If What We Made Were Real

Against Imperialism and Cartesianism in Computer Science, and for a discipline that creates real artifacts for real communities, following the faculties of real cognition

Antranig Basman
amb26@ponder.org.uk

Abstract

I argue that we have not yet succeeded in constructing any real software, but merely simulations or cardboard prototypes of software. I criticise the pernicious imperialism of the so-called "computational thinking" which suggests that society should come to think like computer scientists, and instead suggest that computer scientists should come to think more like ordinary citizens. A greater appreciation of the cognitive underpinnings of everyday thought and everyday life will lead us to make durable, flexible things which are widely useful, as opposed to brittle, fragile things that disrupt everyday activities. We'll make a case study of a paradigm element of traditional programming languages, function application, and consider how alternative building blocks and metaphors can lead to more successful artefacts for humans.

1. Introduction

Imagine if, when we made a piece of software for a particular community, we could be confident that there was a closely related piece of software that met the need of a closely related community. Imagine if the things we created had the status of a vigorous and imperishable characterisation of a need, rather than entering an unsustainable cycle of increasingly frantic maintenance and decay, doomed to be swept from the world in at most a couple of months or years? Imagine, correspondingly, if we could react to an "unexpected user requirement" or a change in technology or context joyfully, as a fresh opportunity to meet a freshly expressed need, as opposed to reacting with fear and despair as we wonder how much of the painful and intricate work we have done so far now needs to be undone?

1.1. Against Imperialism

Rhetoric similar to this has apparently motivated much of the products of Computer Science of the last 60 years, but as each new decade succeeds to the last, there is an increasing lack of recognition of how profoundly we are falling short of what ought to be possible. In a purely mental discipline, we suffer from none of the constraints of material and energy. Instead of delivering on these infinite possibilities, we fall back on not only complacency and cynicism, but active imperialism as we convince ourselves and our users that we have delivered not failure, but success — that rather than needing to try harder, we imagine that the rest of the world should adopt our own methods, most lately under the bandwagon of "computational thinking" (Wing, 2008), because of what we argue are their manifest success and suitability. Rather than redoubling our efforts to understand the nature of real thought and real communities, we spend our time trying to convince the world if it were "more like us", it would be better off — and the way we choose to portray ourselves is as mechanistic, materialistic, unsubtle, inflexible and judgemental. The highest virtues of the new "computational thinking" are those most boring virtues of efficiency and correctness. Is it a wonder that most normal people are alienated by the products of the computational world which they see as an increasing stranglehold rather than an ally — a rising tide of barely functional pieces of "techno-junk" that rarely work properly, constantly promote frustrating interactions and are destined for landfill (both physical and virtual) in short order.

1.2. Towards Practice

As well as moral positioning, this paper comprises description of the aims of real communities[1] that are doing real work steadily to bring about the change in thinking and practice that we need. The concrete results of this thought are being collected in Fluid's Infusion framework (Fluid, 2017). In the last section are links for further reading and how to get involved. Characterising this work is not easy, since 60 years

[1] Fluid (`www.fluidproject.org`), the GPII (`www.gpii.net`) and others.

of increasingly entrenched thinking and practice make it increasingly hard for anyone to even see that there is a problem, or even to recognise what the work might look like that aims at a solution[2]. As mental horizons shrink, any work aimed at more than an immediate payoff is written off as a "boil the ocean" mission, and as each new generation of students appears, they face an ever-more complacent generation of mentors who believe that they hold the tools of solution rather than embodying the problem.

2. What Should Be Possible

Here are some more characterisations of what should be possible:

2.1. Software is worked on by means of itself

Scratch the surface of a physical product such as a chair or a wall, and you find something broadly similar underneath. The physical world is worked on by means of tools that are part of its own idiom — whether we cut a piece of wood into a smaller piece of wood, or make a hole to hold a bracket, we are using the affordances of the world itself to cause change. Contrast this with the nature of a modern piece of software or hardware — scratch the surface and underneath it is an incomprehensible world of blinking lights and mass of wiring that bears no resemblance to the physical form and affordances of the overall object. Now, hardware we can't do much about — we are constrained by the requirements of real engineering. Software, being purely the product of the mind, should be able to be anything we like. In presenting to someone a "computational artefact", we should simultaneously put everything they need into their hands in order to make choices about it, to work on it, to share it with others, and to find communities who have made similar (or even contrasting) choices. Instead we present them with a "locked box" with a limited number of dials to twiddle. The mystical philosophy of Sufism states that "Sufism is studied by means of itself" — what we want to bring about is a world of software that "is worked on by means of itself". I wrote last year (Basman, 2016) on what could constitute durable, forgiving materials for software construction. The Sufi slogan is also closely related to the mantra of live programming "The thing on the screen is supposed to be the actual thing" (Ungar & Smith, 2013) which we treated in (Basman, Church, Klokmose, & Clark, 2016).

