's the same patch, tidied up:



It's still a complex patch, but now the relationships between objects are clear. You can follow the chain of delays that generates the starting gesture, see how the central prob object is supported and find the various destinations for its output. It is clear that the tempo comes from the central metro object, which is controlled by the counter.

In fact, now certain possible improvements become apparent. There are some unnecessary duplications of objects; consolidating those functions makes the opening even clearer and gives the opportunity to add one important feature:



Moral: neat patches really are easier to understand and maintain than cluttered ones.

Guidelines For Readable Patches

• Keep it all on one screen.

At least if you can without cramming everything together. There's nothing wrong with a top screen full of subpatchers- that shows off the logical structure of your program.

• If you can't keep it all on the screen, scroll only one way.

It's just too easy to get lost in huge windows. Patches usually have a single direction, down for a process with a lot of steps, across for several parallel activities.

• Flow downward.

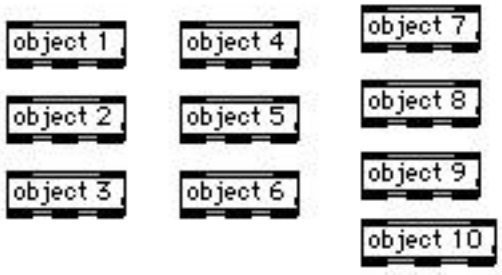
It just seems sensible that consequences are below causes. Besides, the cords get too kinky when you move up.

• Keep related objects together.

Sure, it keeps the cords short, but it also makes it clear how things fit together. When objects aren't related, leave a bit of extra space.

•Align vertically, misalign horizontally.

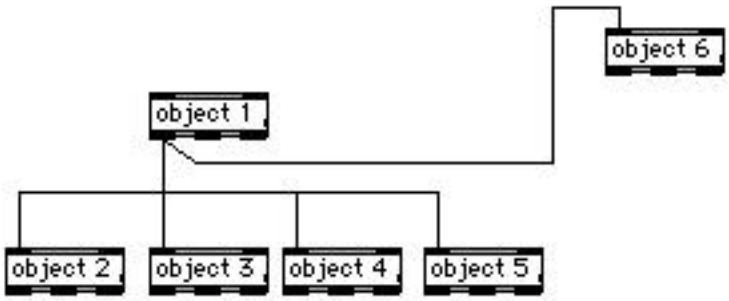
Alignment helps the eye make associations. When side by side objects are related, align them horizontally, but break the line between unrelated clusters of objects. This shows what I mean:



Even without cords, objects 1 through 6 seem to form one group and 7 through 10 form another.

• Use segmented cords, most of the time.

A few angles break up the monotony. There's no point in segmenting temporary connections for diagnostics and tests. In fact, the angled cords make them easy to find when it's time to take them out. Sometimes a short angle on a segmented cord can clarify what might be a confusing connection:

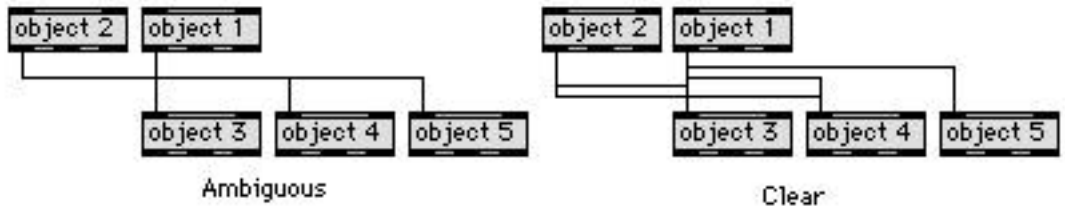


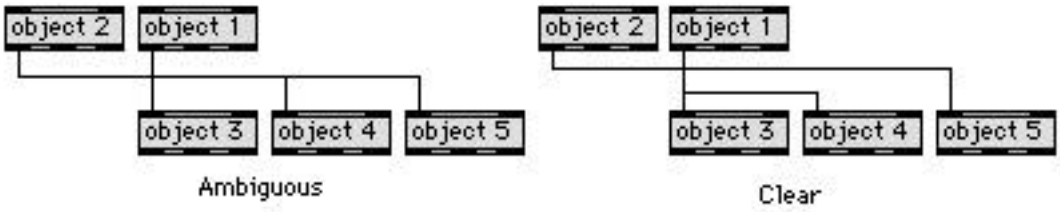
• Don't run cords over objects.

