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[1]. It isn't particularly bad, in fact I choose it because it demonstrates how fundamentally good work can be difficult to understand when it's not clearly organized.

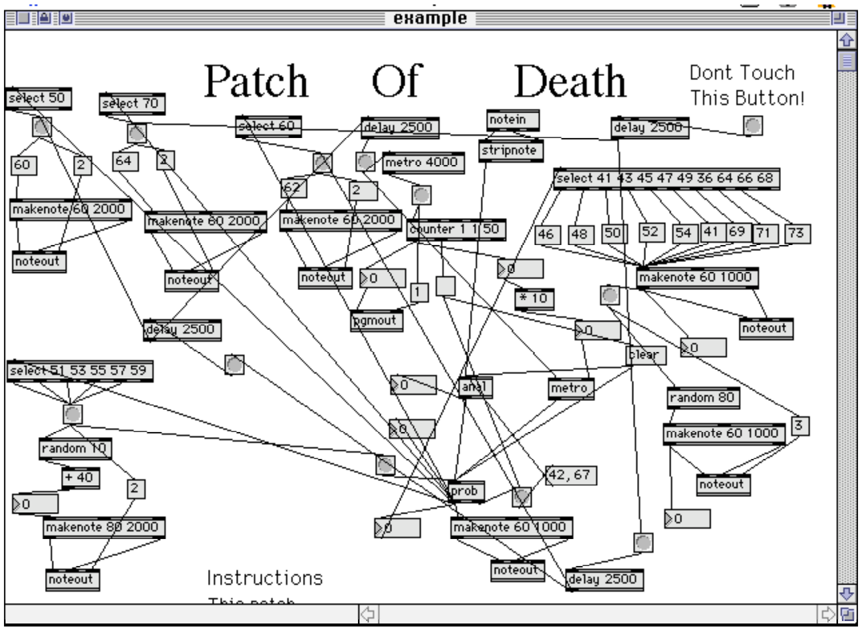

Figure 1.

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

Figure 2 is the same patch, tidied up:

[1] Old style Max patch version 3 or so. Thinks look different now, but these principles still apply.

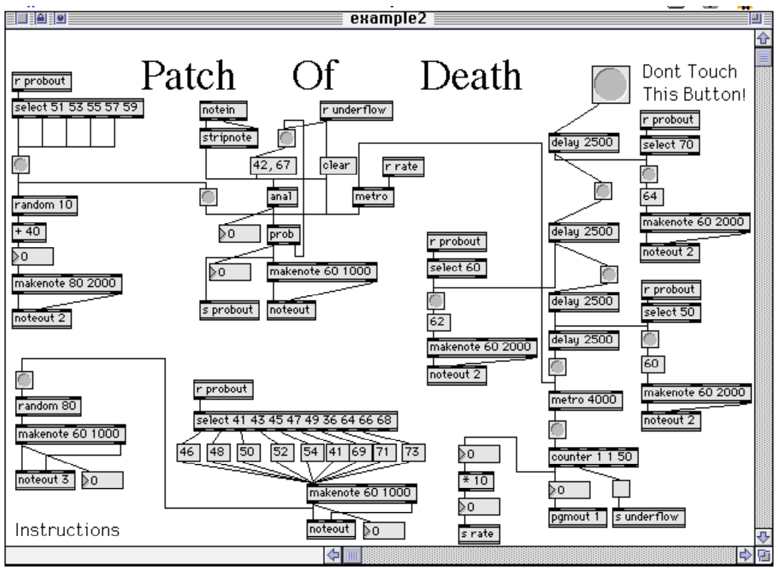

Figure 2.
It's still a complex patch, but now the relationships between objects are clear. You can follow the chain of delays that generates the starting gesture, see how the central prob object is supported and find the various destinations for its output. It is clear that the tempo comes from the central 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[2]:

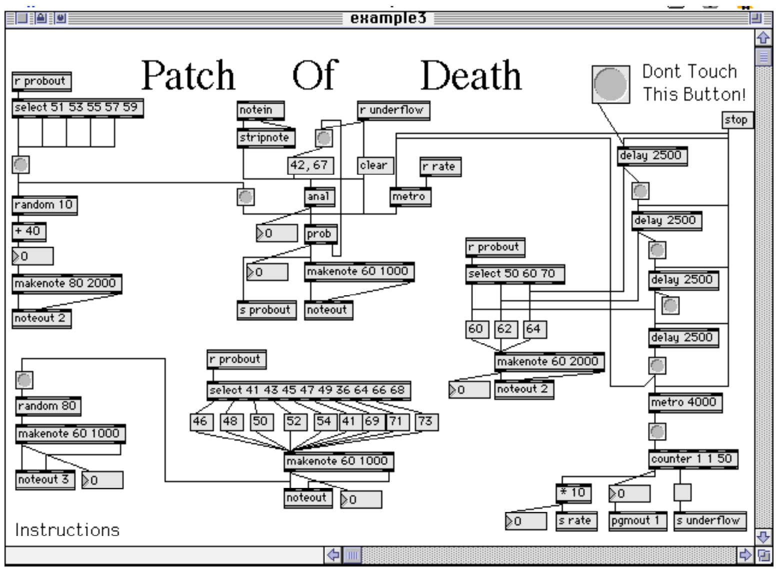

Figure 3.
Moral: neat patches really are easier to understand and maintain than cluttered ones.

