Version 2.0.0 - October 2021

# Computing Manifesto

## Introduction

Let's begin with a simple question: what is the difference between a computer "user" and a computer "programmer"?

Today we have an innate, almost visceral response to this question. "The users," we tell ourselves, "consume what the programmers create." This is neither objectively true nor does it answer the original question. Part of the problem is that we don't have a good definition for what it means to "program a computer." It cannot just be "the act of creating computer programs," because then we have to say definitively what "programs" are. For example, does making some deep modification to the Operating System constitute programming? Probably. Is it what programmers do? That is both self-serving and circular. If we proceed down this line far enough, the best definition for programming we can provide is "telling a computer what to do." And it is under that definition that personal computing has utterly failed in the present.

The current situation can be summarized as elementary education concerned with teaching children how to read and write, but only through the use of pre-formatted bureaucratic forms. No scribbling on paper, no coloring outside the lines – just letters and words within labelled boxes that have been specified for some purpose that remains opaque to the children. This is the situation we have in computing today.

Given the evolution of computing and the tech industry, both "programmer" and "user" are at best anachronistic and at worst corrosively exploitative. Other than optimizing for a market driven, strict, utilitarian separation, they offer no answers, no framework, and no vision of how computers and people should interact in the future, especially if that interaction is to be symbiotic and empowering. Moreover, in the realm of *personal computing*, they fundamentally undermine the core aspects of personal: ownership, inspection, understanding, and malleability.

Our goal for computing should not be to somehow "make the computers smarter," whatever that means. Besides the ill-defined and moving target, making computers smarter focuses on computers and not people, which misses the potential to understand what “smarter” or “intelligent” might even mean. We are skirting the main point of the exercise while undermining its very foundation.

What we should be focusing on is making the interaction of a person with the computer a smarter experience. We need to look no further than the pioneering media based visions of Kay, Papert, et al, or the collaborative augmentation system of Engelbart, or the expressiveness of systems like GRAIL and Sketchpad to see that much of the origins of computing, especially personal computing, were rooted in these ideas.

We reject the self-serving industry mantra that “people are not smart enough to learn much of anything”  or “want the one tool for the job” or “not everyone wants to be a programmer.” We don't believe that 200,000 years of tool building and altering our surroundings has come to a halt when computers came on the scene. How many times have you heard someone complain about an application with the words “why doesn’t it just do that,” where “that” is a minor permutation or alteration of current setup? How many times have you heard someone begrudgingly use multiple applications simply because what is needed is an overlap of their respective features? These few common examples express the desire for change, but are today met with deadends. With the status quo, how would we expect to make the above market driven assumptions?

Using the systems of today, a “user” only has two options: purchase software that bridges the two applications or learn a full programming language, the complex underpinnings of an operating system, a development environment, and perhaps version control tools. The gap between these two solutions is immense. The latter is akin to telling a novice who wants to learn how to build a coffee table to “learn how to describe wood using physical chemistry,” while the former is like saying “buy a table from Target.”

Building malleable systems is not about making *everyone* a programmer but about giving *anyone* the option to understand and the systems they use (in a coherent progress of understanding). There is no contradiction in building systems which people can change and building specific tools designed for special needs. But with every idea that dies, with every desire that meets in a deadend, there is a personal tragedy.

In order to reduce this gap, we need to rethink the concept of a personal computing system from the ground up. Unlike the array of thinkers that originally took up this task half a century ago, we have the advantage of history: real compelling examples, speculative papers, and the wisdom of our elders. In recreating the computing experience we should make full use of these advantages.

To summarize:

- Systems are not expressive, i.e. environments do not correspond to the flow of expression that is natural to the medium of work.
- Systems are not extensible or malleable, i.e. the environments are not natively configurable to the individual flow.
- Systems are not comprehensible, i.e. they are so large in code and complexity that regardless of technical ability no single individual is able to understand the systems bottom-up.
- Systems are not pedagogical, i.e. they do not constitute environments in which people can actively learn how to manipulate them, and they are not "examples of themselves."
-

Byproducts of the above:
- Computing does not live up to its potential as a medium of thought.
- We allow for technology which changes people but do not allow for people to change the technology.
- A user-centric view of computing has pushed us into a consumption-centric view of computing. In this world the user is both the consumer and the consumed.
- The user/consumer model is naturally anaesthetising and erodes out critical faculties.
- Bloated, unnecessary, complexity and obscurity feedbacks onto itself, creating incomprehensible and costly systems on which we depend.