2.2. How we currently have no software

I argue that today we have no software — what we have is merely the simulation of software. What we have today bears the same relationship to real software as the set of a Hollywood movie does to the real places and scenes that it portrays. The movie set creates the impression of a particular scene which is good just for an observer in a carefully controlled place and for a limited set of purposes (the camera and its optics). Similarly, our software meets a set of needs which are good for a tiny set of users under a limited range of contexts — often this set is so idealised that the software doesn't actually adequately meet the needs of any real users. A small change in perspective of the camera or a small change in usage (pushing against a prop wall that wasn't designed to be rigid, for example) instantly exposes the sham of the movie set world. Similarly, a small change in requirements exposes the sham of the software we have — it may end up being treated as an entirely different piece of software with a different set of requirements — just as a complete movie is often shot with several different complete reconstructions of different scenes from different points of view or scales. We should be able to have software which is real, in that it behaves with the same continuity and consistency as real materials — real trees and real mountains expose a consistent and coherent set of linked aspects, affordances and appearances as we

[2]Idries Shah, in his *Knowing How to Know* (Shah, 1998) has this to warn us in searching for a "Golden Age": "How interesting that people think about a 'Golden Age' and hope for the coming or the return, of one. I have noticed that they never give any consideration to these concepts:
1. How would they know a Golden Age if they entered into one?
2. Could they survive in a Golden Age?
3. Have they been in a Golden Age, without recognising it?"

move from place to place, scale to scale and sense to sense[3].

3. How we got into this mess

We got into this mess through 60 years of consistently drawing the wrong people into our field, and continuing to entrench its vices rather than reform them. In the "Garden of Eden" phase of Computer Science when such inspired products as McCarthy's Lisp and Sutherland's Sketchpad were plentiful as tabby cats[4], it was easy to imagine that maturing to solve more ambitious problems for a wider class of people was just a step away[5]. In a world that has given us Java, Haskell and Ruby, success seems further away than it ever has been. Computer Science attracts people who are addicted to control — that is, their ability to have unilateral jurisdiction over some ever-increasing universe of effects and expressions. In George Orwell's terms, such people are "power-worshippers" (Orwell, 1946) — enthusiasts of the strong simply because they are strong, and oppressors of the weak simply because they are weak. This tendency can be seen every day in the common rhetoric of the field — successful programmers are hailed as "wizards" or "ninjas" (and encourage others to do this) — glorying in power simply for the sake of power[6]. The push towards "computational thinking" is simply the same dysfunction dressed up in more respectable clothes — just as "intelligent design" is an attempt to create an acceptable, "highbrow" and intellectual packaging of the same worldview underlying creationism. In this worldview, the technologist is the one who "has power" and has mastered certain "mysteries" through the application of "correct techniques". Others should aspire to be more like him, rather than the technologist humbling himself to put his gifts and worldview at the service of the public[7].

Our current incarnation of this disease can be traced back at least to Newton. As argued by (Lakatos, 1978), Newton consistently falsified the nature of the methods he had used to achieve his startling results, in order to solidify his grip over the nascent community of natural philosophers. Newton argued that he had "deduced his theories from the facts", which is a completely false account of the creative and inductive methods that he really used. Newton's followers were completely convinced by his rationalistic account that he had achieved his results through deductive reason starting from the evidence, and went on to convince others. In this way Newton could be described as "the first wizard"[8], in the tradition that software engineers today conceive themselves. In convincing themselves and the rest of their community to apply these methods, Newton's followers ushered in two centuries of scientific darkness in England, in which no productive results were achieved again until Maxwell and Babbage arrived in the mid-19th century to sweep the stables clean. There are strong grounds for believing that our field is in the middle

[3]This imagery is treated in a more concrete way in (Basman, Lewis, & Clark, 2011) which describes the goal of a "homogeneous tower of abstractions" that is encountered when dealing with a single artefact from a variety of different viewpoints and scales. Our aim is to make these as closely related as possible, with as graceful, gradual and intelligible transitions between the different views, rather than the heterogeneous and unintelligible jumble that today's "fake software" presents. It is also related to the crucial material property of "continuity" that we discuss in (Basman, 2016).

