Deliverable 4: Chapter 4

Edit | New Page

Jump to bottom

Jack edited this page 8 days ago · 6 revisions

Improvement through re-implementation

Deliverable 4 saw our unit testing framework tested against, and re-designed to work with, object-based components. We found this necessary because our project has a severe shortage of pure functions; nearly everything in the nightscout repository is, in fact, an object. Unit testing of functions is great and simple; you import the file, pass the function name, pass the value, and it returns an output that can either agree or disagree with your expectations. Object-based unit tests, however, require a few more steps to the process.

The testing scenario for an object goes as such:

1. Initialization: creating an object and giving it an initial state such that an internal function can be tested
2. Testing: calling the unit-test target function on the object
3. Post-processing: in the case of a function that returns an object, running the 'getter' necessary to actually receive the value produced in step 2.

Compare and contrast to the function testing scenario:

1. The whole thing: calling the function, which returns a value

Our original test-case pipeline was not well suited to the new process, as it only supported the one step; the 'middle' part of an O.O. test. Thus, redesigns were required.

Redesign 1

Our first variant involved no changes to the test cases or executable generator at all; it was merely hacking together a functioning test. In our test-case layout, we kept it as

`filename:`
`function name:`
`input:`
`output:`
but made allowances by adding a new parameter: `*args:`

thus, any non-function object calls could be placed like so:

`function name:function`
`input:inval`
`*args:.get()`

which would translate to, roughly: `function(inval).get()`

however, the *args were not an elegant solution when we realized some objects had to be inititialized in particular ways. For instance, how would we parse `functionName.input` vs `functionName(input)` vs `functionName.input()`? Clearly another design was required.

Redesign 2

This one didn't last very long, but it's here anyways. Part of the evolutionary design process is having dead branches to the family tree, after all. We considered that, perhaps, the issue was the type of test being run. For instance, if you're testing a function, it needs the `()`, and if you aren't, it doesn't. Simple, right? With that in mind, our next design was implemented.

`Method being tested:`
`Input:`
`Isobject:`

The idea was that `functionName(input)` could have an isobject of 0, and `functionName.input` could have an isobject of 1. But what about `functionName().input`, the function that returns an object? Or even `function1(input1).function2(input2)`, the function that returns an object of functions? A more dynamic approach was required, and as such this design never got past the theoretical stage.

Redesign 3

The final and current design, therefore, had to account for dynamic number of objects, parameters, and functions, in any order. After all, why would we go through all the effort of making the test cases accept `function().objcall` and not accept `objcall.function()`? We had to go full optional number of parameters. As such, we went with a fully * optional `call` line, and a fully * optional `param` line. The new implementation is like so:

`component:`
`(opt*)call:`
`(opt*)param:`

and can be combined in any permutation necessary. For instance, what if the file import is itself a constant?

`component:constant.js` (no lines following)

-> which generates into `component` when called

What if the import returns an object, the function is behind a subobject and takes no parameters, and the answer is only accessible by getting the 1st index of a method called on that object?

`component:crazyobjects.js`
`call:objmethd`
`call:function`
`param:`
`call:resultmethd[1]`

-> `crazyobjects.objmethod.function().resultmethd[1]`

While this loses some elegance, one must admit that testing this particular case has very little by way of "elegant" solutions. And these are exactly the kind of objects we're dealing with. Take `languages.js`, for example. To get it to translate a word, our test case looks like this:

`Component:language.js`
`param:`
`call:set`
`param:'fr'`
`call:translate`
`param:'Clock'`

which generates into this: `language().set('fr').translate('Clock')`

We hope that y'all agree when we say that, though we admit the redesign lacks the simplicity of a pure-function testing environment, it gains far more in functionality than it loses in elegance. Either way, it was necessary.

+ Add a custom sidebar

Clone this wiki locally

```
https://github.com/csci-362-02-2019/2-2.wiki.git
```