Our framework for computing is outlined below. In earnest work on the project began summer of 2020 with one year of funding provided by the UnitedLex Corporation. Our initial focus was on the Hypercard-like interface layer (github).

# Authorship

________

________The synthetic and corrosive separation of "users" and "programmers" begs for a new framework where the barrier between creating and consuming, i.e. internalizing, is inherently absent and the transition between the two acts is a seamless feedback loop. For this framework we propose a concept of computing *authorship* for two principle reasons.

First, if computing is to be a medium of thought then it behooves us to look at other such media which have historically made possible qualitative leaps in our understanding of ourselves and the world. There are a number of such examples from music, to figurative arts, but the most obvious is conventional literacy, i.e. reading and writing. Conventional literacy has gone through a similar history of "user" vs "programmer" separation. Starting with  the earliest forms of writing of ancient Sumer around 3200 BC until the arrival of the printing press in the late 15th century in Europe, the act of creating was relegated to "scribes." An entire class of people were given special tools, special training, and special knowledge very much reminiscent of today's coder and hacker elites. This separation broke the creator and consumer link and undermined what makes conventional literacy so powerful: the thinking to writing to reading to thinking feedback loop.

Conventional literacy shares another hardwon characteristic which thrust it into the realm of a personal medium critical in the kind of philosophical thoughtfulness that we value so greatly. This is silent reading. Again it took millenia, until the 10th century, for this introspective practice to be the norm (at least in the west), but it was a necessary dynamic for a deep internalization and ownership of the process. This is when conventional literacy took a large step to being truly personal.

Having proven its worth, conventional literacy became a core part of our social institution, with reading and writing becoming the foundation of our education bringing about the mass literacy of today.

Combined with the copying and disseminating power of the printing press and the centrality of conventional literacy in society, we finally arrived at a medium that was both personal and global, private and collaborative, omnipresent and forgotten, conventional and cultural.

Secondly, thinking around medium and conventional literacy was integral to the development of personal computing. We have to look no further than the "no barriers" and "peeling the onion" systems developed at Xerox PARC, to the democratisation goals of the Homebrew Computer Club with the explicit desire to undermine the "priesthood of computing," to the ATG *Authorship* work at Apple through the 80's and 90's.

