# Markup-driven polymorphism
## A little journey
## (Work in Progress)

**Antranig Basman**
Raising the Floor - International
amb26@ponder.org.uk

**Abstract**
This paper describes a little journey through an interesting design space.

As the author sat as a humble contractor in Cambridge University's "Centre for Applied Research in Educational Technologies" in 2005, it struck him how useful it would be if people who had the capability to edit markup, but who were not necessarily very technical, could use that capability as far as possible to edit entire applications.

This started a journey through many technologies and revisions of worldview, a whiteboard in Boulder in 2009, a realisation that there was not in fact any useful number of people in the target audience, and on finally implementing the capability (again) 16 years later, a realisation that everything had been designed wrongly and must be rebuilt from the start.

## 1. Experiences of End-User Programming

Starting in 2005, I was employed for a few years at Cambridge University's "Centre for Applied Research in Educational Technologies" (CARET), where I came to have many instructive experiences about how software, and the process of building it, manifests in the lives of people who do not have very much money. Our job was to help staff around the University with the use of technology in their teaching, but whose requirements were too diverse or poorly characterised to benefit from the off-the-peg solutions available from the central computing services. Something came into greater focus for me, that had already been a latent idea for many years, that the process of building software was utterly unsustainable, and was only made to appear affordable through the capitalistic processes which the already dominant large corporations had accumulated around this mode of production.

CARET by this point had already accumulated an unreasonable amount of technical debt, both abstract and concrete — the latter, for example, in the server room which at that point consisted of a room whose floor was littered with various mismatched desktop-grade computers of different colours and vintages, some plastered with post-it notes reading "DO NOT TURN OFF" and some, as I learned, whose purpose had already been forgotten. But the abstract debt was far more worrying — applications whose function had never been terribly well characterised written in forgotten frameworks and languages, as well as essentially the same function being delivered in multiple incommensurable forms.

It was quite clear that if we had to meet the needs of our teaching staff by writing programming language code, our mission was unsustainable — there would always be too few of us, and the results of our work would always be too brittle and incomprehensible, even to ourselves. This kind of realisation, of course, was already well established at CARET — before my arrival there had already been (failed) experimentation with "declarative" programming techniques such as Apache's Cocoon, based on XSLT pipelines, and slightly more durable use of markup templating systems such as Java Server Pages (JSPs). This realisation, indeed, has been solidly present in the industry before and since — leading to current fashions such as the "no-code" movement [1] and its slightly more realistic successor, the "low-code" movement.

But what is very clear is that no single idea or combination of ideas has made substantial inroads on this problem in the decades it has been characterised, and we are arguably little closer to being able to build sustainable and affordable systems than we were in 2005 or even in 1985. As I see it, the problems lie at the heart of how we conceive of computational systems, describe their activities, and our relation to them, and will not be solved without decades of sustained effort at conceiving and creating wholesale alternatives.

During my time at CARET, several seeds of ideas started to germinate about how to make progress on these issues, and this paper describes the evolution of one of them. As it turns out, my original characterisation of this issue was partial, addressed an audience that largely did not exist, and was ultimately solved by the industry in an unrelated way — but the core of the idea is still valid and continues to be developed.

## 1.1. The FlowTalk System

Early in my time at CARET I was commissioned to write an application which in retrospect was absurdly ambitious. Across many of the environments we supported, a common requirement had emerged which was being met in unrelated and hard to support ways — attaching threads of commentary to different resources or activities, which would allow the communities interested in the resource to keep in touch for short-term or long-term conversations. The idea was to unify all of the different implementations under a common system, which would expose different "personalities", appearing as a native part of each environment. The previous attempt at such a system, "LETSTalk", based on Apache Cocoon XSLT pipelines, had just been retired before I arrived. The planned new discussion engine would feature

- Integration with the user account and authorization system of the host platform in which it exposed a "personality"
- Incoming and outgoing integration with email - as well as using the various in-place web UIs, one could receive email notifications of replies and dispatch messages back into the system by replying to the email without needing to revisit the web UI
- A fine-grained workflow and permissions system, allowing for per-site customisable moderation workflows and various categories of message visibility
- A completely customisable "look and feel" so that the discussion engine appeared identical to a native tool in each host platform

The 3rd requirement gave rise to the new engine's name, "FlowTalk", but it is the 4th requirement that will be most interesting to us here.

Figure 1 shows the FlowTalk interface customised for two different environments. At the time, and still today, the primary means available to non-developers of customising a user interface is via CSS, but many of the customisations shown there go beyond what was possible with CSS then in 2007, and a few are still out of reach of today's much more powerful CSS. A defining moment was when my manager showed that in one environment, the attribution should precede the message in the email style ("From: John Norman") whereas in another it should follow the message ("Posted by John Norman on 15 June 2002")[1]. I thought — how can the power to adjust an interface in this way be given to those who just have the tools to edit markup, rather than needing to consult a developer?

This kind of requirement can retrospectively be considered part of the movement towards ownable and "malleable" software described in Basman and Tchernavskij (2018). At the time, it was hard to align with any philosophic or technical tradition, but seemed sufficiently radical that it motivated the development of a new programming framework, named RSF[2] ("Reasonable Server Faces", by extension from Sun's JSF, Java Server Faces). I worked on RSF between 2005 and 2008, at which point it became clear that a server-side Java framework was the wrong vehicle to move forward the research programme towards ownable software.

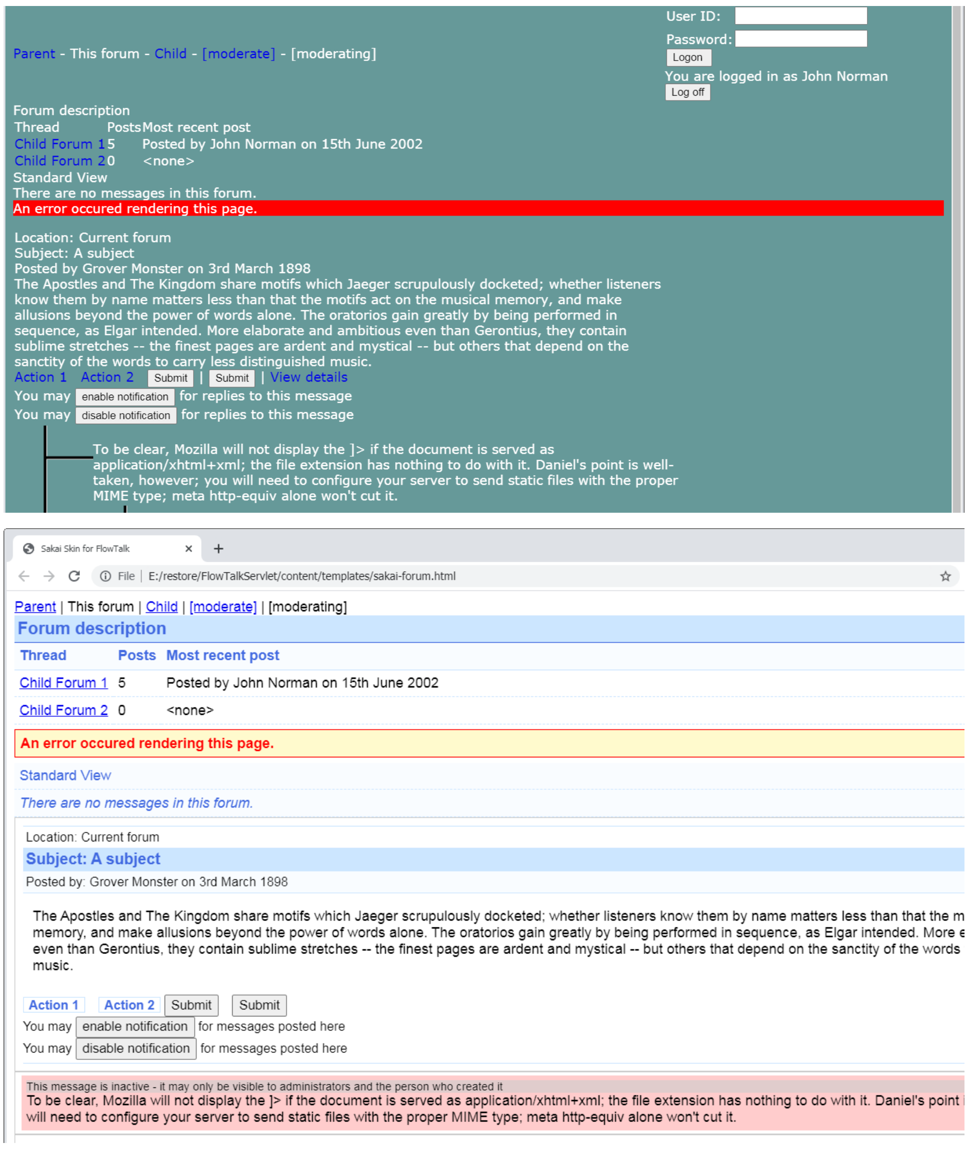
FlowTalk was successfully implemented and put into limited production in 2008, but fairly quickly withdrawn as it became apparent that its configuration and maintenance costs were essentially unbearable. 13 years later, such a thing is still beyond the capabilities of our industry's technology. The closest approximations are systems like Disqus[3] and IntenseDebate[4] which provide some small subset of its capabilities and are highly non-ownable and non-customisable. Users of Disqus who are happy to up-

