Contractive Delegation

by Henry Kissinger

Well-factored code often has many small functions. If each adds value, and doesn't just pass the buck, then what do they all do? Typically, they contract their input by making it more specific. Then they delegate these specific data to a delegatee. For example, this function takes a `fileName`, augments it with the current document, then passes them both into the File system delegatee:

```
def save(fileName)
    string = get_chars(0, get_length())
    File.open(fileName, "w").write(string)
end
```

Every short method solves one part of the problem, and delegates the remaining part of the problem to other methods. Every method transforms its input before passing it to other methods, and/or transforms the output of other methods before returning to the caller.

Refactoring a big long ugly function—past the Method Object Refactor—can lead to many small delegates. Refactoring these to remove duplicated delegation can lead to Contractive Delegation.

After implementing a complex feature, and after the first round of refactoring, the design might contain many absurd chains of delegation, where A calls B calls C repeatedly. The second refactoring phase isolates and removes the intermediate B methods. This level of abstraction forces special cases to isolate from each other efficiently. Refactoring to merge behaviors forces structures to flex.

Each delegating method accepts its inputs, contracts them by adding this method's special ingredient, then passes the inputs to the next delegate. New features are very easy to add to code following Contractive Delegation. Code following these patterns can be a little difficult to read. No more 30-line methods with obvious procedures.

End a refactoring session with Rename Method Refactors. Name things after their new intentions.

If you see duplication, but can't imagine how to improve its design without obfuscating what it does (or can't imagine any way at all), move all the duplicating lines next to each other. This practice forms little tables, with columns that are easy to document and scan.

Early refactors often stabilize a design, enabling new features of the same kind without refactors. Suppose you add two abilities, and they generate similar statements. Merge duplication into an abstraction just as soon as two abilities require it. The odds that future abilities will reuse that abstraction are now very high, because applications by nature present users with lists of similar options. This simple technique prevents code from becoming increasingly disordered and hard to change.

Good designs approach the "Open Closed Principle", which essentially means (in an organic design context), "Reuse me the way others have reused me. Don't reuse me by typing on me, adding '`if`' statements for special cases, etc. Reuse me by adding to one of my lists—metadata, derived classes, clients, plugged-in classes, etc. I am Open for re-use but Closed to edits."

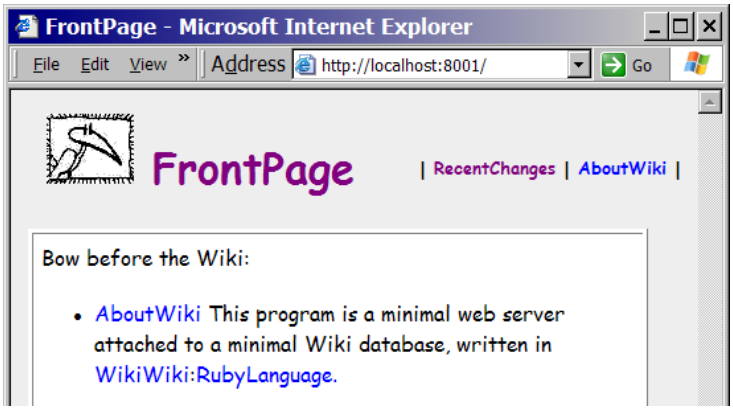
Refactor tests too, but follow slightly different rules. Test cases will duplicate much trivial behavior on purpose, to self-document. Merge such duplication to improve readability, and to create fixtures that make new tests easier to write.

Abstract Tests

MiniRubyWiki (MRW) uses a variation of the practice "Version with Skins", so its tests constrain its behavior under two different kinds of servers.

`http://minirubywiki.rubyforge.org/`

A "Wiki Wiki" is a dynamic Web site that looks like this:



When you click the big purple "FrontPage", the server searches for that string in its Web site, and returns an XHTML page containing links to all pages containing the text "FrontPage".

The module `miniServer.rb` implements a minimal Web server based on a raw `socket` connection. It pulls strings directly out of a TCP/IP connection and parses their variables, obeying only the most important details of the HTTP recommendations. This efficiently bypasses many features that "real" Web servers enforce. MRW can serve on port `8001`, to avoid other servers.

Another skin, `wiki.rb`, bonds with real Web servers via the traditional Common Gateway Interface. I test that with Apache `httpd`, serving on the default HTTP port, `80`.