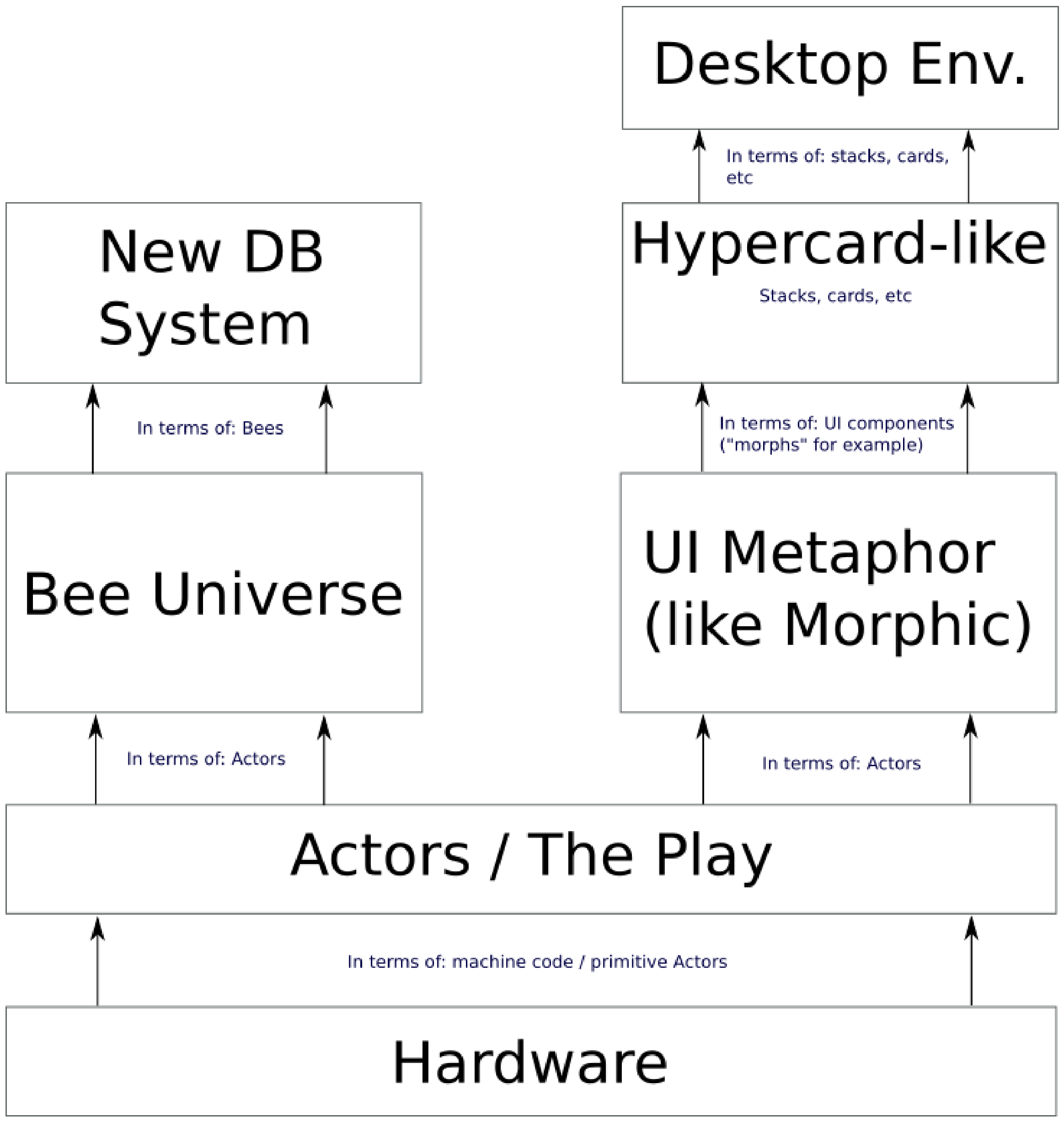
Two canonical examples of such systems are Hypercard, developed by Bill Atkinson and others at Apple in the 80's, and Smalltalk created in the early 70s by the Learning Research Group at PARC led by Alan Kay. But there were many, many others.

Hypercard was centered around a strong spatial metaphor of stacks, cards, buttons etc and an easy to read scripting language. Creators in Hypercard were referred to as *authors*. Smalltalk was centered around a biological metaphor of a message passing between independent entities. It was a critical foundation of what later became known as Object Oriented programming systems (although the current use of OO has little of the original ideas). Both systems had a deep commitment to the "peeling the onion" principle. You could always look inside, change, adapt whether you were an author or a consumer in the system. In Smalltalk you could continue to peel all the way down to the bytecode. Moreover, both systems exhibited strong environments.

*Authorship* for us involves these takeaways:

> Environments empower languages and imbue them with literacy.
>
> Creating (“writing”) and consuming (“reading”) must be well formulated in the medium, and the loop between the two must be seamless.
>
> The system must allow for continual (guided) exploration, with no barriers to subsequent (deeper) layers of understanding.
>
> Metaphor, or a structure of metaphors, is a powerful organization framework for any such system.
>
> For computing to achieve its potential as a medium for thought, it must ensure that people get to a place where they can forget about the medium in which they are thinking.