[1]This design decision was later rescinded

[2]RSF's wiki has been lovingly scraped and archived from its JSP original into GitHub pages at https://rsf.github.io/wiki/Wikib2ab.html

[3]`https://disqus.com/`

[4]`http://intensedebate.com/`



User ID:
Password:
Logon
You are logged in as John Norman
Log off

Parent - This forum - Child - [moderate] - [moderating]

Forum description

| Thread | Posts | Most recent post |
|---|---|---|
| Child Forum 1 | 5 | Posted by John Norman on 15th June 2002 |
| Child Forum 2 | 0 | <none> |

Standard View
There are no messages in this forum.
An error occured rendering this page.

Location: Current forum
Subject: A subject
Posted by Grover Monster on 3rd March 1898
The Apostles and The Kingdom share motifs which Jaeger scrupulously docketed; whether listeners know them by name matters less than that the motifs act on the musical memory, and make allusions beyond the power of words alone. The oratorios gain greatly by being performed in sequence, as Elgar intended. More elaborate and ambitious even than Gerontius, they contain sublime stretches -- the finest pages are ardent and mystical -- but others that depend on the sanctity of the words to carry less distinguished music.
Action 1 Action 2 Submit | Submit | View details
You may enable notification for replies to this message
You may disable notification for replies to this message

To be clear, Mozilla will not display the ]> if the document is served as application/xhtml+xml; the file extension has nothing to do with it. Daniel's point is well-taken, however; you will need to configure your server to send static files with the proper MIME type; meta http-equiv alone won't cut it.

Sakai Skin for FlowTalk
File | E:/restore/FlowTalkServlet/content/templates/sakai-forum.html

Parent | This forum | Child | [moderate] | [moderating]

Forum description

| Thread | Posts | Most recent post |
|---|---|---|
| Child Forum 1 | 5 | Posted by John Norman on 15th June 2002 |
| Child Forum 2 | 0 | <none> |

An error occured rendering this page.

Standard View

*There are no messages in this forum.*

Location: Current forum
Subject: A subject
Posted by: Grover Monster on 3rd March 1898

The Apostles and The Kingdom share motifs which Jaeger scrupulously docketed; whether listeners know them by name matters less than that the m
memory, and make allusions beyond the power of words alone. The oratorios gain greatly by being performed in sequence, as Elgar intended. More e
even than Gerontius, they contain sublime stretches -- the finest pages are ardent and mystical -- but others that depend on the sanctity of the words
music.

Action 1 Action 2 Submit Submit
You may enable notification for messages posted here
You may disable notification for messages posted here

This message is inactive - it may only be visible to administrators and the person who created it
To be clear, Mozilla will not display the ]> if the document is served as application/xhtml+xml; the file extension has nothing to do with it. Daniel's point i
will need to configure your server to send static files with the proper MIME type; meta http-equiv alone won't cut it.

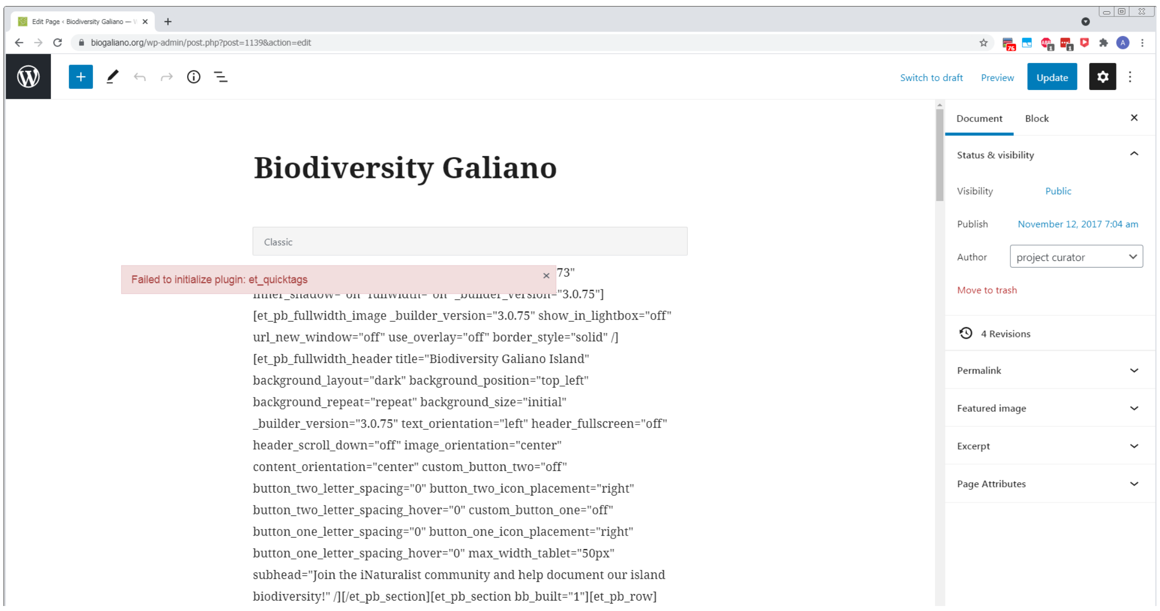
*Figure 1 – Two different "personalities" for the FlowTalk system, which simply take the form of static HTML files, with an element for every possible user interface configuration*

grade to their $105/month "Pro" plan gain access to very limited customisations (font size, disabling branding, etc.[5]), and have to consent to their data being stored in a central silo by a company whose data policies are frequently found in breach of the EU's GDPR. There is nothing resembling FlowTalk's workflow and permissions system, and the discussion interface is still rendered in an iframe, the "poor man's transclusion" mechanism of HTML with its well-known accessibility limitations.

## 2. Reversing information flow

What is it about the way that a web interface (and more widely, all of our software) is constructed, and, more widely, that makes the appropriation and reuse required by platforms like FlowTalk so incredibly hard? A central problem is that all of our techniques appeal to a unidirectional, authoritarian dataflow. Information flows into the system at the top in a highly structured form, is successively processed by irreversible functions[6], and finally ends up as markup which gets rendered on the user's screen.