Both servers front the same Wiki features, so MRW's test rig defines the tests once, and calls them twice, following the Abstract Template Pattern.

The base test suite wraps Internet Explorer, to fetch and interpret pages. All tests follow the same general pattern. They `surf()` to a specific URL, return a page, find some text on it, and compare that text to a reference:

```
        class TestViaInternetExplorer < Test::Unit::TestCase
    …
            def test_defaultToFrontPage()
                surf('')

                regexp =
        /href="#{getPrefix()}SearchPage(&amp;|\?)search=FrontPage"/

                assert_match(regexp, getHTML())
            end
    …
        end
```

That case reuses fixtures, so it's short, inscrutable, and only obvious if you know that `@ie` is an unseen member variable referring to an instance of Internet Explorer, and `surf()` navigates it to a Wiki page.

`surf()` wears two concrete skins:

```
class TestMiniServer < TestViaInternetExplorer
    def getPrefix()
            return ''
        end
…
    def surf(page)
        href = "http://localhost:8001/#{page}"
        navigate(href)
    end
…
end

class TestApache < TestViaInternetExplorer
…
    def getPrefix()
        return 'wiki.rb\?'
    end

    def surf(page)
        href = "http://localhost/wiki.rb?#{page}"
        navigate(href)
    end
…
end
```

`TestMiniServer#surf()` heads for `http://localhost:8001/`, and `TestApache#surf()` hits `http://localhost/wiki.rb?`. The default Wiki page is `FrontPage`, so `surf('')` is the equivalent of `surf('FrontPage')`. The example test ensures both `miniServer.rb` and `wiki.rb` return a page with a hyperlink to the search system containing "`FrontPage`".

The fulcrum of the test is the overridden method `getPrefix()`. It returns either an empty string or `'wiki.rb\?'`. When `getHTML()` queries IE for the HTML contents of its page, our Regular Expression Match then determines that our page has a link on it that searches all Wiki pages for "`FrontPage`". That's the FrontPage's title's behavior when you click on it, so only only `miniServer.rb`'s FrontPage will contain this:

```
<a href="SearchPage&amp;search=FrontPage" title=
"Click to search for this string">FrontPage</a>
```

But the Apache's FrontPage contains a longer `href` value:

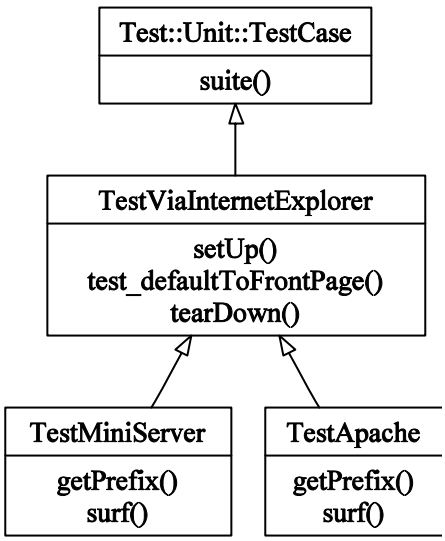
```
<a href="wiki.rb?SearchPage&amp;search=FrontPage" title=
"Click to search for this string">FrontPage</a>
```

To detect that difference, the test's `assert_match()` calls `getPrefix()`, nestled inside its Regular Expression Match:

