Bestiary of Reuse Failures

On this page we will curate a growing bestiary of examples of real-world reuse failure, caused by the use of traditional development techniques - including, but not limited to, the use of "information hiding", "encapsulation" and the structuring of large-scale artifacts using programming language code.

Vega

Vega is a highly interesting, declaratively structured JSON grammar for authoring data visualisations. Its goals are admirable and its development staff are talented and industrious. Vega required a built-in expression language in which users will author calculations and live data bindings between elements of a visualisation design. This expression language was specified to be "a restricted subset of JavaScript", which is a perfectly reasonable choice. However, the implementation strategy chosen by the developers was to take an existing, fully-featured JavaScript parser, esprima, fork it and hack off the parts of the parser that were not required. The result is a 1500-line stovepipe, parser.js, within the vega-expression package, which can never conceivably be merged back with the upstream implementation. Note that, given the computer science community has been working on parsers for 50-odd years, one would expect this to be a solved problem, and that such a parser would be generated using one of the many off-the-shelf parser generators such as jison, peg.js, etc. However, these generators invariably are so hard to use, and produce such bloated, unmanageable parser code that it was a perfectly reasonable decision for the esprima developers to write their own by hand. In any case, the use of any of the parser generators would have had virtually no impact on the reusability values of the resulting parser, since the idea that grammars might form a composable ecology is simply treated as a "research problem" - that is, researchers will produce one-off demonstrations of their own cleverness in this area and then quickly move onto another problem before their publication record suffers. Noone considers it their business to curate a long-term project hosting reusable, portable composable parser generators together with all the tool chain they require. Here's a ycombinator thread on the subject.

The node.js "internal" debacle

A well-respected member of the node.js community, isaacs, wrote a widely-used library, graceful-fs, as an improvement in portability and resilience over the built-in node.js filesystem handling library, fs. In the course of doing so, he discovered that he needed to reuse the module source for fs in a particular way which suddenly broke when node.js was updated to the 6.x version, since the developers had decided to follow "industry best practices" by hiding the module sources for built-in libraries in a freshly created package, "internal" that was invisible to userland code. The author started the following thread as node issue #8149, asking the core node developers to explain the benefits of this encapsulation approach. The thread makes highly interesting reading, since the points of view are very clearly articulated on each side. There is a clear division between the "elites" on the core team who stick to their abstract best practices, regardless of what evidence is produced by the "ordinary users" on the thread that this encapsulation simply interferes with their ability to get value from the node.js codebase. isaacs also interestingly draws attention to what he sees as the "drift" in the core values (both articulated, and implicit) held by the core node.js team (with whom he has been familiar since the project start) between the 0.x/1.x days and the io.js/4.x days. He states that a core value held by the former team was that "the user should not be protected from themselves". Some commenters suggest that this drift in values could be explained as a result of numerous, unskilled users joining the node.js community who are more likely to need protecting from themselves. I am unconvinced, and instead see this as a very traditional "senescence" phase of a project, where the core team comes to attract dogmatists and rule-followers, after the initial phase where the team was built of enthusiasts who simply wanted to make things that worked, and did not see a fundamental distinction between themselves and their users.

nyc

This invaluable and current coverage checking tool depends on a tower of libraries to select and traverse the files requiring instrumentation. In the GPII, we are following what has come to be called the "Alle proposal" for operating micromodules, which involves the use of an internal node_modules directory holding the module root. It turns out that one of nyc's dependencies, "test-exclude", includes a hard-wired regex check for node_modules anywhere in the path of a file to be instrumented, as follows: index.js#L35. As usual, this requires forking of the entire tree of dependencies to fix a problem which is hidden in a nested closure.

Unifying input device tracking and browser events

The DOM event model does not associate user input events with their source device, only with the abstract type of device and the 'target' DOM elements. This presents a design challenge for multi-user or bimanual interfaces that may, e.g., want to track multiple mice, with each their associated cursor and behavior. ptchernavskij has attempted to get around this by using node-usb and node-hid on a locally-running server to identify devices and forward their state to the browser, thus maintaining models of actual input devices. However, at this point, there is a difficult design trade-off:

1. emit only "in-house" events generated by the device server. This requires significant re-implementation of processes such as acquiring targets of pointing events.
2. adopt DOM events and forget about decorating them with source devices. This strongly limits the possible bimanual interface designs.
3. combine DOM events with device information. This requires some sort of asynchronous event synthesis module that reliably matches DOM events and "in-house" events.
4. use both kinds of events without attempting to match them up. This appears to be the most "clumsy" design, but at least makes the desired designs possible.

This is an interesting reuse problem because, in addition to the traditional closed-ness of the DOM event system, it is particularly difficult to coordinate two asynchronous information sources.

AOFF's website

Bødker and colleagues have studied a volunteer community working to distribute locally sourced organic vegetables in Aarhus, Denmark (AOFF). Their paper contains many stories of how a grassroots organizations design their technologies in real life. AOFF started out using a facebook group both as their public face and means of communication, but subsequently members of the group operated a wiki and several different websites. The first of these websites was created and hosted by a web developer volunteer, who eventually abandoned it: "the initial web developer became less and less involved with AOFF, resulting in minimal development, slow communication and lack of access to the basic configuration on the back-end, forcing the community to "invent" alternatives around the website." While the abandoned website was still in use, another member with development skills implemented a calendar feature on the website essentially by hacking it: "Paul, the member who later went on to develop the second website, "*went into the database and put in an iframe as a content element...that's not done through the CMS at all, that's just some injected some SQL, into the database, which case the calendar feature [...] but I mean, that's what we had, that's what we could do, it's the only possibility*".