[2] It's hard to read in this size, but the main flaw of the original was it wouldn't shut up. A stop button is a useful element in any patcher.

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 are working on a large screen, don't fill it entirely with the patch- it's still pretty easy to loose things in the corner. If you work on a laptop, use windows that will fit in the lowest resolution you use. You can get stuck with an over size window if yo connect to a projector.

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:

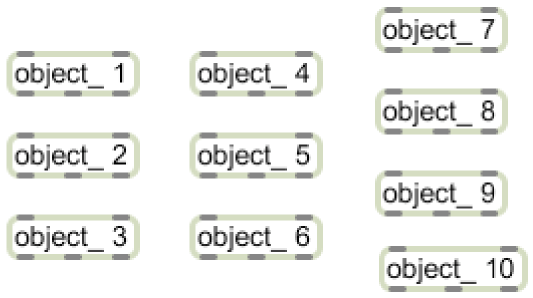

Figure 4.
Even without cords, objects 1 through 6 seem to form one group and 7 through 10 form another.

Use panels to show groupings

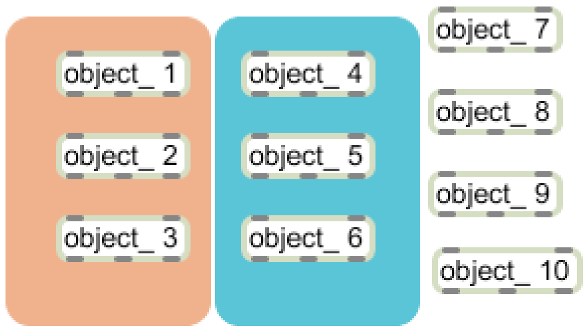

Figure 5.
Use colored panels behind objects to show the relationships. This can also show association between related items that are not adjacent. Pale colors work best.

Use some colored patch cords

Colored cords can help trace important signals, but use colors sparingly. For instance, if I have a lot of actions triggered from a single metro, I might use red for the cords coming of it. Too many colors make the path harder to follow.

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.

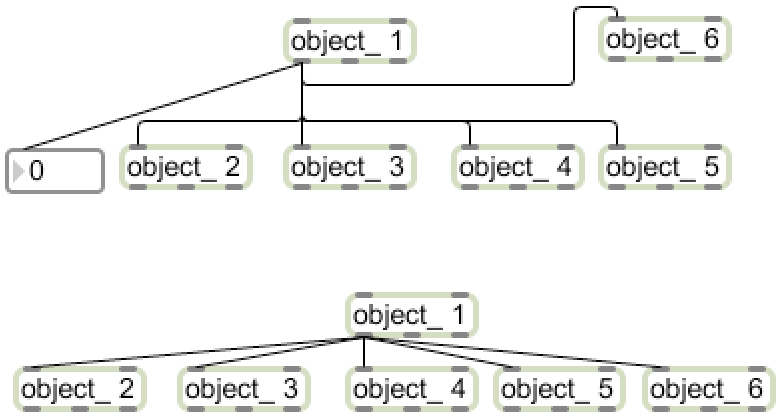

Figure 6.

Don't run cords over objects.

It makes the lettering hard to read, and you can't tell what's connected. It also makes it difficult to select the underlying object.

Make intersections unambiguous.