[5]https://help.disqus.com/en/articles/1717201-disqus-appearance-customizations
[6]I problematise more widely the use of such functions as our central programming structuring device in (Basman, 2017)



# Biodiversity Galiano

73"
inner_shadow="off" fullwidth="off" _builder_version="3.0.75"]
[et_pb_fullwidth_image _builder_version="3.0.75" show_in_lightbox="off"
url_new_window="off" use_overlay="off" border_style="solid" /]
[et_pb_fullwidth_header title="Biodiversity Galiano Island"
background_layout="dark" background_position="top_left"
background_repeat="repeat" background_size="initial"
_builder_version="3.0.75" text_orientation="left" header_fullscreen="off"
header_scroll_down="off" image_orientation="center"
content_orientation="center" custom_button_two="off"
button_two_letter_spacing="0" button_two_icon_placement="right"
button_two_letter_spacing_hover="0" custom_button_one="off"
button_one_letter_spacing="0" button_one_icon_placement="right"
button_one_letter_spacing_hover="0" max_width_tablet="50px"
subhead="Join the iNaturalist community and help document our island
biodiversity!" /][/et_pb_section][et_pb_section bb_built="1"][et_pb_row]

*Figure 2 – Interface showing an attempt to use the built-in editor on the PHP system WordPress to edit a page that has been apparently reauthored by some other plugin — we have no idea how to recover from this situation*

We can study a microcosm of this issue by considering what it would take to reverse a small part of the application's workflow. What if we could take some effective control over an application by editing something resembling its markup? This notion, in 2005-2008, was motivated by the still-lingering use of pure markup tools such as DreamWeaver (since acquired by Adobe) and FrontPage (acquired by Microsoft) which allowed moderately technical users to manage the content of their websites as directly managed HTML. These tools were somewhat unfriendly and made excessive demands on the user's power of abstraction, but at least allowed the guarantee that the user's content could not "get away from under them". An example of the kind of unrecoverable failure that's possible with tools that work on irreversibly abstracted representations can be seen in Figure 2, showing the editor view of a page managed by the very popular PHP website builder, WordPress. The page, whilst it renders fine to the user, appears in the editor as an unreadable jumble of some unfamiliar markup intermediate. As far as we can tell, this site is now lost to us, and will have to be rebuilt from scratch by scraping the markup from the front end and copying it back into some other editing system. If the system will not reverse its workflow, the user will be obliged to do it themselves, following Lialina's "Turing Complete User" 2012.

Reversibility of our computing constructs is already widely recognised in the industry as an important and relevant problem. For example, (Mayer et al., 2018) presents a powerful algorithm, "Bidirectional Evaluation", capable of reversing the action of a small functional programming language in producing markup. However, such systems treat the symptoms rather than the disease — if our dominant construction idiom is a functional one which effaces data provenance, there will never be sufficient context in the design to safely and comprehensibly figure out how to interpret a particular user gesture trying to edit the application's interface. Trying to work with our existing technology stack in this way[7] is going to offer low returns, as situations like the ones shown in Figure 2 demonstrate — if our technology stack is so tottering and fragile that it can't even recognise its own output when operated via its ordinary workflow, how can we expect such an environment to remain stable if it is in addition being adjusted by an automated algorithm reversing its logic? Reversing general functional programs is an endeavour similar to "extracting the sunlight from cucumbers" (Swift, 1726, Part III, Chapter 5).

[7]as some following work of the authors of (Mayer et al., 2018) does, e.g. with startups such as `https://tharzen.com/`

Before moving on however, I should note that I think the notion of reversing the direction of evaluation of what are textually written out as conventional functions is highly promising, *assuming that the functions are very short and restricted*. For a function written as $C => 9C/5 + 32$ it is clear to all readers that it is invertible and what the inverse is, and it is possible to characterise and invert such functions very cheaply. Organising the framework around such "small, good functions"[8] I expect to give rise to an easy-to-parse extension of JSON (and subset of JavaScript), similar to JSON5, codenamed R-SON.

## 2.1. What might a solution look like?

A more reciprocal stack of technologies might enable more influences from other sides of the design and from other kinds of participant. But such a "new stack" faces the problem of growing up amidst, and competing with, very established solutions to the same problems. Figure 3 shows a visual metaphor laying out the different levels of our current technology stack as barriers as if around a prison camp, preventing influences travelling against the established flow (and in general, by their very complexity and numerousness, preventing any kind of influence on the design by non-specialists). A common response to frustration with part of the technology stack is to imagine a replacement that mitigates some, usually technocratic, concern — the web itself most frequently comes in for criticism of this type, despite its arguably representing the most malleable layer of the stack. I reproduce in Figure 4 a Twitter conversation amongst some capable developers in which the web is imagined replaced by a "reliable UI API". These attempts don't solve the problem that we have, which is to assist citizens to work with and integrate our existing technologies, and in the unlikely event they were successful would only reproduce the same authoritarian workflows in a new form. In fact, they would only intensify existing integration problems since the current web represents one of the most malleable systems in widespread use — a fact exploited by interesting systems such as Webstrates (Klokmose et al., 2015) which treat the browser's DOM as the primary representation of the application.

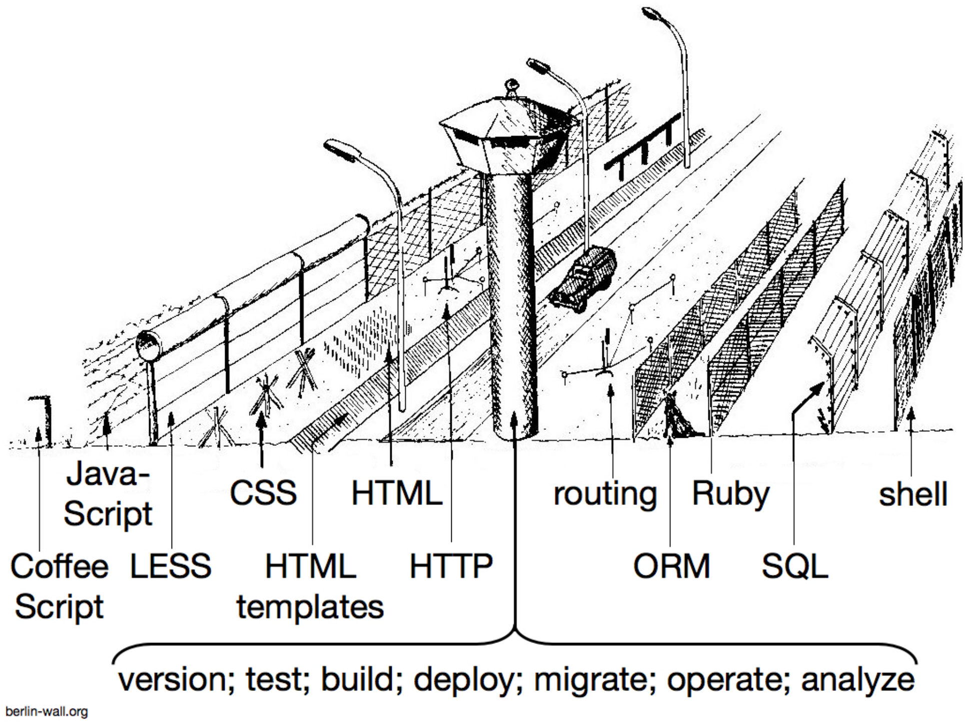


*Figure 3 – Cartoon showing a schematic representation of the obstructive, nested nature of our technology stack — I am indebted to Jonathan Edwards' excellent blog at* `https://alarmingdevelopment.org/`

[8]in the sense of `https://wiki.fluidproject.org/display/fluid/A+Good+Function`

```
   @rsms: Here's what we need to do:
   - take a browser code base
   - add a reliable UI API alongside HTML
   - when a http response says "content-type: app" load it with the
reliable UI API instead of HTML
   This way HTML can be used for what it's good for and app quality goes
up, cost goes down.
```

*Figure 4 – Discussion on replacing HTML in favour of a "reliable UI" linked to from Twitter conversation on Jan 14th 2021 at* `https://twitter.com/rsms/status/1349751077818695680`

Instead, I see that the only route to success is to highlight an increasingly illuminated "blessed path" threading through particular kinds of uses of the technologies that we have already. This will take the form of not just particular libraries, but also explanations of ways of working and idioms. These libraries and idioms, loosely integrated at first, but increasingly cooperating, address parts of the issues of malleability that emerge at every layer of the stack simultaneously. Uses of technology that don't align with the idioms of these libraries will not be forbidden, but will be presented as an "uninterpreted" part of the design together with a notation that they may constitute a risk to malleable development. In the end, front-end tools will emerge that allow the entire application structure to be addressed, which will necessarily have to be capable of representing the same artefact from multiple related points of view. These tools can only be developed by working alongside communities of interest for an extended period, and building pluralistic tooling which supports usage by those communities of their tools as it actually emerges. Rather than an academic "smash and grab" raid which produces a dazzling and powerful theoretical framework before moving on to some other problem, whilst leaving the existing power structures and idioms in place, we need cycles of patient work where we repeatedly evaluate our means of expression against the same designs and requirements.

