# To disclosable computing through concrete abstractions

## Substrate vision statement

Google docs version
Antranig Basman

## What is a substrate?

Jonathan Edwards defines a substrate as embodying the following properties:

1. A complete and self-sufficient programming system,
2. with a persistent code & data store,
3. providing a direct-manipulation UI on that state.
4. Supports live programming.
5. Programming & using are on a spectrum, not distinct.
6. Conceptually unified — not a “stack”.

> Summarized as a slogan: “A PL, DB, & WYSIWYG document unified together.”

Whilst I subscribe to all of these points, in my vision most of them are not essential definitional aspects, but instead essential *possibilities* – that is, that the substrate should be designed in such a way that they can be brought into view or “disclosed” idiomatically in a context where they are relevant.

For example, direct manipulation and live programming may not be appropriate or necessary for many users of a particular substrate's deployment – they may prefer to view it as a regular application, indistinguishable from one not built on a substrate, or even as a static document. But the path to bringing these capabilities into view should be reasonably direct and not involve a fundamental change in the structure of the application.

## Substrates and Malleability

Out of these 6 properties, to me the definitional point is #5: “Programming & using are on a spectrum, not distinct.”. This is also the core intersection with the Malleable Systems Collective, whose principle #1 is “Software must be as easy to change as it is to use it”. Comparing the properties with the collective’s own principle 6:

> 6. Modifying a system should happen in the context of use, rather than through some separate development toolchain and skill set

– I see the same “loosening” of this principle as desirable. Modifying a system should be ***possible and idiomatic*** in the context of use, but this should not also preclude the use of more or less standard toolchains or skillsets that can be used to work on the system by specialists.

## Disclosure and Integration

Boxer is in my opinion the most successful substrate satisfying Jonathan's principles. We can learn a lot from its community. Henri Picciotto, a Berkeley Mathematics educator, wrote in Boxer: A Teacher's Experience (2022) of 19 years of his students' experience with Boxer, that it "defied their expectations of how to interact with their machines", felt "bland and antiquated", and that by the end that they hated it. I argue a big contributor to this is that, despite its many great virtues, Boxer has no model for disclosure of its capacity for computation – the controls for executing and modifying boxes are always visible.

We can't build substantial communities for substrates unless we can use them to build interfaces which are seen as *wholly satisfactory* by communities. This implies that, if deployed as web pages, they use standard layout technologies, can render statically and don't incur appreciable costs on startup. This unpacks the 6th property of a substrate – whilst it does not "form a stack", it should be able to coexist naturally amongst the levels of stacks that exist. This brings in an important strand in the literature, Stephen Kell's notion of an integration domain, described in The Mythical Matched Modules (2009).

In an integration domain,

- languages and tools are specialised towards composition of software, and so do not resemble conventional languages
- relations are expressed between runtime values, predicated on the context in which they occur

Through Stephen's principle of *interface hiding*, dependencies do not explicitly manifest themselves in the domain except through the contextualised values which the domain puts into relation.

## Composition and Components

Central to its role as an integration domain is the model for composition that a substrate establishes. A composition model determines how parts of designs written separately can be combined. Traditionally in software engineering this implies a model for reuse – that it's possible to bring parts of a design written elsewhere into one's own, by referring to them, rather than copying them. Again we can learn from Boxer's community – Andy diSessa has written about the negative impact of a component-based composition model on community agency in Issues in Component Computing: A Synthetic Review. Components brought in by reference are opaque and mostly impossible to modify. This led to Boxer's standard model of "reuse by lithification" – useful code is simply copied into one's world. This is naturally an unscalable approach but an essential one for small-scale communities.

Notions of reuse in traditional programming are tied to notions like "objects" or "types". These need to be completely reconceived in the context of a substrate. Stephen Kell's In Search of Types (2014) covers many of these notions very well – especially two non-orthogonal senses of the notion of an "abstraction".

1. (Parnas et al, 1976) "an abstraction is a concept that can have more than one possible realization"
2. "abstractions as a repertoire of things that we can *refer to*"

We are interested in primarily the second notion[1]. With respect to the notion of types, a popular definition (Krishnamurthi, 2003) is “any property of a program we can determine without executing the program”. If a substrate folds together the contexts of design and execution, this notion of a type largely collapses. Indeed, Jonathan Edwards’ description of Subtext 10 declares “Subtext has no syntax for describing types: it only talks about values” and also “Concrete values serve as witnesses of types”.

My notion of an integration domain, implemented in my work in progress substrate, Infusion, features ***concrete abstractions***. These are blocks of pure state, with a natural representation in JSON, which are treated as aligned ***layers***. Rather than being composed at build time in the machinery of a compiler as types or classes, and perhaps largely erased at runtime, they are composed in the running substrate in a visible way, with the resulting merged structure allowing access to the provenance of each separate layer. The system state is determined by the complete contents of such layers which, since it is intelligibly serialisable, is easy to transport from place to place as well as store in traditional backends such as GitHub or more fruity ones such as ShareJS or Automerge. This solves the *image problem* of some pseudo-substrates such as Smalltalk where the design content of the running system can diverge over time and can only be manipulated as a whole by loading it.