[4]Lord Chancellors were cheap as sprats,
And Bishops in their shovel hats
Were plentiful as tabby cats (Gilbert, 1889)

[5]This great "retrospective hope" is the basis of an amusing and insightful presentation by Bret Victor (Victor, 2013) in which he purports to be addressing a 1973 audience comprised of "programmers of automatic computing machines". Based on the inspiring achievements of 1973, he imagines numerous marvellous developments for the software of 40 years in the future, none of which have actually transpired.

[6]A good rhetorical example appears in (Hoyte, 2008): "but macro programming is, of course, not about style. It is about power."

[7](Winner, 1995), describing the UTOPIA project of the 1980s in its rare attempt to push back against the "ritual of expertise", explains that

> ... those who came to the process with university degrees and professional qualifications explicitly rejected the idea that they were the designated, authoritative problem-solvers. Instead they offered themselves as persons whose knowledge of computers and systems design could contribute to discussions conducted in democratic ways.

[8]By contrast, Newton was in fact recently portrayed in (White, 1997) as *The Last Sorcerer* — a rewarding book which is rich in facts though thin in philosophy.

of a similar period of darkness for quite similar reasons — let's hope we can bring it to an end in fewer than 200 years.

3.1. What's wrong with efficiency and correctness?

Whose efficiency? What correctness? The elevation of these virtues reflects a dominant culture of accountants rather than creators. It imagines a single universal viewpoint from which these virtues can be consistently judged. (Feenberg, 1995) explains that technocratic criteria of efficiency, applied for example when considering environmental effects of technology, often result in less efficient end to end processes. But in fact, if we can't even meet the needs of one user, what value could we ascribe to the consistency or correctness of the approaches we use to fail to meet them? It's crucial to concentrate on positive virtues first, before turning to negative ones. Positive virtues include expressivity, the promotion of creativity, diversity of viewpoints and the understanding of relationships between them. These are the virtues that are appropriate for a young field that is not yet confident in its capabilities to do real work. As a field matures and becomes clearer about its engineering terrain, it then becomes appropriate to spend time consolidating our hold by turning to such negative virtues — negative because they involve the censoring or the restraint of expression rather than promoting it. Our field suffers right now from a kind of "premature senescence" where we somehow imagine a capability that we have not, and that it is already the time for expressing the virtues of senescence. In fact, we have merely "become old without becoming wise"[9].

4. What can we do about it?

Now we must seek out practical directions for achieving the aims of having real software. As we alluded to above, a significant part of this work will involve finding ways to give up power, rather than hungrily seeking it. This relinquished power will then be freed up to be delegated to our users.

4.1. Giving up power rather than accumulating it

Here are a number of kinds of power, widely considered traditional amongst software engineers, that we should give up:

i) The power to create grammars with infinite numbers of valid sentences
ii) The power to construct programs that might consume unbounded time and/or space, or perhaps never terminate
iii) The power to hide pieces of state behind abstractions (APIs or other kinds of interfaces)[10]
iv) The power to construct pieces of software through irreversible or nearly irreversible machines such as compilers
v) The power to divide up a particular domain into a single hierarchical decomposition of entities with properties, connected by relations[11]
vi) The power to prescribe the exact sequence of operations needed to achieve a particular result[12]
vii) The power to import machinery, definitions or methodologies from related disciplines, without evaluating their tendency to result in appropriate products
viii) The power to change the form or behaviour of a program in an updated version, without giving a cost-free (to both users and developers) choice to retain the old form[13]

From time to time there have been movements aimed at delegating at least a couple of these powers — for example the "sequence of operations" power vi) has a number of incarnations of technology aimed at delegating it, for example the logic programming language Prolog, or modern control flow packaging technology involving monads. But by and large the majority of these powers are not only considered sacrosanct, but also keeping hold of them has been made the basis of virtue in several major traditions

[9] King Lear, Act 1, Scene 5: "Thou shouldst not have been old before thou hadst been wise"

[10] In (Clark & Basman, 2017) we discuss how this power can be relinquished in favour of working with an externalised state idiom.

[11] In (Basman, Clark, & Lewis, 2015) we refer to possessors of this power as entitled to express "excess artefact boundary intention".

[12] In (Basman et al., 2015) we refer to possessors of power vi) as entitled to express "excess sequential intention".

