Introduction - Structuring Asynchronous Components

[This essay builds upon explanations given in a previous essay - https://github.com/guitarvydas/ag-js/blob/master/HTML%20Components%20-%20Arrowgrams.pdf]

In this essay, I discuss how to pull the callback logic out of the "part2" component.

In essence, I am describing how I *structure* asynchronous components.

Callback.js

Refer to the diagram `drawing.drawio`.

I've given better - more descriptive - names to the components.

I've renamed "part1" to "FileSelector".

I've renamed "part2" to "Display".

I've created a new JavaScript file called `callbacklogic.js` and have inserted a 3rd component in between the FileSelector and the Display components.

I've moved the callback logic from Display (part2) into the new component CallbackLogic. The code now resides in the `react()` function of CallbackLogic.

Interface Between HTML Events and AG

CallbackLogic is, now, the interface between FileReader() events - HTML +

JavaScript - and AG events.

The initial call to CallbackLogic is done by the FileSelector component. It sends an AG event to the CallbackLogic component.

In that first call to `react()`, the code sets up JS callbacks for `onload`, `onerror`, `onabort` and a timer. Then it starts the ball rolling by calling `readAsText().`

After that initial call, all subsequent events are AG events.

After fielding the HTML+JS events, CallbackLogic makes calls to `kernel.send()` (and `kernel.io()`) to propagate events further into the system. In this example, the events from CallbackLogic flow into one of 4 downstream components - `Display`, `ErrorHandler`, `AbortHandler` and `NoResponseHandler`.

The *happy path* `send()s` a pair of items to Display - the filename and the file contents. CallbackLogic handles the unpacking of the HTML+JS event data.

Event Interface & Conversion

HTML + JavaScript cause one of 4 events - `onload`, `onerror`, `onabort`, `timer`.

These HTML + JavaScript events are fielded by the CallbackLogic component.

For each HTML + JavaScript event, the component re-calls `this.react()` with a pin tag. The pin tags correspond to the pins shown on the diagram `drawing.drawio`.

CustomEvent()

The blog article https://javascript.info/dispatch-events discusses the use of HTML+JS CustomEvent()s.

This article points out one crucial problem with the design of CustomEvent().

Events sent from within an event handler are *synchronous* - they happen immediately, they are sent in a *depth-first* manner.

All other events, though, are processed differently - in a *breadth-first* manner.

There are two ways that events are handled, where there should only be one way.

In addition, the event handling is *implicit*. The onus is on the programmer to figure out what will happen. This is an unnecessary implementation detail that interferes with design-flow.

I strongly recommend that all events be dispatched in a *breadth-first* manner, as this promotes the use of the *concurrent* programming paradigm and prevents future problems similar to priority-inversion.

The article explains how to dispatch events in a *breadth-first* manner - using a 0-time timeout. This is counter-intuitive, but could be hidden behind a well-chosen SCL[1] syntax.

Structuring Asynchronous Activity

The above implicitly shows how to structure event handlers - *make all events arrive at the same place*.

In fact, this is what *Structured Programming* recommended. 1 entry point and 1 exit point.

[1] Solution Centric Language (kind of like a DSL).

This allows us to view the full gamut of events together, in one piece of code. This structure retains one's sanity and allows any state handling to occur in one place, instead of being spread out across many pieces of code. This kind of funnel also benefits from being arranged in a hierarchical manner - viewing and trying to understand 7±2 events is about the maximum for any one person. A flat space of anything is a poor choice - hierarchy is a better choice.