As a metaphor for how this solution process may manifest, I reproduce a further Twitter exchange in Figure 5 in which it's imagined we will simply "come to see" our new technology stack latent in the old one.

```
@jonathanoda: FFS if we never redesign old technology our stack will
continue to get ever more complex and creaky #FFSTechConf
@stephenrkell: Being a bit Devil's advocate: how can we avoid
"redesign" leading to a new stack living within the old stack? People
often want old and new to coexist. There's some sort of collapsing trick
we need to plan for; I believe usually *during* the new design. Tetris
is my metaphor....
@amb26ponder: Indeed the new stack must live within the old stack. But
the old stack will not collapse, it will merely cease to be salient ...
at least to those wearing the appropriate set of "2d goggles"
```

*Figure 5 – Discussion on how a new technology stack may "emerge", Twitter conversation on Jan 11th 2018 at* `https://twitter.com/jonathoda/status/1016420988488122368`

## 3. Markup-driven polymorphism

It is clear that a good point to start our explorations of reversing power flows in design is at the markup end, since under "naive realism" it is the representation we are most likely to mistake for the thing itself.

### 3.1. What could we do from markup

How far can we push the idea of giving users and designers power over application design, given just tools which let them adjust markup? The templates shown in Figure 1 show some more and less ambitious customisations. As well as rearranging the layout of UI elements, the designer can substitute any static material with any other, as well as selecting from and suppressing any of the controls implemented by the developer (that is, suppressing their implementation, not just their rendering). But how much further can we go — in terms of actually letting the designer adjust application function?

## 3.2. Elementary polymorphism within HTML

A good starting point in exploring how designers can express policy about application function from markup is considering what kind of "polymorphic" function already exists within HTML as written. A straightforward example of this can be found in the controls which HTML provides to accept textual input. From their data binding and functional actions, one could consider controls implemented as `<input type="text"/>` and `<textarea></textarea>` to be peers. RSF did indeed implement them as a single, polymorphic component named `UIInput`, and placed complete control over the markup strategy used for it in the hands of the markup author. If it detected the former markup in the template it was provided with, it applied data binding via the `value` attribute, whereas if it was the latter, the binding would be applied via the DOM element's textual value.

One could imagine taking such polymorphism much further — for example, substituting one of the dizzying array of strategies for implementing and styling HTML buttons for another simply by swapping out the markup.

## 3.3. Limitations of the model

During 2008-2009, it was becoming increasingly clear that there were fundamental problems with RSF's usage model. The most important of these was that its target audience, the "moderately technical designer", essentially did not exist. Whilst there were not zero people of this kind, and the few we did have were enormously productive and valuable, they were far from typical. In fact, the social space broke down almost completely into "technicians proper" who were happy slinging around code and only slightly less happy slinging around markup, and "designers proper" who were primarily visual designers, likely to produce an application sketch in Adobe Illustrator, Omnigraffle (or today, Figma), or even a hand sketch on paper. In practice, it was assumed that by the time the design had been rendered as markup, it had already been fully handed over into the space of developers.

An equally great problem was that RSF's model of authority was *still* authoritarian — all I had managed to do was flip the arrow in one area so that it always pointed backwards. This had further knock-on effects — the integrator of the application had to lay their hands on a complete set of markup for the application, or it couldn't render, as well as any decision that integrator made being impossible to override by a further integrator. This set in motion trains of thought that 7 years later would result in the "Open Authorial Principle" (Basman et al., 2018), but in the meantime I had joined the Fluid community[9], working on our JavaScript framework Infusion[10], seeking these open authorial ends, and then moved to Boulder, Colorado for a 5-year visit with Clayton Lewis.

## 3.4. Arbitrated control

Close to the beginning of that visit, in November 2009, there was an important whiteboard conversation[11] in which a crucial idea emerged — what if the markup's control of the application could be *arbitrated* somehow?

On the whiteboard were sections of component tree and markup like those in section 3.2 — and the crucial turn was a gesture towards it with a phrase like, "If the user has written a `<input type="text"/>` there, then it uses that, otherwise it uses what it has". In practice this was still a bit naive since it was still imagined that there could be a model where every choice could be ultimately sourced to the markup, but in practice the seed of the idea of there being multiple routes for influence over the application's structure, through multiple representations, had been planted.

12 years then passed, during which the implications of this idea started to be worked out in the design of Infusion. Ironically so much time passed that although at the back of my mind this was one of the guiding targets of the work, some of the implications, and indeed the original impetus of the idea came to be forgotten, especially given the realisation in section 3.3 that suggested that markup-based control

[9]`https://fluidproject.org/`

[10]`https://github.com/fluid-project/infusion`

[11]Sadly, given the technical culture of the time I did not have a smartphone capable of taking decent pictures, and had not lugged my camera into the campus, so the contents of this whiteboard have not been preserved.

was comparatively unimportant given the apparent absence of its market. In the end, the system that was built by 2021 included assumptions that made delivering this idea almost impossible within the framework, for reasons that will be worked out in the following sections.

## 4. Implications for system workflow

Part of the reason that so much time passed at this point[12] was that numerous other issues in the design of openly authorable systems needed to be worked out (also bearing in mind that the notion of open authorship itself only came into the open late in this period in (Basman et al., 2018)). These issues were somewhat orthogonal, but also somewhat intersecting, in that solutions to one issue would become either source materials or obstructions to solutions to others. They can be summarised roughly under three headings, corresponding to three periods during the development of Infusion —

1. Producing an coordinatised arena, or "substrate"[13], in which the intentions of multiple authors could be combined (Period I, "Demands blocks", 2009-2011, Period II, "IoCSS", 2011-2015)
2. Ensuring that the substrate properly *separated* the intentions of multiple authors in a transactional way until it was appropriate to combine them ("Potentia II", 2015-2019)
3. Becoming agnostic about whether an author's materials were available immediately or required I/O in order to resolve them, and ensuring that the workflow of fetching them and processing was regular across the substrate ("Workflow/New Renderer", 2019-2021)

In this section I will talk about the the third of these issues, since it exhibits how the issue of markup-driven polymorphism (and other issues) forced the examination of how a system's workflow is organised, and represents a definite split between Infusion's design and the traditional functional pattern which is now dominant in the industry.

### 4.1. The dominant functional pattern

Figure 6 shows a schematic representation of the dominant industrial pattern for constructing the user interface of an application, which is summarised by the slogan "The view is a function of the model"[14]. This representation is a gross simplification of many of the details which appear in many descriptions of this pattern (which appears, for example, as "The Elm Architecture" described at `https://guide.elm-lang.org/architecture/` or "The Reactor Pattern" described at `https://read.reduxbook.com/`), but it covers the two essential points common to all these designs, being that i) going from left to right is a synchronous activity described by a pure (mathematical) function, and ii) going from right to left via the dashed arrow is an essentially uninterpreted, asynchronous activity. Key to the functional paradigm is that the view function is opaque, like all functions — it consumes its arguments, and by means of some arbitrary computation replaces them with its result. Whilst in practice there is a lot of rich internal structure in this function (e.g. it will be composed of a cascade of similar functions), this internal structure is not intended to be of interest to anyone other than developers.

Now, we can comprise section 3.4's demand for arbitrated markup-driven polymorphism as a subcase of Infusion's general mission towards open authoriality — it should be possible for an arbitrarily late interaction with the system to change its structure. But this kind of interaction makes the problem with Figure 6 particularly clear — in this case, the interaction is from the domain that we have already designated to be our *output*. We can't afford a system with a functional arrow that points in one direction. Instead, we need a system which will mirror the structure of its output with a tree of "components", each of which is put into a bidirectional[15] relationship with the generated interface.

