Complex Patches

Once you start writing relatively complicated programs, try to build them out of different parts, rather than one enormous, tangled patch in a single Patcher window. The way to do this is to divide your program up into different Patcher files. The different files can be subpatches of one main patch, so that they are all loaded when the main patch is opened.

Subpatches can communicate with each other via inlets and outlets, or via **send** and **receive** objects, and they can share data by using **coll**, **table**, or **value** objects which have the same name as an argument. There is no reason that a large and complicated program cannot be composed of many smaller parts, and the advantages of this approach are considerable.

Modularity

There are several important reasons why it is a good idea to use a modular approach to programming. One reason is that it makes it easier to verify that your program actually works, especially in extreme or unusual cases. This becomes harder and harder to do as a program grows in size and complexity. By building small modules and ensuring that each one works as its supposed to in and of itself, you reduce the number of possible problem spots when the modules are combined in a larger context.

A second reason is that many tasks in a program are used again in different contexts. Once you have built a small module that performs a certain task, you can use that module wherever the need for that task arises, rather than rewriting it each time.

Another reason is that many tasks in a program are similar to other tasks. By writing a small, general-purpose module (usually one that takes arguments so that its exact function can be modified by the argument), you can use that one module with different inputs or arguments, to do many similar things.

Finally, by encapsulating different portions of the program, you make it easier for yourself (or others) to see how the program works long after you develop it.

Encapsulation

The different modules of a program are best designed to encapsulate a single task. Name the module for what it does, and reuse the module should you ever wish to perform the task again in another program.

Encapsulation

By keeping certain values in one place, you only have to change them once if you decide they need to be modified. If the same values are distributed throughout your program, you have to find every instance of that value, and change each one individually.

One way to keep values in a single place, yet still make them available to many different objects is to store the values in a single file that can be accessed by any patch. For example, many different patches can read in values from the same **table** file by using **table** objects with the same filename as an argument. Changing the contents of that one file then changes the values used by all the patches that share that file.

Messages between Patches

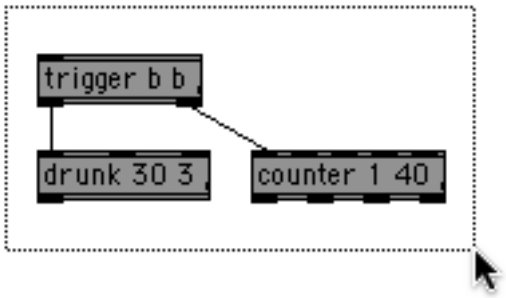
When designing small modules (patches) which will be combined in a larger program, it is important to consider not only what the patch does internally, but also the context in which it will be used. The context will determine what kind of messages you want each patch to produce and accept. For example, you might wish to use a `bang` to trigger a process, numbers to toggle something on and off or to provide values for calculation, or symbols (such as `start` and `stop`) to control a more complex task such as a sequencer.

The simpler the messages that a patch receives and sends, and the simpler the function of each patch, the greater the number of contexts in which that patch is likely to be effective.

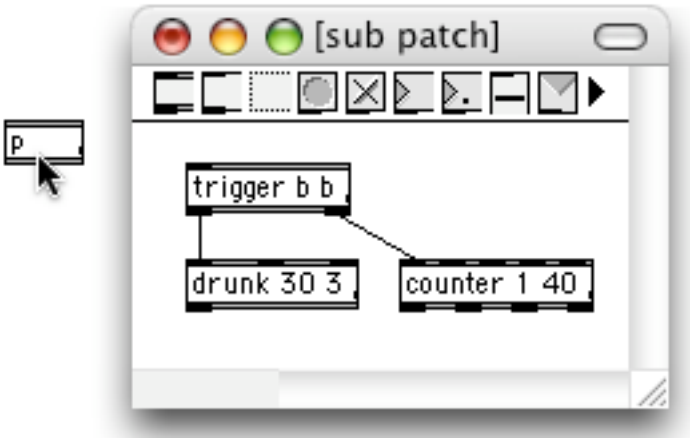
Encapsulation and De-Encapsulation

You can use encapsulation to clean up a patch you are making by putting a group of objects in a subpatcher.

- Simply choose the objects you wish to place in the subpatcher.



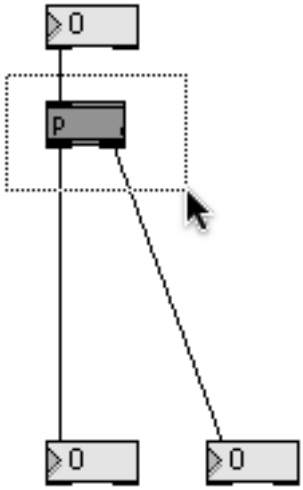
- Then, choose **Encapsulate** from the Edit menu.



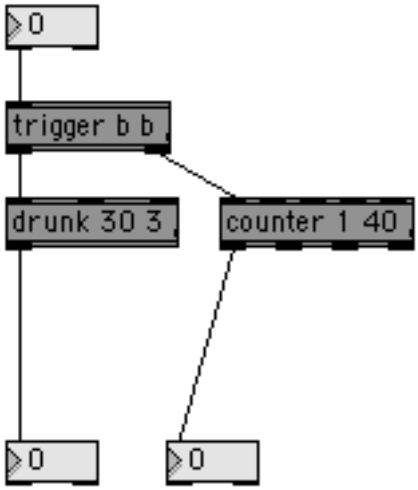
The objects are swept into a newly created subpatcher, inlet and outlet objects are added as appropriate. Don't like what happened? You can undo it.

The inverse operation is also possible. Sometimes objects stuck in a subpatcher are bothersome when trying to manage two windows to keep track of everything. You can now bring objects in a subpatcher "home" to their parent patcher with the *De-encapsulate* feature.

- Select a subpatcher



- Choose **De-encapsulate** from the Edit menu.



The subpatcher disappears and its contents are placed in the parent patcher, preserving any existing connections. De-encapsulation can be undone too.

Documenting Subpatches

Here are three tips for documenting your own patches that will be used as subpatches:

1. Give your subpatches informative names, so you'll remember what each one does.

2. Put Assistance text in each **inlet** and **outlet** object, to remind you of the inlet or outlet's purpose when using the patch.

3. If your subpatch is complicated, include **comment** boxes inside it to explain its operation.

See Also

Debugging	Techniques for debugging patches
Efficiency	Issues of programming style