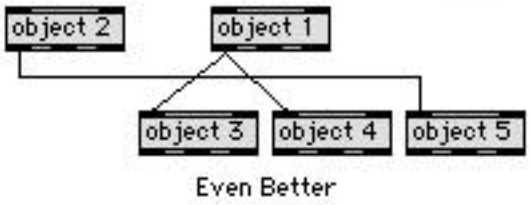
It makes the lettering hard to read, and you can't tell what's connected.

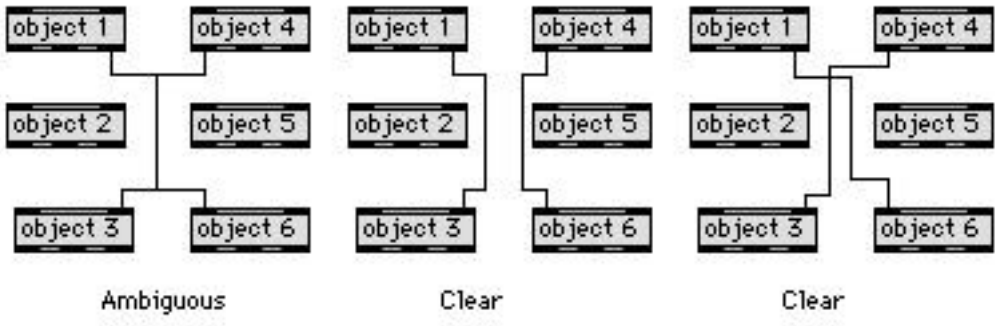
• Make intersections unambiguous.

A full cross should always mean the cords cross with no connection. A three way means two cords join going to a common inlet or outlet. Don't stack cords that aren't going to the same place. Here are some examples:

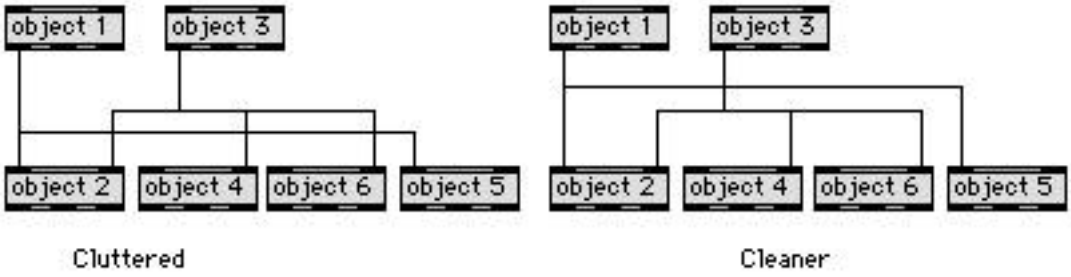








• Route cords to minimize intersections



• Use trigger objects for fanout at the end of long cords.

If you take an output across a window with two or three cords, use one cord to a trigger rather than several parallel cords. This is more reliable as well as neat.

• Use send and pv instead of convoluted cords.

You can only have five corners on a cord. You should never need more than three.

• Make your names for send and value mean something.

Names like rate and theNote can help you understand what is going on. Names like Fred and G3 don't.

• Subpatchers should only have one function, but they should have all of it.

A subpatcher should have a simple purpose. All of the objects needed should be in there, even if it means duplicating a bit. A subpatcher should be ready to move out on its own if it gets the opportunity.

• Subpatcher names should mean something too.

If the subpatcher has a single purpose, the name won't be hard to find.

• Never use two objects where one will do.

Don't send a message to two of the same objects.
Don't use messages to set values that can be set with arguments.
Use the comma in message boxes when two messages are always needed.
Don't use a button to make a bang if the message is already a bang. (Unless you need a visual indicator)