[12]As well as, unavoidably, that I "had a number of other things to do" — Infusion was only a funded project during 2008

[13]This term is not being used in a very precise way, but is derived in our tradition from (Beaudouin-Lafon, 2017) — more on this issue at `https://twitter.com/jonathoda/status/1185888711210389504`

[14]The currently fashionable incarnation of this representation has emerged from a long period of taxonomising whose "angels dancing on the head of a pin" character has led it to be described as the "Model-View-Whatever" paradigm (`https://stackoverflow.com/questions/13329485/what-does-mvw-stand-for`)

[15]Jonathan Edwards' Twitter feed produces the goods once more. Whilst there was a phase of experimentation with bidirectional data binding frameworks, the industry long ago standardised on unidirectional data binding, as `https://twitter.com/jonathoda/status/1311018137085595648` of September 29th 2020 reports, on principally anecdotal and aesthetic grounds and largely without a satisfactory exploration of the arguments.

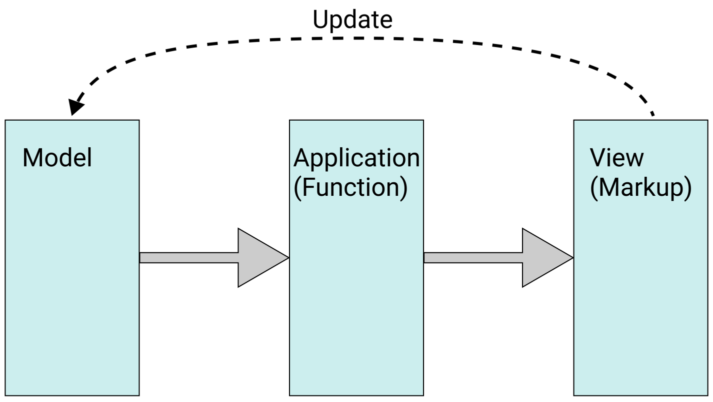


*Figure 6 – Diagram showing the dominant industrial idiom for building a UI — "The view is a (pure) function of the model"*

## 4.2. How does the running system build itself?

But we face a problem describing how this system is meant to instantiate itself, if we look to previous models in computer science. If it were composed of "objects", each of these is meant to fully construct, before it returns its object reference to its parent. This is clearly impossible, since until we begin to construct such an object we can't know whether its markup is there, or whether instead it requires an asynchronous fetch of some piece of markup available elsewhere. And circularly, we can't know whether we need to do such a fetch until the object has got to a sufficient point in its construction as we can resolve all the other intentions in the system influencing what markup it should be bound to — another author may have decided to arbitrate away its markup-driven nature after all.

## 4.3. Avoiding premature commitment

Infusion's design (indeed, the design of any sufficiently authorially open system) instead calls for a simultaneous wave of instantiation across the entire tree. A heuristic that appeared essential from an early phase could be derived by punning on one of the central cognitive dimensions of notation from (Green and Blackwell, 1998), *Premature Commitment*, or perhaps more relevantly since we are designing a system rather than necessarily a notation, a principle from AI planning known as the *Principle of Least Commitment* (POLC, (Weld, 1994)). Under this principle, given a number of potentially equivalent choices available, one picks the one which restricts one's future choices as little as possible.

It's still not entirely clear, other than heuristically, why this principle is so important, but it was pretty clear that "letting one component run ahead of the others" led to system behaviour that was undesirable. Markup again provides a helpful microcosm of this — given how expensive it is for the browser's rendering process to traverse the DOM converting changes into visual effects, it is much more performant to batch one's updates to the DOM into a single operation. In practice, this means that one should not touch the document's markup until the point one is ready to make all updates to it at once. This implies that a component that runs ahead of its siblings in entering the phase of its workflow where it makes markup updates would be a serious performance hazard.

But in "old Infusion" (versions to date) there was a far more crucial reason to avoid premature commitment, which was that, in a nod to industrial virtues, the parts of a component's structure which were not explicitly part of its (mutable) model were evaluated *immutably*. This implied that the effects of any prematurely evaluated component option would be stuck for the lifetime of the component at an unrevised value not reflecting as many authorial intentions as possible.

As we will see, this is a design blunder in Infusion which must be corrected in the upcoming rewrite, but it still seems likely that some form of POLC will always be necessary even if its importance will be less sharp. In general, all frameworks that permit bidirectional influence need to be rich in heuristics in order to make them stable and somewhat predictable, and perhaps one day it will be possible to do adequate research into the ergonomics of this.

## 4.4. Infusion's workflow

Figure 7 shows Infusion 4's workflow for an individual component. As we outlined in the previous two sections, one transaction's worth of instantiating components go through the phases to the left of the dashed line as a unit, following the principle of least commitment. The complication is that any workflow stage for one component might trigger the discovery of a further component, which enters the transaction at the start of its own workflow. Once they have all reached the stage of the dashed line, "orthochronous time" re-establishes itself and each component then separately, in reverse order, goes through an initialisation phase similar to that of a traditional object-oriented component — primarily to retain compatibility with code written against previous versions of Infusion.

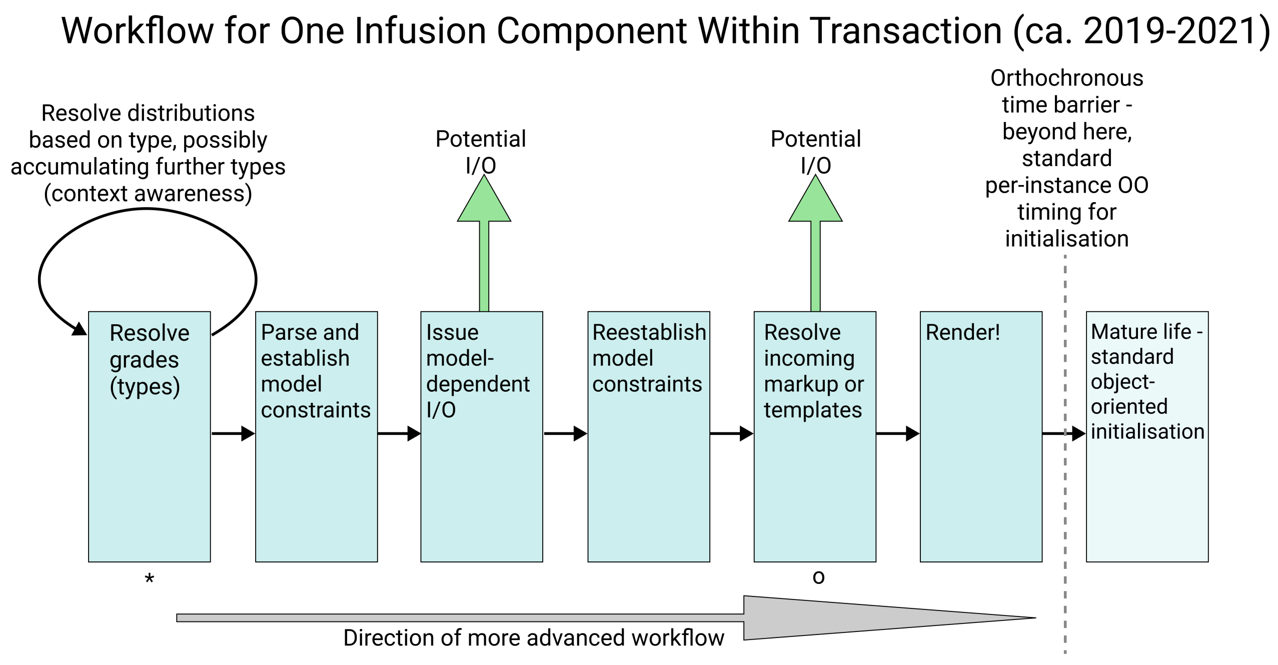

Workflow for One Infusion Component Within Transaction (ca. 2019-2021)
Resolve distributions based on type, possibly accumulating further types (context awareness)
Potential I/O
Potential I/O
Orthochronous time barrier - beyond here, standard per-instance OO timing for initialisation
Resolve grades (types)
Parse and establish model constraints
Issue model-dependent I/O
Reestablish model constraints
Resolve incoming markup or templates
Render!
Mature life - standard object-oriented initialisation
*
o
Direction of more advanced workflow

*Figure 7 – The workflow stages for instantiating a single Infusion component which is participating in a markup transaction, as of Infusion 4*

This workflow system gave rise to a "napkin diagram" as reproduced in Figure 8[16] in which workflow as seen in Figure 7 is now plotted vertically, and on the horizontal axis are plotted components in order of discovery. In this scenario it is imagined that the system at first discovers two components, brings them to workflow level 3 by stabilising their models, then issues some I/O which then triggers the discovery of 5 more components which must be brought to the same level before markup rendering begins. Note that the arrows above the dark line run backwards, as per standard object-oriented semantics whereby the most deeply nested component finishes its construction first.