[13] In (Basman, 2016), this is listed as the first of our "house–clearing" tasks to be tackled as a prerequisite for establishing durable materials for software.

of engineering. For example, the power iii) of hiding state is the bedrock of Object Orientation (as is the power v) to create "entities"), and Functional Programming goes yet further in its insistence that state should not only be hidden, it should be claimed to not exist at all. Similarly it is considered axiomatic that a grammar without an infinite number of valid sentences can't be interesting or worthwhile, and many accounts of human language try to shoehorn it into this view by claiming that these are realistic models of the kinds of languages we actually speak! Naturally this creates a number of purely factitious problems in trying to explain how learning works as a result of its blatant inaccuracy.

5. A case study: Function composition as the first evil

As an example of "importation" (the 7th power mentioned in the list of delegations), we can consider function composition, a seemingly harmless idiom imported from mathematics. One author might write the expression $h(x) = f(g(x))$ as a seemingly reasonable way to define a new function in terms of two pre-existing ones. This technique is actually at the foundation of the entire subdiscipline of functional programming. This kind of definition is invariably portrayed as virtuous, without a consideration of the costs incurred relative to the benefits achieved. And the costs are considerable: to the user of h, the composition forever afterwards behaves as a "black box" — the inner details of f and g's existence will never be revealed again. And the mere fact that it is such a black box is seen as the virtue rather than the vice — since h is now interchangeable for any other function achieving the same effects as the composition of f and g, regardless of how they were created. The fatal difficulties that this "blind composition" poses for further creators in the same space as the original author are rarely considered. Should a second or third creator want to interpose themselves in this chain, and express some other choices relative to this application process, they have their work cut out for them. For example — imagine that what author 2 really wants is to adapt the creation of author 1 so that it reads, $h'(x) = f(v(g(x)))$.

In many environments, this is impossible since the application point is simply notationally lost forever. In the Lisp programming language, uncovering the application point is technically straightforward since every function application is simply represented as a list data structure. However, although it is technically straightforward, it is not morally straightforward — since there is still no stable point representing the name or location of the application point of f and g. That is, it has been, and can be provided with, no name that further creators could use to identify it. If the 2nd creator "happens to know" that they are faced with an expression that contains exactly 2 function applications, they can easily perform the list manipulation required (by means of a Lisp macro) to convert creator 1's expression into the one they want. But this process is "informationally unstable" — 3rd and subsequent creators will struggle more and more with an increasingly disorderly terrain in order to find how to get their intentions expressed. This is because the 1st creator was facilitated in his crime of "creating new facilities without creating new landmarks" by the nature of the language he was provided with — the one imported from the language of mathematics.

Basing our expressions on such inappropriate idioms raises risks at all levels of design. For example, (Elliott, 2007) shows an example of this idiom surfacing at the user interface level of an application — when two elements are combined, they vanish, to be replaced by their composition, never to be recovered. Since the author, a programming language expert, regards this as a normal and virtuous design idiom, he sees no problem using this as the core building primitive exposed to users.

6. Some Practical Directions and Inspirations

Given that some of the central structuring idioms and metaphors for software construction, function composition and function application, are informationally faulty, we consider how they might be reformed.

6.1. Landmarks Rather Than Mazes

We argue that any author in the kind of terrain of "real software" that we are imagining should be facilitated by the natural modes of expression made available to him in his creative tools, to create new

landmarks that more or less keep pace with his rate of creating new facilities. This is a necessarily imprecise statement — since it might well be burdensome for a new landmark (that is, a new named feature) to appear for every act of composition in the environment. This would lead in the extreme to a language similar to that used by Borges' "Funes the Memorious" who used the entire catalogue of his perceptions as counting numbers. However, the opposite extreme that we just considered, that of "blind function composition" is clearly poisonous since it provides no means at all to create these landmarks — the only possible landmarks are the functions themselves (such as h and f) rather than the application point of the functions.

One way of seeing the problem and possible solutions can be taken from the world of web programming, and the use of the DOM to represent a tree of nodes constituting the state of a web UI rendered in a browser. The "blind function composition model" is analogous to the unreformed way in which developers of the 90s would be encouraged to navigate "blindly" around the DOM as a raw tree of nodes, using constructs such as `myNode.parentNode.parentNode.parentNode` expressing the "incidental knowledge" that the node of interest "just happened to be" 3 levels of containment higher in the tree[14]. Compare this with the "incidental knowledge"of the Lisp programmer above who "happened to know" that he was dealing with a composition of exactly two functions, the second of which he had an interest in. This kind of "blind navigation" is extremely brittle in the face of acts by collateral creators in the same space. In the "power-hungry" model we are describing in this essay, the natural response to this situation is to try to seize more power, by finding ways to exclude other creators from the same space, rather than trying to find ways of coexisting with them. The classic embodiment of this power-hunger in the domain we chose for our analogy, the world of DOM programming, is the current drive towards Web Components (W3C, 2014), an innocent-sounding name for a fascistic domain in which the rights to navigation of the DOM by 3rd parties are eliminated. This is the form of solution that would be blessed by a proponent of "computational thinking" — it tries to eliminate a problem by seizing more control.

