Coding Workflow for STM Project

This is the coding workflow that we are advocating for the STM project

Summary

1. Design the routine
2. Break the design into small independent pieces
3. Write the Doxygen description of that piece, what it will require, what it will do, etc
4. Plan for a test for each piece, if desired put empty tests in place in testing code as place holders, empty tests should return a “fail” result
5. Discuss the design plan with Delta Modeling staff
6. When design plan is approved, implement code by writing code/test pairs
7. Complete each code/test pair (Doxygen, code and test) prior to moving on
8. Add analytical tests when routine is complete
9. Revisit the design at anytime as needed

Example for Diffusion code:

1. Design.

Write down the subroutine interfaces for the jobs we need to do. Think through the algorithm for a single step:

subroutine diffuse(conc, area, dx, dt....)

call explicit\_diffusion\_op(diffusion\_flux)

call boundary\_diffusive\_fluxes (diffusion\_flux)

call construct\_diffusion\_matrx(matrix,area)

call construct\_rhs(rhs,diffusion\_flux, other\_stuff)

call solve(conc, matrix, rhs)

end subroutine

Of course you should also specify the smaller subroutines as well. You should include the intent and types of the arguments and documentation.  Commit this, build the Doxygen and we can talk about the implications of the design. The initial design should be close, but doesn't have to be perfect. The main thing is that we can get on the same page at this point without disappointing anyone about code.

2. Plan for tests.

Establish how you are going to test each of the above subroutines. Nicky or Eli can help you if you have questions about what the unusual or "corner cases" are that need to have predictable, testable behavior. The test bodies can be empty and call "add\_failure", but we should get a notification about every step of the algorithm that either isn't tested or fails its test. The things you want to test should be thought out. Once we "sign off" on the interfaces, you guys will have lots of freedom to play around with the implementation for each component.

3. Implement routines and tests in pairs.

Write one routine, test one routine. If you can reuse chunks of code during this step, great,  The last unit test will be a convergence test on a Gaussian plume or other shape... if you have good unit tests for all the subcomponents this will go smoothly.  If you discover things in the system-level convergence test, we will want to add tests at the lower level to cover the issue.