Fluid Component Checklist

This is a proposed checklist for Fluid components. Be warned that this is in a very early state and may change completely.

1. The final completed component has been approved by a designer.
2. There is at least one working example with documentation in the repository.
3. The component has automated javascript unit tests.
4. If the Fluid component has a server side aspect there are automated java unit tests for it.
5. New parts of the Fluid framework have automated unit tests.
6. The component has automated functional tests.

Working examples of the Reorderer

The Reorderer currently has two working examples in the repository.

Lightbox.html contains a working Reorderer presented in the context of an image ordering component. It works as a stand alone example of the Reorderer and contains tests that prove basic working functionality.

unordered-list.html contains a more abstract example of the Reorderer, this time using a different layout handler. Notice the documentation at the top of the file explaining the contract between a template and the Reorderer. This file also contains automated unit tests ensuring that changes in code do not effect the expected functionality of the component.

These examples should be linked to from the wiki component page allowing a designer or developer to easily look at and play with components when selecting components for their page.

Unit testing strategy

Unit tests have many uses and benefits including providing documentation, regression testing, making refactoring easier to do and encouraging a more loosely coupled code design. We likely do not have the resources required to create unit tests that provide full verification of a component. However, some investment in unit tests will benefit the Fluid framework and libraries. Here is a suggested strategy for unit tests:

1. Write unit tests as examples to other developers of how to use the underlying Fluid code. Choose names in tests carefully to communicate clearly to other developers.
2. Keep tests short so that future breakages point directly to the problem.
3. Write unit tests for all common cases. Cover edge cases in your tests as well but balance these with the diminishing returns of the test and the cost of creation.
4. When fixing a bug, write a unit test that proves the bug fix has worked and remains fixed.
5. Ensure changes in implementation do not require a full rewrite of the unit tests.

Functional testing strategy

Functional tests cannot take the place of testing with real users. However, there are some benefits to having functional tests.

What will functional tests do for us?

1. Capture interaction design requirements in code and provide regression for them.
2. Test a solution end to end from the perspective of a user.
3. Ensure less used but important functionality (such as keyboard alternatives) remain in working condition.

How can we implement functional tests?

Use Selenium (or another tool) to write common work-flow tests for components.

Progress, issues and plans.

I've started by trying to write some simple accessibility tests for dojo using Selenium. The basic tests have proven to be easy to write, however there are some issues to be ironed out. Cross platform compatibility for certain keyboard actions has been an issue. For example bringing up the context menu in windows is done with shift-f10 but on the Mac with ctrl-space. The choice is to test at a lower level or write something that checks the platform and performs to correct keystrokes. Keyboard handling in general does not seem complete probably because most people are testing mouse interactions.

I intend to look into Windmill by writing similar tests to determine which tool is better suited to the work that we are doing.

Any other tools that people have used and recommend?

Is this worth doing?