A successful substrate needs to minimise what I call divergence – the discrepancy between its runtime state and the state from which it can be authored. This implies minimising reliance on traditional runtime storage such as the stack and the heap with their coordinates which are meaningless in the visible substrate. Instead, the substrate needs to make it easy to trace ***causes from effects*** – given any piece of the UI, to be able to fully explain the causes that led it to be that way and intervene with them. This is consistent with Michel Beaudouin-Lafon’s role of an information substrate in Towards Unified Principles of Interaction (2017). A related treatment is Don Norman’s Gulf of Execution as discussed in Jonathan’s Subtext: Uncovering the Simplicity of Programming (2005).

## Errors and Asynchrony

A successful substrate needs to solve several other problems for its users. Firstly the notion of **errors**, especially what were once design-time errors, need to be surfaced in the substrate as it runs. Boxer’s model for this is a good example – a faulty reference for example results in a message displayed on the surface of the substrate which is then navigable to the site of the error. Common reactive libraries offer little support for recognising and propagating these errors, as well as tracing them back to the part of the substrate responsible.

Secondly, we need to deal gracefully with **asynchrony** – both in terms of operating on asynchronously available data, as well as asynchronous demands for “code” within the substrate as it evaluates. Successful user programming systems do not bother the user with issues relating to whether values are available right now or require I/O which again stems from the faulty reliance on the program stack underlying runtime state. Traditional programming languages make this a viral issue affecting the semantic of the whole codebase as per Bob Nystrom’s What Color is Your Function.

---

[1] Although I do see a role for “opportunistic abstractions” in terms of spotting the “coeffect image” of a replaceable unit of configuration’s unbound references and helping the user to see if another unit would fit. This is a kind of dual of the role of “opportunistic types” emerging through looking at the structure of concrete value witnesses.

Interestingly it seems like both of these issues can be dealt with under a common scheme. We allocate a special kind of payload to a reactive function, an ***unavailable value*** which accumulates the addresses in the substrate which are responsible for *design incompletion for any reason* – e.g. whether the syntax underlying the substrate is incorrect, or a I/O request is pending. The reactive graph short-circuits on these values, accumulating their payloads much as exception handlers did in conventional languages. This allows the user to continue working with those parts of the substrate which don't depend on these unavailable values as normal, whilst being able to direct their attention to the addresses where the design might need to be corrected if necessary – again, in allowing causes to be traced visibly from effects.

## Why improve notations if all code will be written by AI?

I argue that the time has never been more favorable for the substrates community. Rather than representing programming as a "solved problem", LLM generation of code heightens existing problems of code oversight and management of technical debt, as well as offering new opportunities. As I write in An Era for New Notations, notations which make it easier to determine whether a code structure aligns with the intentions of a community by minimising divergence are more attractive than ever, as well as the incumbency advantages of existing notations being diluted through the availability of quick and reliable LLM translation.

## Upcoming challenges:

Better reactive primitives:

Whilst miniAdapton of Hammer et al (2016) seems to offer somewhat more forgiving semantics than current "best of breed" JS signals implementations in the case of dynamic allocation of signals during a computation, it feels like there's a lot of room for improvement in this area, re. issues such as supporting writeable computed values, supporting cyclic graphs of reactive values and/or bidirectional relations. These cases are coming up a lot in Infusion development. Likely there are ideas in Jonathan's Coherent Reaction that can be applied.

Better layout primitives:

In the spirit of "living within the stack with stack goggles" I would like to see some scheme for gracefully embedding a more humane layout system within CSS. Systems such as CSS0 are far too primitive, yet full-blown CSS frameworks I've looked at are prohibitive at the user level. A system such as LayoutIt can spit out some Bootstrap definitions given some visual tinkering but it is closed source. Cassowary-based constraint systems such as GSS are promising for people willing to leave real browsers behind.

References:

Beaudouin-Lafon, M. (2017). *Towards Unified Principles of Interaction*. In *Proceedings of the 12th Biannual Conference of the Italian SIGCHI Chapter (CHItaly '17)* (pp. 1–2). ACM.

diSessa, A. A., Azevedo, F. S., & Parnafes, O. (2004). Issues in Component Computing: A synthetic review. *Interactive Learning Environments, 12*(1–2), 109–159

Edwards, J. (2005). Subtext: Uncovering the simplicity of programming (OOPSLA '05) (pp. 505–518). ACM

Edwards, J. (2009). *Coherent Reaction*. In *Proceedings of (OOPSLA '09)* (pp. 925–932). ACM.

Fisher, D., Hammer, M. A., Byrd, W. E., & Might, M. (2016). *miniAdapton: A minimal implementation of incremental computation in Scheme*. arXiv. https://arxiv.org/abs/1609.05337ResearchGate+4

Kell, S. (2009). *The mythical matched modules: Overcoming the tyranny of inflexible software construction*. In *OOPSLA '09)* (pp. 881–888). ACM.

Kell, S. (2014). *In Search of Types*. In *Proceedings of (Onward! 2014)* (pp. 227–241). ACM.

Picciotto, H. (2022). *Boxer: A teacher's experience*. In *Boxer Salon 2022*, part of the ‹Programming› 2022 conference. Retrieved from https://www.mathed.page/t-and-m/boxer-2022.pdf