When the display is subdivided into compartments, will each compartment be a module that contains all its relevant view components, or will each view component be able to display itself across subdivision?

* animations want to transcend compartments
* granularity controls on the display (e.g. deciding when to abstract lines into functions) wants to understand its compartment
* is there a way to define a compartment module that can be mingled into all the many views? does this take the form of a compartment view that they all inherit from, or does it take the form of an external module that they all mix in?

Conclusion: don’t compartmentalize. Calculate compartmenting up front and reference this calculation during updates.

Suggested resources for integrating SFP into a development process (c/o Rob):

* Javascript: Tern JS by Marijn Haverbeke. See <https://github.com/marijnh>, <http://ternjs.net/doc/demo.html>, <http://marijnhaverbeke.nl/blog/tern.html>
* Python: iPython web notebooks

Past filters should be stored in chronological order along with screenshots of the resultant plot and optional names or stars. They should be displayed for easy return to that filter. When the display has been sequentially refined by additional restrictions, escaping the current filter should pop up one level of refinement until all filters have been removed. (Perhaps something like shift-escape would clear all filters immediately.)

One option for displaying filtered-content delineators is to treat them the same way functions are treated, i.e. as simply another composite object used for compressing multiple flow points.

There should be a way to spatially overlay time sections e.g. in order to see looping patterns. This would be similar to spatial overlays of different programs’ flows.

One valuable way to color points will be in terms of control flow – which points refer to returns, which to function calls? I want to know when I’m stepping down, when I’m pulling back up. Maybe (in default mode) make side effects different shapes, like a star for a side effect, and save coloring for things related to control flow.

When generating flow points, users should be able to capture just portions of the flow and should be able to flag specific parts of the flow during which to furthermore capture specific state information.

Control flow isn’t just the order in which things execute, it’s also who told whom to execute. When A calls B and C in series, both of whom do lots of things on their own, it should be clear that C has been called by A. This will be somewhat clear from the fact that an additional line in A executes between B’s and C’s executions, but how can we show this more directly? → Initial thoughts: What about drawing arrows from the point where B and C are both getting called to the points where they start? Grouping B and C in larger sets may also help; what about coloring patterns or other ways of showing that everything has happened within the context of the call to B, and that then lots of things happen within the context of the call to C?

rename library prototype extensions to “mixins”?

example of displaying text at mouse location: <http://bl.ocks.org/mbostock/5247027>

at some point it’s probably worth reconsidering the SVG viewbox attribute: <http://tutorials.jenkov.com/svg/svg-viewport-view-box.html>

Observation: When learning a new codebase (Para) without guidance, I start off understanding how the components fit together. (I tend to use files as proxies for components.) I search for entry points and dependencies. I’m interested in how components interact and how they’re linked together. This takes awhile.

possible steps forward:

* look into pdb python debugger api
  + 1: find documentation, run basic debugger on coins.py, find basic way to interface with it
  + 2: find specific step-through calls, write logic for this
* look for python preprocessing tools that can detect where full lines occur
* look into python compression tools that, without compressing the logic of the code, will reduce the file into exclusively single-line statements
* look into gdb c code debugger, work first with c instead of python
* forge straight on with logging python files, manually handling multi-line statements