6.2. Inspiration from the Web - CSS Selectors

A more appropriate kind of solution to this problem can be seen in the strategies actually chosen by web designers in the last decade — who, unlike many computer scientists, have internalised the fact that they must find ways of getting on with each other, as well as those with different skills and interests. Web designers have by and large moved over to the use of (CSS) selectors in order to identify parts of a document of interest, rather than either i) relying on blind navigation rules and/or ii) trying to find ways of expressing unilateral control over all aspects of the document structure. These selectors are strings with a reasonably simple format, which are able to express decisions about the identity of pieces of the document that are of interest, that are expected to remain reasonably stable with respect to evolving structure in the document at the hands of a community of related creators.

6.2.1. CSS Selectors as a Negotiated Space

I want to note three key aspects to the stability of reference of CSS Selectors as experienced within their communities of use. Firstly that the stability is only "reasonably good" rather than being absolute or to some provable standard — and that it is one that results from some process of "negotiation" with a group of other creators. Secondly, it is enabled by certain kinds of substructure — in particular a facility for supplying supporting names in an open way to an underlying collection of things — in this case these names take the form of CSS class names which can be freely applied to the DOM nodes supporting the space of selectors. These are "open" in that any creator can supply further names to any node they are interested in — assuming that they are happy with their quality of communication with the other creators that they are cooperating with. Thirdly, the stability is "opportunistic" — that is, each creator can choose between a variety of tradeoffs in the strategies they use for writing selectors — ranging from i) "chancing their arm" on existing aspects of the DOM structure without using class names, ii) piggybacking on some existing collection of names operated by another creator for some purposes which they judge sufficiently related, to iii) deciding that they need to take control of a new collection of names of their own.

[14] Sadly as guides such as (Nativ & Fankhauser, 2009) demonstrate, this technique remains in vogue into our current decade.

Now, to a proponent of "computational thinking" this kind of messy negotiated process is simply anathema. A computational thinker is not satisfied with anything other than completely predictable results within previously agreed bounds — and is the kind of person that are seen regularly over the past 20 years declaring that "the web is broken" (Tiselice, 2015) when encountering these kinds of "negotiable solutions" rather than the "closed boxes" which their training and mentality have brought them up to expect. These negotiable solutions are in fact highly successful adaptations to the problem posed by a space in which multiple creators have to cooperate — the space of real software.

6.2.2. Selectors within Infusion

In the Infusion framework, we take a leaf out of the book of web designers and apply a highly similar solution to the problem of stably naming and identifying pieces of an implementation in an unstable or shared environment. Our IoC configuration system allows selectors in the form of IoCSS strings to match onto one or more pieces of an application, guided by their ability to match onto one or more context names as landmarks. Similar to CSS class names, these context names form an open system in that any creator may freely contribute any number of names of their own onto any existing artefact. We discuss our inspiration and use of IoCSS in (Basman, Lewis, & Clark, 2017).

In this way, we facilitate creators to create and employ landmarks, without which they would become lost in an unfeatured maze of expression trees or function applications. These "mazes without landmarks" are traditional features of languages which promote the use of unbounded recursion in the designation of artefacts — that is, those which allow grammars which permit an infinite number of sentences to describe a single artefact.

6.2.3. Landmarks as Secondary Notation

This notion and use of landmark names has an interesting status in the powerful Cognitive Dimensions of Notations framework (Green & Blackwell, 1998). Such landmarks are interesting because they could be said to occupy a position intermediate between what are called in that framework primary and secondary notations. They are intermediate because they are not primarily functional — in many cases, the entire edifice (considered as a single design) could function without them, encoding the same behaviour. This, as well as the fact that they can be freely added and removed from the structure supports the view of them as secondary. However, they are not purely secondary because without them, certain crucial uses of the artefact could not be made — that is, it could not be properly adapted into an ecology of related artefacts managed by related creators without them. They are a kind of "secondary notation with teeth". Many of the cognitive dimensions come to have a freer meaning once one steps back from considering a single program written by a single creator (or a group compelled through Computational Thinking to behave as if they had no individuality), to considering an ecology of real software maintained for a real community — that thing which we imagine could be created. We have considered some of the implications of such collaborative dimensions in (Basman et al., 2015).