## 4.5. A Hiatus

Throughout the 2010s, pursuit of markup–driven polymorphism was a back-burner issue, since we had seemed to determine that

- The target audience for it (the "fairly technical designer") was very limited

[16]Sketched in October 2019 – this kind of diagram in which workflow progress is plotted vertically against component allocation horizontally has been named a "prokoptogram" following the Greek προκόπτειν representing "progress", although it could more accurately be named a "exelixigram" following εξέλιξη for "evolution".

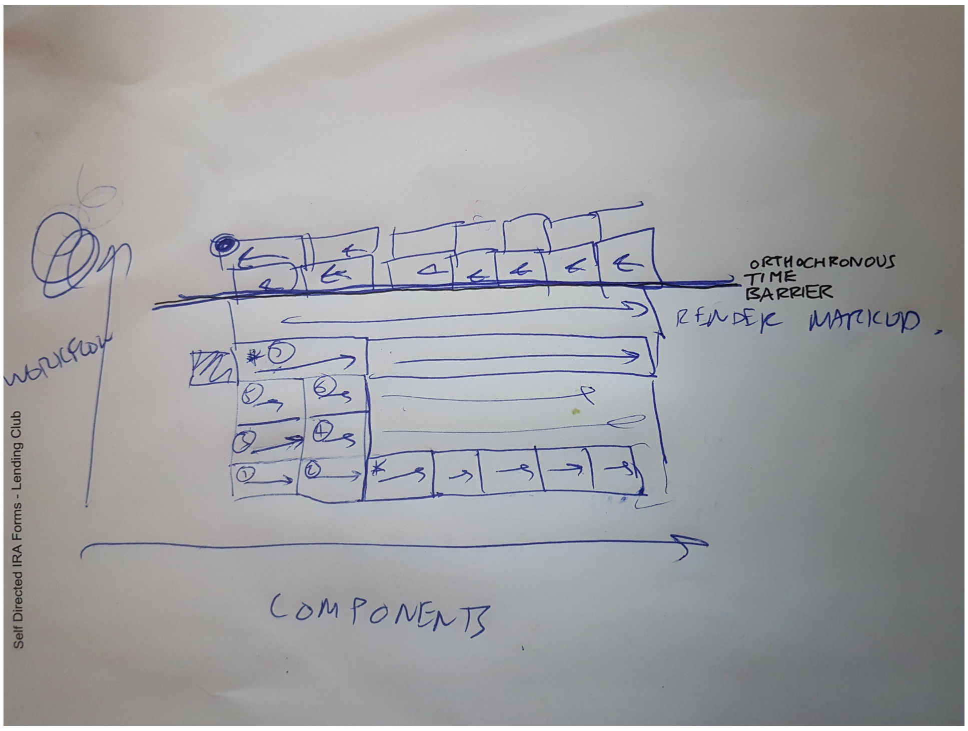

ORTHOCHRONOUS TIME BARRIER
RENDER MARKUP.
WORKFLOW
COMPONENTS
Self Directed IRA Forms - Lending Club

*Figure 8 – A "prokoptogram" showing the collective progress through workflow of a group of components each individually passing through workflow stages of the kind diagrammed in Figure 7. In this diagram, Figure 7's horizontal workflow direction is plotted vertically, and our horizontal direction represents discovery of components*

- The vast majority of markup-oriented customisations, as those shown in section 1.1, could be achieved by modern CSS features, particularly those from the "flexbox" layout system widely available from 2014
- The use of markup-oriented tools was generally waning

and the workflow system shown in section 4.4 was developed without directly considering it. However, a few months ago an interesting incident occurred when Philip Tchernavskij was attempting to bind an Infusion component against markup produced from the Webstrates system (first mentioned in section 2.1). Despite my focus in the 2010s on the Infusion renderer unilaterally rewriting markup it found in the document (in line with the dominant industrial paradigm of React et al.), Philip found that he was still able to bind existing markup managed by Webstrates to new Infusion components. This was as a result of the backwards compatibility[17] of the rest of the Infusion system with the API which still supported the "progressive enhancement" idiom of many of our 2008-2009 era components.

At this point it dawned on me that markup-driven polymorphism, whilst having few urgent practical uses, was essential to Infusion's open authorial mission, and as well as directly supporting cooperation with untypical citizens such as Webstrates, would also indirectly support cooperation with the increasingly fashionable static publishing systems[18] which once more put "progressive enhancement" uses back into scope.

## 4.6. Workflow Problem

I set about implementing this in the version of Infusion 4.x which had just completed a rewrite away from the "virtual DOM" (another React-idiom construct which I had realised was unnecessary) and quickly found that it was impossible under the workflow ordering which had been established in Figure 7. The problem is as follows — we only encounter the markup which we will render against at the workflow

[17]Despite the great upheaval of the rewrites described at the head of section 4, code up to Infusion 4.x remained almost perfectly compatible with code back to the Infusion 0.x system of late 2008 — this will not be possible to maintain in the next rewrite.

[18]Hugo, Jekyll, Eleventy, etc. broadly comprised under the movement named JAMStack `https://jamstack.org/`

stage marked with an "o" (Resolve incoming markup or templates), but our goals of open authoriality require that we honour this by an adaptation which is indistinguishable from one that could have been enacted by any other author, and so we must return to the very first workflow stage marked with a "*" (Resolve types), fetch the adaptation, undergo type evolution, apply the new model constraints, etc. working through the remaining parts of the workflow.

Any attempt to finesse this can't get around the fact that the component may have issued two or more pieces of I/O (e.g. one to fetch its model holding a language localisation preference, and another to fetch particular localised markup template based on the language choice) — putting it definitely beyond the moment in time when the component's "type" should ordinarily be completely settled[19]. At this point for a while I gave up the task as impossible, before deciding that it was so essential to the framework's mission that it had to be attempted somehow.

In the framework's defence, I eventually managed to cobble together a sequence of manual adjustments to its bookkeeping structures that achieved most of the desired effect without having to change the internal framework code, but this incident represented the final piece of a sequence of evidence which had been growing over several years that Infusion couldn't meet its open authorship goals without a complete rewrite (see section 6). Ironically, the unidirectional pipeline shown in Figure 7 is extremely reminiscent of a similar sequential pipeline within Sun Microsystems' JSF, RSF's predecessor from 2004, which I had derided at the time. At least Infusion's pipeline is freely interceptible from outside the framework, but not routinely so. In practice it was put in as a stopgap to allow some asynchronous activities during instantiation without having to rewrite the entire framework to asynchronise every primitive in it.

## 4.7. What progress?

So what progress did we in fact make between 2005 and 2021? As the section 7 discussion establishes, markup-driven polymorphism is just one strand amongst many in the general effort towards open authorship, and it interacts in a not entirely orthogonal way with the other strands.