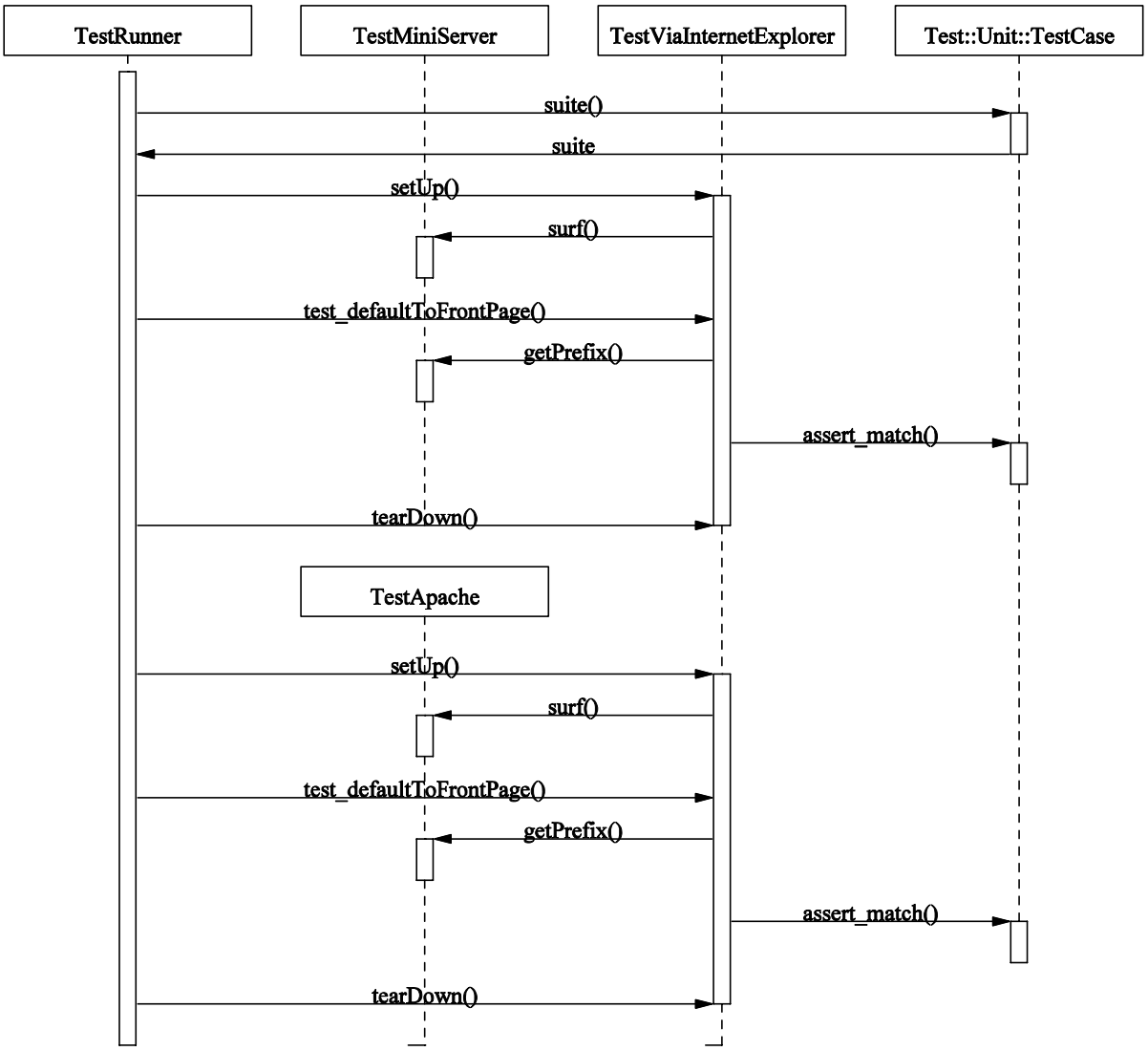
```
        regexp =
/href="#{getPrefix()}SearchPage(&amp;|\?)search=FrontPage"/

        assert_match(regexp, getHTML())
```

Ruby embeds executable statements inside strings or regular expressions, escaped with `#{}`, so `#{getPrefix()}` introduces different search string details for each skins.

This test program's `main()` function adds `TestMiniServer.suite()` and `TestApache.suite()` to its list of test cases, so both concrete suites express the same shared tests, including `test_defaultToFrontPage()`:



A TestRunner kicks off the game by calling `Test::Unit::TestCase::suite()`, to fetch a global list of cases. For each case, it calls `setUp()`, the case, and `tearDown()`. Virtual dispatch permits `setUp()` to call a concrete implementation of `surf()`, in one of the derived classes. Then `test_defaultToFrontPage()` calls a concrete implementation of `getPrefix()`, again in one of the derived classes:



That sequence diagram illustrates Contractive Delegation. `TestRunner` and `TestCase` are completely generic. `TestMiniServer` and `TestApache` are specific, and `TestViaInternetExplorer` is partially generic. Method calls going to the right become more generic. The parameters to those methods are specific—they contract their delegates. Method calls to the left dispatch virtually into concrete test classes, and fetch data out for those parameters.

Sharing a test suite between many skins (or versions or flavors) helps engineers think about only one skin without accidentally breaking another one.