6.3. Inspiration from Optics and Reversibility

Another fruitful source of better analogies for building "real software" is the world of optics, rather than mechanics.

6.3.1. Lenses rather than machines

When dealing with light, we accept that it is going to go its own way, rather than trying to find ways of stopping it, packaging it, and manipulating it. In optical systems, components such as prisms and lenses are used to divert and redirect light as it passes from place to place — with the general expectation that the operation of the component is broadly, if not perfectly, reversible in that the effects of one such component can typically be undone by another one. In fact Newton's Experimentum Crucis (Takuwa, 2013), proving that white lights are mixtures, and that only certain coloured lights are pure, directly took the form of "inverting" the action of one prism on a beam of light with another. This reversibility results from a crucial property guaranteed by the laws of optics, that the path traversed by any individual ray of light could be perfectly traversed by one travelling in the opposite direction. It is this interesting property which led to the centuries of confusion only dispelled by Alhazen as to whether the faculty of vision

operated by rays that were emitted from the eye in order to strike objects in the world, or conversely by rays collected by the eye which had been scattered off the objects.

This form of analogy currently has an embodiment in the Bidirectional Programming model (Foster & Pierce, 2009). We believe that such a model is crucial to delivering on many of the core facilities of real software. For example, the last power viii) from section 4.1 granted to users, the "power to resist change", can be seen to require this kind of model — as well as our key idiom of "working on software by means of itself". Let's try to imagine what this entails in practice: in practice, the user is presented with a surface to a piece of software, that exists in both space and time. This surface constitutes the user interface of the software as the user operates it, as it exists from moment to moment. Presented with some behaviour on its surface, some real software would allow the user to express an intention directly coordinated with it: for example, the user might say "I don't like this; make sure I never see this again" — or conversely, "I like this; make sure this never changes". Without the ability to directly correspond all behaviour exposed on the surface of the software right down to the lowest-level pieces of state and configuration that the software was derived from, we could never deliver any real software. We must be able to always "reason from effects back to causes". But what this implies is that the operation of the entire software has to be able to be conceived as the action of some kind of lens acting on these inputs — that is, that at any time we can trace the "rays" which lead out from the software to the user back in the other direction to discover their cause, and to allow the user to express their intention relative to them.

6.3.2. Free flow of information through externalisation

All of the "fake software" we have today is not like this, and does not allow this kind of reasoning. Instead, it consists of a number of "locks" through which water only flows in one direction: conducting power out from the worlds of developers into the worlds of users, and accepting no inflow in the other direction. This is precisely what APIs and abstraction boundaries, compilers and modules are designed to achieve — to concentrate power in the hands of those who have it, and to ensure that none of it leaks outwards. This falls into the "machine analogy" that we identified starting section 6.3.1: the precious resource is controlled by stopping its flow and allowing it to move only in controlled packages from place to place. Typically the person who defines the rules by which the resource is packaged has little motivation to draw up at the same time the inverse roles for unpackaging it and transmitting it in the other direction — because this involves extra work, as well as being anti-religious through giving up the control that they crave. As an example of this, consider how much pointless work is involved in every standard architecture when it is decided that at some point some crucial data structure doesn't just have to exist privately in memory but in fact needs to be serialised to disk or wire in order to be shipped somewhere else. This normally involves a significant redesign as the same people who felt they were virtuous in designing abstraction boundaries have to suddenly scramble to discover how to circumvent them just to meet their own ends. That this work is endlessly being repeated is never interpreted as evidence that the entire enterprise of data hiding is completely misguided — developers are too well-trained in order to perceive this.

The Infusion system includes a direct embodiment of the lens model in its Model Relay and Model Transformation systems[15]. Creators can set up publically advertised bodies of state which other creators are free to attach to, having their own copies of the data available for both reading and writing either in the original or a transformed form. End-to-end, this allows the "rays" of dependency to be traced in either direction across an entire application. We are currently working on the new Infusion Renderer which will allow this transparency to be extended by the final hop into the process of binding behaviour onto markup constituting the interface presented to users.