Figure 9 shows side-by-side the 2005 RSF Java implementation and the 2021 Infusion JavaScript implementation of markup-driven polymorphism. Whereas the RSF implementation is expressed in code, deeply buried in a method body hundreds of lines long holding numerous conditions, the Infusion implementation consists entirely of JSON configuration in a few short snippets with global names. The process of determining how to bind the markup is organised by a sequence of pattern-matching rules in the block `markupChecks` which test the incoming markup at a particular selector path for matching a selector. This kind of pattern-matching is familiar from the functional programming world, and it is easy to imagine how it could be democratised by being exposed in some form of structure-editing tool.

| **2005 RSF Implementation** | **2021 Infusion Implementation** |
|---|---|
| Unidirectional markup polymorphism - control over rendering and binding strategy always lies with markup | Arbitrated polymorphism - markup or component tree may drive strategy |
| Implementation is closed within a large unit of programming language source code | Implementation is open in a globally named block of configuration |
| Repertoire of basic components and binding strategies may not be extended | Repertoire of components may be freely extended and reconfigured |
| Polymorphism may only be configured on the server | Polymorphism may be configured on server or client, and client may take further control following decisions made on the server |

*Table 1 – Tabulation of relative merits of 2005 and 2021 markup polymorphism systems*

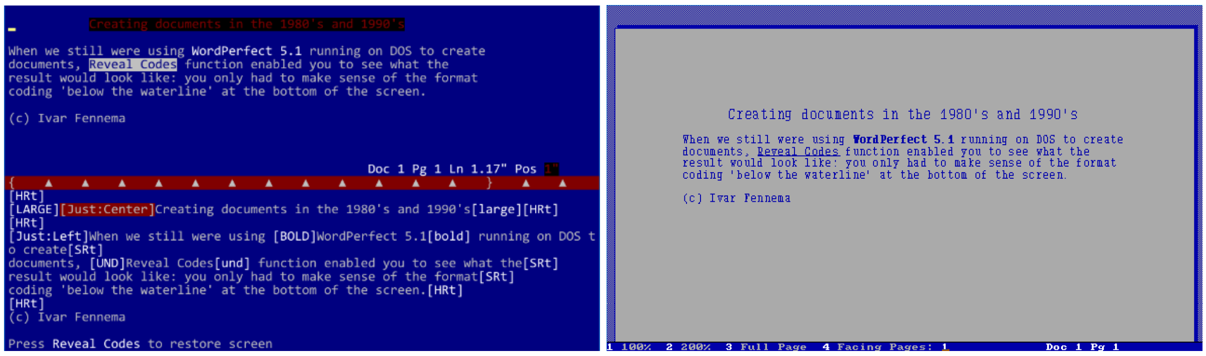
[19]Note that problems of this kind can't arise in the Figure 6 industrial pattern because of its insistence that all I/O and the "model" or reactor state must be completely resolved at the moment rendering begins. Note that in a functional paradigm, issuing I/O represents a side-effect which must be prohibited, or at least "pushed to the periphery of the system". This is incompatible with an open authorship model whereby we can't expect to know what adaptations the system needs to perform until it has started to instantiate.

```java
public class BasicHTMLComponentRenderer implements ComponentRenderer {
...
// 15 lines of setup
...
  public void renderComponent(UIComponent torendero, View view,
    TagRenderContext trc) {
    ...
    // 60 lines of conditionals
    ...
      else {
        String value = ((UIBoundString) torender).getValue();
        if ("textarea".equals(tagname)) {
          if (UITypes.isPlaceholder(value) && torender.willinput) {
            // FORCE a blank value for input components if nothing from
            // model, if input was intended.
            value = "";
          }
        trc.rewriteLeaf(value);
        }
      else if ("input".equals(tagname)) {
        if (torender.willinput || !UITypes.isPlaceholder(value)) {
          attrcopy.put("value", value);
        }
        trc.rewriteLeaf(null);
      }
    ...
    // 150 lines of further conditionals
    ...
    }
  }
```

```javascript
fluid.defaults("fluid.uiTextBinding", {
  modelRelay: {
    value: {
      target: "dom.container.text",
      source: "{that}.model.value"
    }
  }
});

fluid.defaults("fluid.uiValueBinding", {
  modelRelay: {
    value: {
      target: "dom.container.value",
      source: "{that}.model.value"
    }
  }
});

fluid.defaults("fluid.uiInput", {
  gradeNames: "fluid.polyMarkupComponent",
  resources: {
    template: {
      resourceText: "<input type=\"text\"/>"
      }
    },
    markupChecks: {
      "fluid.uiValueBinding": {
        selector: "",
        tagName: "input"
      },
      "fluid.uiTextBinding": {}
  },
  parentMarkup: true
});
```

*Figure 9 – 2005's and 2021's implementations of markup-driven polymorphism.*
*Left: Excerpt from RSF's* `BasicHTMLComponentRenderer.java`*,*
*Right: Except from Infusion's* `fluidRendererComponents.js`

The merits of the different approaches are tabulated in Table 1. It's clear that almost all these merits and demerits can be squarely positioned in the framing of the open authorship values established by (Basman et al., 2018) — they relate to what kinds of authors can take control of how an expression is interpreted, and in what circumstances. This correlates with the fact that the vast bulk of the work done over this period was in the supporting infrastructure to enable open authorship in general, and virtually none of it in the actual implementation of the markup polymorphism mechanism specifically. Each of the implementations in Figure 1 was in itself the work of at most an hour or so, in contrast to the 16 years which elapsed between it being possible to express and implement one rather than the other.



```
         Creating documents in the 1980's and 1990's

When we still were using WordPerfect 5.1 running on DOS to create
documents, Reveal Codes function enabled you to see what the
result would look like: you only had to make sense of the format
coding 'below the waterline' at the bottom of the screen.

(c) Ivar Fennema


                                    Doc 1 Pg 1 Ln 1.17" Pos 1"
{  ▲  ▲  ▲  ▲  ▲  ▲  ▲  ▲  ▲  ▲  ▲  ▲  }  ▲  ▲
[HRt]
[LARGE][Just:Center]Creating documents in the 1980's and 1990's[large][HRt]
[HRt]
[Just:Left]When we still were using [BOLD]WordPerfect 5.1[bold] running on DOS t
o create[SRt]
documents, [UND]Reveal Codes[und] function enabled you to see what the[SRt]
result would look like: you only had to make sense of the format[SRt]
coding 'below the waterline' at the bottom of the screen.[HRt]
[HRt]
(c) Ivar Fennema

Press Reveal Codes to restore screen
```

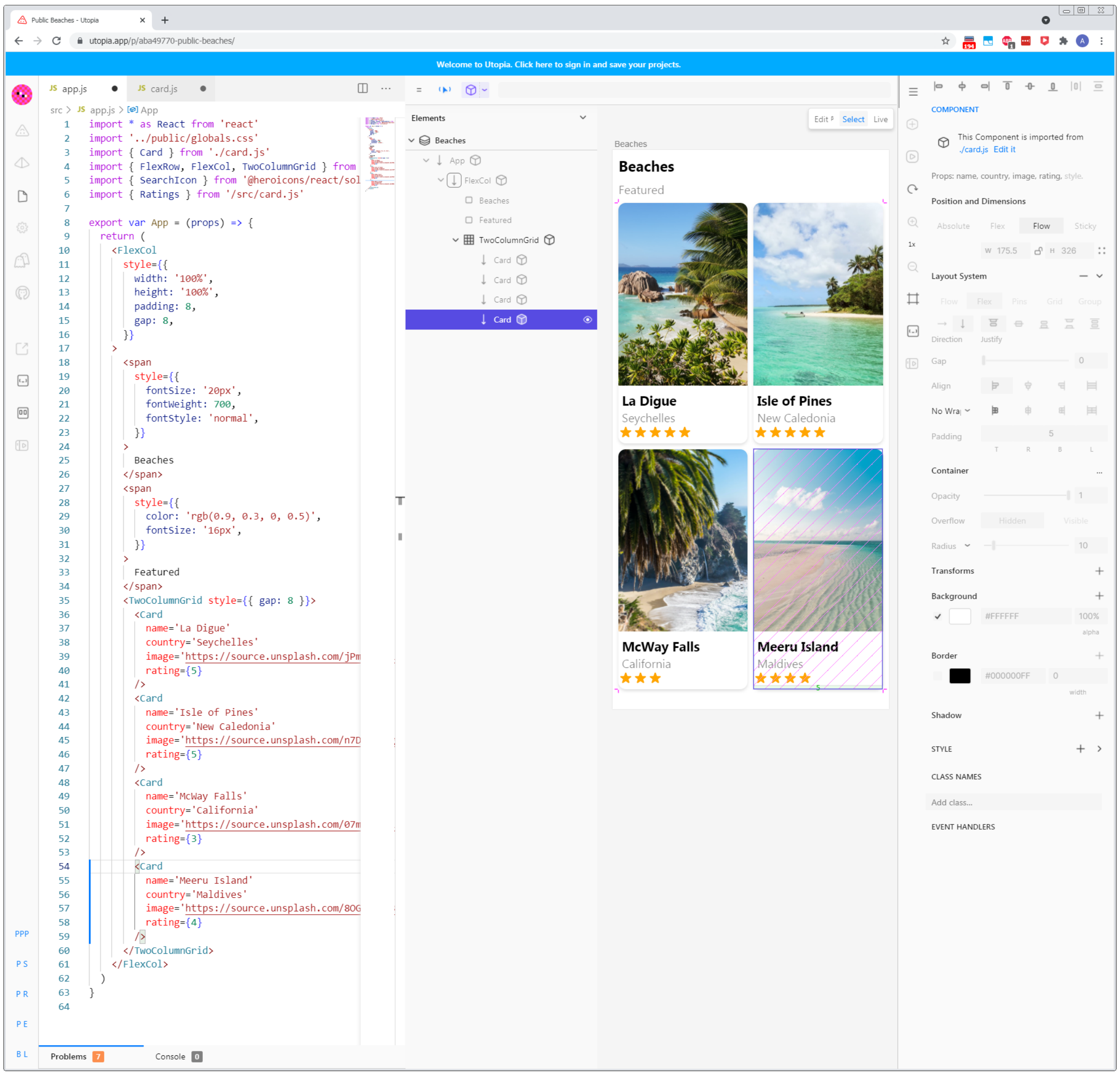
```
            Creating documents in the 1980's and 1990's

When we still were using WordPerfect 5.1 running on DOS to create
documents, Reveal Codes function enabled you to see what the
result would look like: you only had to make sense of the format
coding 'below the waterline' at the bottom of the screen.

(c) Ivar Fennema

1 100%  2 200%  3 Full Page  4 Facing Pages: 1          Doc 1 Pg 1
```