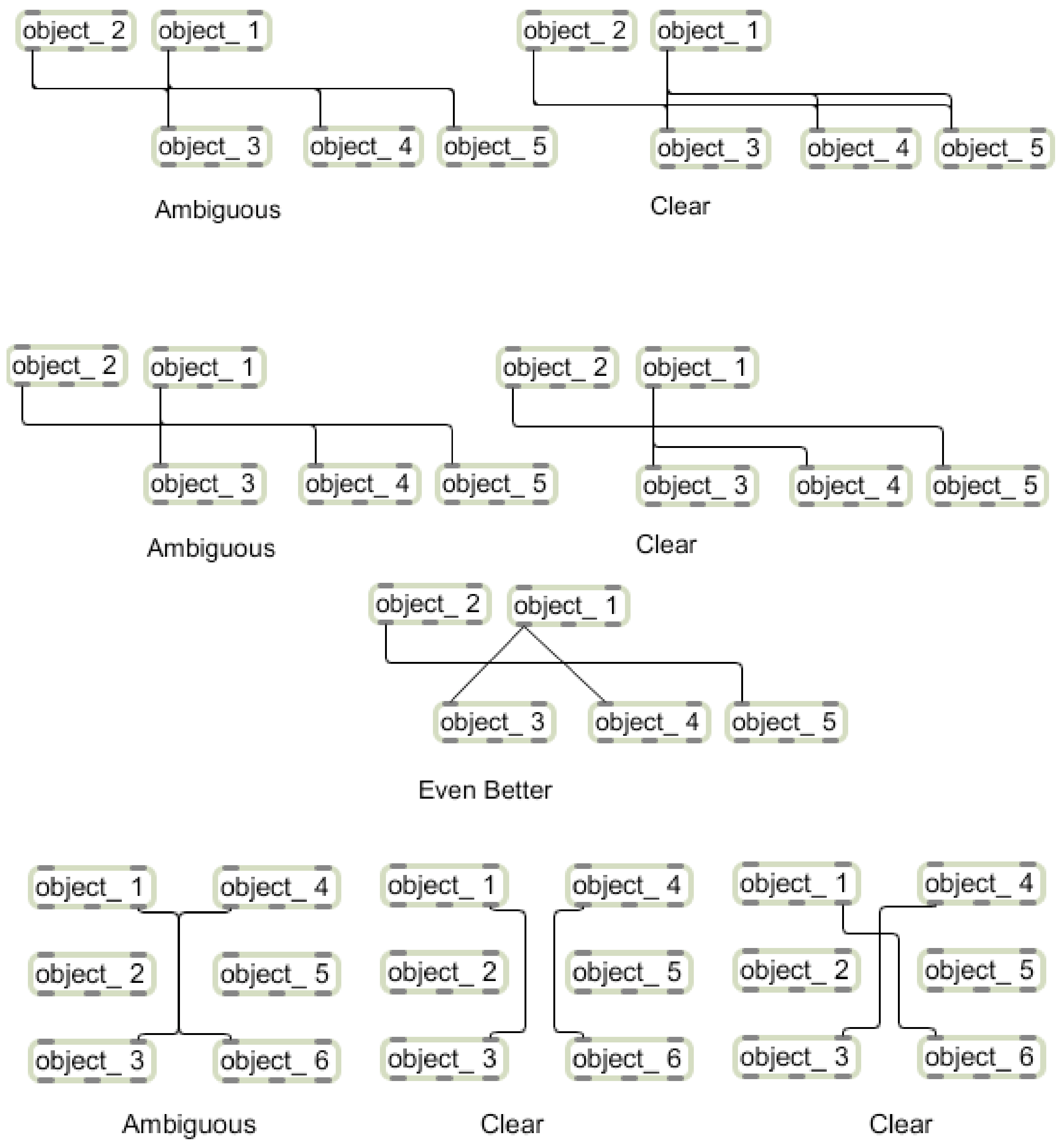

Figure 7.

The curved cord corners introduced in version 5 help a great deal, but cord crossing can still be a source of confusion. Don't stack cords that aren't going to the same place. Figure 7 shows some examples.

Route cords to minimize intersections

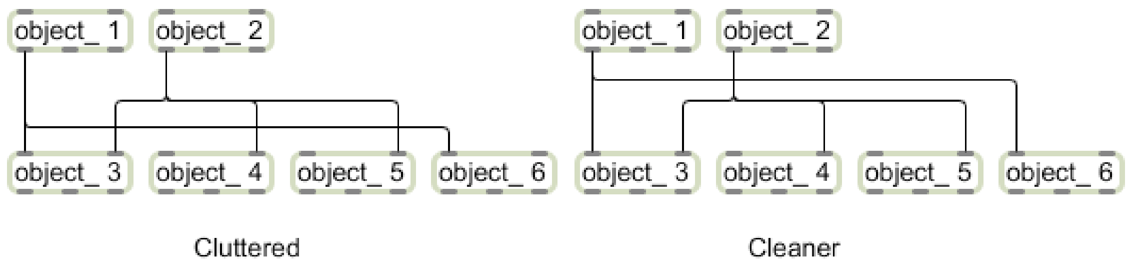

Figure 8.

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 should seldom need more than three corners on a cord.

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.

Patch around number boxes, not through them.

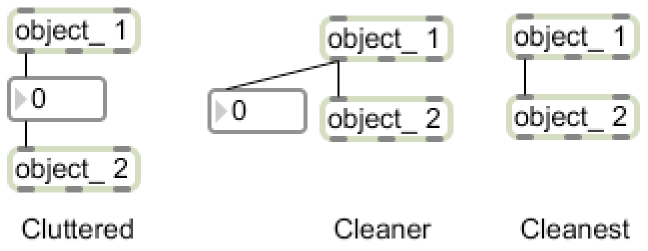

Figure 9.
We usually stick too many number boxes in a patch, especially if we are testing things as we go along. Most number boxes can be removed once they are no longer necessary. This is awkward if we continued patching by taking the output of the number rather than the number that feeds it.

Never use two objects where one will do.

Don't send a message to two instances of the same object.
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.)