7. The Information Revolution Hasn't Happened Yet

Wikipedia's noble manifesto asks us to "Imagine a world in which every single person on the planet is given free access to the sum of all human knowledge. That's what we're doing." Our mission is just the

[15] `http://docs.fluidproject.org/infusion/development/ModelRelay.html`

same, only broader. Wikipedia has now authoritatively won the battle against encyclopaedias constructed via centralised and authoritarian models. Its coverage is vastly broader, more up to date, and on average more accurate than that of any competition. But for all of its Internet-age wizardry, Wikipedia appeals to an ancient model of what knowledge is. The structure and content model of Wikipedia would have been completely comprehensible to the Emperor Xuanzong of Tang who ruled China between 712 and 756. The encylopedia which he commissioned, the Tongdian, was itself a compilation of several previous works, and was part of an already established model of compendia which centuries later resulted in the Yongle Encyclopedia of 1408 with its 11,095 volumes occupying 40 cubic metres. This, impressive and useful though it is, is a model for "dead knowledge"- it sits on the page after it is written, and later it is read and perhaps remembered. This is the total of interaction offered by the "encyclopaedic model of knowledge". What we aim to put into effect is a model for "active knowledge" — for which we currently have little name other than the bland catch-all term software — and it's clear that not all software represents knowledge of this type. Active knowledge has behaviour, it is connected to communities and the real world, it has awareness of context and an individual's faculties for producing and receiving information. A model for distilling active knowledge was described in (Winner, 1995), reporting on (Ehn, 1988)'s account of the UTOPIA project within the Scandinavian newspaper industry of the 80s, in which "key insights, lessons and prescriptions must arise from a process in which project members, regarded as equals, join to explore the properties of both technical artifacts and social arrangements in a variety of configurations."

Much is made in the academic and journalistic literature of the so-called "Information Revolution" which is presumed to have coincided with the creation of the Web. However, I think this examination makes clear that this is no true kind of revolution since it has been accompanied by no revolutionary change in our model of what knowledge is, and how it is accessed and represented. Compare this with the Industrial Revolution, which created a model for a vast array of artefacts, modes of transport, machines — machines constructing materials, machines constructing other machines, converting and transmitting power from place to place, all products even whose categories would be hard to comprehend by the readers of the Yongle Encyclopedia. Instead of trying to push our methods into other disciplines, let us instead marvel at the incredible achievements of mechanical engineers who have produced far more substantial physical and cognitive progress even while saddled with the intractable limitations of the physical world. Rather than trumpeting our mental models, let us instead be humble and admit that we have not produced a fraction of a comparable achievement whilst being given a completely free hand to produce any imaginable structures without constraint. When the Information Revolution really comes, you can be sure we'll know it.

8. What We Want

What we want is a new generation of Software Engineers and Computer Scientists, who are willing to give up all their imagined wizardry[16]. Prepared to give up the recognition of their peers, industrial-scale salaries — prepared to work more slowly than they might, as a result of trying to produce work that still has a meaning 3 years in the future. Prepared to admit they have no real idea how to build software and have never seen any. Prepared to both study their colleagues and be studied, to understand what the real meaning of their work is. Prepared to read widely, both in other fields, and in the history of their own — that is, to accept that they are not wiser or have any surer models than their colleagues in other fields,

[16]This possibility is alluded to in a moving blog post from Jonathan Edwards (Edwards, 2013), responding to the selfsame Bret Victor talk that we referred to in section 3. Edwards subverts Victor's ultimately naive narrative of how we might recover the stolen future of computer science through open-mindedness and out-of-the-box thinking, by candidly explaning that our community is going to have to incur real and substantial losses, both financial and moral, in order to reverse the directly wrong-headed course it in fact took since 1973. Edwards' rendition of our point in the main text reads:

> As always, disruption will come from our blindspot. From amateurs and children playing with toys, untainted by the sin of knowledge. Perhaps aided and abetted by a few turncoat hackers rejecting the dark side of super-intelligence.

We can also align our mission with the programme of "subversive rationalisation" announced by (Feenberg, 1995).

or in the past — and to take the time to rummage through the vast trash-heap of Computer Science to sift out the few scattered gems in it. We want a generation ready to build the true Cathedrals of software which will exist — rather than today's imagined Cathedrals (Raymond, 1999) which to any but our own biased eyes are simply shanty-towns built out of any old trash we have to hand, destined to be swept away and built again after the first change in the weather. The builders of real Cathedrals were happy to begin on work that they knew would never be completed in their lifetimes, or even their grandchildren's — how did we come to think we could measure ourselves against these, with our surest building blocks compared to soap bubbles[17]?

9. Further reading

This paper has described the top-level motivations for our approach, and sources and models for inspiration. As we said at the outset, as well as philosophical positioning, this is the blueprint for a practical system that we are in the process of building. The core framework, Infusion is documented at (Fluid, 2017), with its externalised form, the GPII Nexus explained at `https://wiki.gpii.net/w/The_Nexus`. More forward-looking, speculative framework development documents are available on the wiki at `https://wiki.fluidproject.org/display/fluid/Development`.