*Figure 10 – Word Perfect for DOS interface with split screen showing two different representations of the same document in a split screen on the left, and a further representation on the right which is close to its final printed form*

## 5. Authorship from Multiple Representations

This discussion has brought us full circle to my first PPIG paper of (Basman et al., 2015) which set out the desirability of authorship from multiple representations . In retrospect, just as with the special topic of this paper, multiple representation authorship can be situated within the frame of open authorship in general. It is desirable to apply multiple representations to the same artefact to widen the space of authors who can engage with the artefact, or complementarily, to widen the space of faculties which a single author can bring to bear on the artefact. Multiple representation authorship in itself is nothing new, as Figure 10 shows. In this figure, we see the same word processor document from three different views, as rendered in the remarkable 1989 product, WordPerfect 5.1 for DOS — this shows at top left "flat" document representation suitable for rapid editing, at bottom left, a symbolic representation showing the system's representation suitable for fine-tuned adjustments of otherwise invisible formatting directives, and on the right a "print preview" representation suitable for previewing the final appearance of the document prior to printing.

This topic is highly relevant to our current discussion based on our realisation from the outset that markup-driven polymorphism could only be effective as part of a spectrum of strategies addressed at different representations. The early RSF days tried to push this angle as far as possible based on the availability of apparently relevant off-the-shelf tools addressing this representation, and the infeasibility of developing multi-representation authoring tools in a reasonable period of time.



```
import * as React from 'react'
import '../public/globals.css'
import { Card } from './card.js'
import { FlexRow, FlexCol, TwoColumnGrid } from
import { SearchIcon } from '@heroicons/react/sol
import { Ratings } from '/src/card.js'

export var App = (props) => {
  return (
    <FlexCol
      style={{
        width: '100%',
        height: '100%',
        padding: 8,
        gap: 8,
      }}
    >
      <span
        style={{
          fontSize: '20px',
          fontWeight: 700,
          fontStyle: 'normal',
        }}
      >
        Beaches
      </span>
      <span
        style={{
          color: 'rgb(0.9, 0.3, 0, 0.5)',
          fontSize: '16px',
        }}
      >
        Featured
      </span>
      <TwoColumnGrid style={{ gap: 8 }}>
        <Card
          name='La Digue'
          country='Seychelles'
          image='https://source.unsplash.com/jPm
          rating={5}
        />
        <Card
          name='Isle of Pines'
          country='New Caledonia'
          image='https://source.unsplash.com/n7D
          rating={5}
        />
        <Card
          name='McWay Falls'
          country='California'
          image='https://source.unsplash.com/07m
          rating={3}
        />
        <Card
          name='Meeru Island'
          country='Maldives'
          image='https://source.unsplash.com/8OG
          rating={4}
        />
      </TwoColumnGrid>
    </FlexCol>
  )
}
```

*Figure 11 – Interface from the "Concrete Utopia" project showing a React application being developed live from three representations*

Such implementations as those shown in Figure 10 are the exception rather than the rule. WordPerfect was eliminated from the market by Microsoft Word which has no comparable feature. However, the idea remains alive and well — a very recently previewing product, Utopia (`https://utopia.app/`), as shown in Figure 11 shows a highly impressive interface showing an industry-standard React app being developed from three points of view — the literal React JavaScript code in the left pane, an structural view in the middle pane, and the live running app in the right. This brings a kind of malleability to this very stiff domain — but naturally only within the confines of this heavyweight editing environment. The actually running app once it is deployed is as remote from influence and authorial intervention as ever. Some of the rhetoric surrounding the development shows an encouraging similarity with that of section 2.1 — "And most importantly, we made it safe: whatever Utopia doesn't (yet) understand, it leaves as-is.". An interface like this combined with an actually malleable substrate could be marvellous.

## 6. The Future

As the previous discussion has established, Infusion has accumulated enough design flaws impeding open authorship that soon it will need to be rewritten entirely. This rewrite will focus, as well as on the workflow constraints that we mention in Section 4, on an extremely important issue not touched on in this paper, that of preserving the lineage and provenance of the expressions of different authors as they are entered into the system. This needs to be combined with massive improvements in performance through "memoisation" of the effects of previously visited configurations. Eliminating the spurious immutability notion that plagued Infusion development to date will produce a system which smoothly enacts any required adaptations in-place with minimal upheaval in memory. The upcoming version of Infusion will more aptly appear as a "Fluid" framework.

## 7. Acknowledgements

This paper is dedicated to Clayton Lewis, in thanks for his boundless generosity and attention in years of Sunday morning hikes and chats during which early versions of many of these ideas emerged.

## References

A. Basman. If What We Made Were Real. In *Proceedings of the Psychology of Programming Interest Group*, 2017.

A. Basman and P. Tchernavskij. What Lies in the Path of the Revolution. In *Proceedings of the Psychology of Programming Interest Group*, 2018.

A. Basman, C. Clark, and C. Lewis. Harmonious authorship from different representations. In *Proceedings of the Psychology of Programming Interest Group*, 2015.

A. Basman, C. Lewis, and C. Clark. The Open Authorial Principle. In *Proceedings of the ACM Splash'18 Onward*, 2018.

M. Beaudouin-Lafon. Towards unified principles of interaction. In *Proceedings of the 12th Biannual Conference on Italian SIGCHI Chapter*, CHItaly '17, pages 1:1–1:2. ACM, 2017.

T. Green and A. Blackwell. Cognitive dimensions of information artefacts: a tutorial. BCS HCI Conference, 1998. URL `https://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf`.

C. N. Klokmose, J. R. Eagan, S. Baader, W. Mackay, and M. Beaudouin-Lafon. Webstrates: Shareable dynamic media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, pages 280–290. Association for Computing Machinery, 2015.

O. Lialina. Turing complete user. *Contemporary Home Computing*, 14, 2012.

M. Mayer, V. Kuncak, and R. Chugh. Bidirectional evaluation with direct manipulation. (OOPSLA), 2018.

J. Swift. *Gulliver's Travels.* Motte, 1726.

D. S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27, Dec. 1994.