10. References

Abelson, H., & Sussman, G. J. (1985). *Structure and Interpretation of Computer Programs*. MIT Press.

Basman, A. (2016). Building Software is Not a Craft. In *Proceedings of the Psychology of Programming Interest Group.*

Basman, A., Church, L., Klokmose, C., & Clark, C. (2016). Software and how it lives on - embedding live programs in the world around them. In *Proceedings of the Psychology of Programming Interest Group.*

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

Basman, A., Lewis, C., & Clark, C. (2011). To Inclusive Design Through Contextually Extended IoC. In *Proceedings of the ACM Splash'11 Companion (Wavefront).*

Basman, A., Lewis, C., & Clark, C. (2017). The Open Authorial Principle. In *Proceedings of the ACM Splash'17 Onward (submitted).*

Clark, C., & Basman, A. (2017). Tracing a Paradigm for Externalization: Avatars and the GPII Nexus. In *Proceedings of the 2017 International Conference on the Art, Science, and Engineering of Programming Workshop: Salon des Refusés.*

Edwards, J. (2013). *Leaked transcript of censored Bret Victor talk*. Retrieved from `http://alarmingdevelopment.org/?p=797`

Ehn, P. (1988). *Work-Oriented Design of Computer Artifacts*. Stockholm: Arbetslivcentrum.

Elliott, C. (2007). Tangible Functional Programming. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming.*

Feenberg, A. (1995). Subversive Rationalisation: Technology, Power and Democracy. In A. Feenberg & A. Hannay (Eds.), *Technology and the Politics of Knowledge.* Indiana University Press.

Fluid. (2017). *Fluid Infusion Documentation.* Retrieved from `http://docs.fluidproject.org/infusion/development/`

Foster, J. N., & Pierce, B. C. (2009). *Boomerang Programmer's Manual*. Retrieved from `http://www.seas.upenn.edu/~harmony/manual.pdf`

Gilbert, W. S. (1889). *The Gondoliers – libretto*. Chappell & Co, London.

Green, T., & Blackwell, A. (1998). Cognitive dimensions of information artefacts: a tutorial.. Retrieved from `https://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf`

Hoyte, D. (2008). *Let Over Lambda*. lulu.com.

Lakatos, I. (1978). Newton's effect on scientific standards. In J. Worrall & G. Currie (Eds.), *Philosophical papers of Imre Lakatos vol. 1* (p. 193-220). Cambridge University Press.

Nativ, G., & Fankhauser, M. (2009). *Developing a Web 2.0 application using the InfoSphere Business Glossary REST API.* Retrieved from `https://www.ibm.com/developerworks/data/library/techarticle/dm-0909infosphererest/`

Orwell, G. (1946, May). Second Thoughts on James Burnham. *Polemic*. Retrieved from `http://orwell.ru/library/reviews/burnham/english/e_burnh.html`

[17] Alan Perlis: "Is it possible that software is not like anything else, that it is meant to be discarded: that the whole point is to see it as a soap bubble?", quoted in (Abelson & Sussman, 1985).

Raymond, E. S. (1999). *The Cathedral and the Bazaar*. Retrieved from http://www.catb.org/~esr/writings/cathedral-bazaar/

Shah, I. (1998). *Knowing How to Know*. Octagon Press.

Takuwa, Y. (2013, December). The Historical Transformation of Newton's experimentum crucis: Pursuit of the Demonstration of Color Immutability. *Historia Scientarum*, *23*(1).

Tiselice, D. (2015). *Web sucks and here's how we can make it awesome.* Retrieved from https://www.presslabs.com/blog/web-sucks-how-to-make-it-awesome/

Ungar, D., & Smith, R. B. (2013). The thing on the screen is supposed to be the actual thing.. Retrieved from http://davidungar.net/Live2013/Live_2013.html

Victor, B. (2013). *The Future of Programming*. Retrieved from http://worrydream.com/dbx/

W3C. (2014). *Introduction to Web Components.* Retrieved from https://w3c.github.io/webcomponents/explainer/

White, M. (1997). *Isaac Newton: The Last Sorcerer*. Fourth Estate.

Wing, J. M. (2008). *Computational thinking and thinking about computing.* Retrieved from https://www.cs.cmu.edu/afs/cs/usr/wing/www/talks/ct-and-tc-long.pdf

Winner, L. (1995). Citizen Virtues in a Technological Order. In A. Feenberg & A. Hannay (Eds.), *Technology and the Politics of Knowledge.* Indiana University Press.