## Layered Metaphors (The Allegorical Machine)


Desktop Env.
In terms of: stacks, cards, etc
New DB System
Hypercard-like
Stacks, cards, etc
In terms of: Bees
In terms of: UI components ("morphs" for example)
Bee Universe
UI Metaphor (like Morphic)
In terms of: Actors
In terms of: Actors
Actors / The Play
In terms of: machine code / primitive Actors
Hardware

Metaphor is the central unifying “feature” of any system we build. It pales in the face of technical specifics and implementation details. A powerful metaphor, realised in a supporting environment, provides both the framework for development but also the understability of the system. It is immediately universal and, in our view, the most difficult aspect of building empowering technology.

Given the complexity -- and no clear uniformity -- of the many aspects that make a computing system come to be, we chose to separate the different “layers” into their own self-contained metaphor based systems. In turn, each layer allows for junction, i.e. a way to peel away the hood and jump into the next environment or world. The diagram you see is an overview and each layer is outlined in more details below.

As of writing, we have begun work on the *Hypercard-like* layer..

## Hypercard-like Layer

*Simpletalk* is a language and environment for both creating and catering software interfaces. Deeply inspired by the clarity of Hypercard and the object and message oriented ideas of languages like Smalltalk, Simpletalk removes all barriers between the "users" and the "programmer" by creating an expressive authoring system. Work began in the summer of 2020 and a prototype version was released in the summer of 2021. The project is open source (https://github.com/dkrasner/Simpletalk) and a live version can found at https://simpletalk.systems

*Simpletalk* is a vocabulary of parts. These include "spatial" or organizational parts such as a *world* which is made up of *stacks* which themselves are made up of *cards*, as well as *windows, areas, buttons, images, browsers,* and so on. All parts are defined by their properties, which describe everything from appearance, layout, behavior and relationships, and these are fully inspectable and malleable.

Communication is handled via message passing. Parts can send messages to each other via pre-defined relationships (example, a part can be the target of another) or direct part identification (example, "first button of area 'My Area' of card 3"). In addition to directing others, parts are aware when they themselves are changed. This creates a rich and dynamic environment which is supported by a highly readable language.

Properties essentially define the state of the environment and any of the parts. This makes it easy to import and share anything built in *Simpletalk*. In addition to importing entire worlds, *copy & paste* are core messages which each part understands. Hence, anything from a stack or card, button, menu, or music player can be copied, shared, and changed.

Moreover, *Simpletalk* is written to be run on the most ubiquitous VM today - the web - making it as accessible as it is natural and understandable to people in the world.

Next steps for the environment include the following:

> Addition of parts such as *spreadsheet* to the vocabulary.
>
> Extension of the language to include more powerful mathematics as well as string manipulation.

Shared environment live collaboration.

Continuing to bring *Simpletalk* outside of the physical machine with video recognition, projects, spaces.

A *Simpletalk* which is constraint based in the spirit of Ivan Sutherland's *Sketchpad.*

A *Simpletalk* environment to write other *Simpletalks.*

Clear transition between the *Simpletalk/HC* layer metaphor to the UI/morphic-like layer metaphor, i.e. peeling open to the layer below.

## UI

The next layer below HC/authorship one. Peeling away would bring you down to the interface layer allowing change of fundamental aspects of that interaction.

The interface layer would include graphics, peripherals (mouse, keyboard, stylus etc), and other physical interactive layers such as video recognition and project. The UI layer would be a self-contained system with its notion of object, language, communication centered around a consistent metaphor.

Peeling away would bring us down to the Actors/The Play layers of the computing stack.

## Storage (any store)/DB

The ability to store any kind of data, formats, etc whether we know of them now or don't

New types of stored object should self-associate with already known ones if possible

The world is not an inventory and exploring the world is not a record lookup.

The most prevalent systems of storage today seem to be unaware of this very fact. Outside of the situation where we are dealing with a priori defined inventory, these databases exhibit inflexibility or complete inability to integrate new information and data types, and do force lookup into a deterministic framework for which contradict how we process information in general and openly hide the contradiction that you must be able to determine how to find something that you do not yet know.

Let us consider the SQL database as one example (there are many others, but they all exhibit many or all of the same behaviors as SQL). It is the most popular database type to date and is the foundational software layer of information driven sectors such as health, finance, records and compliance, libraries, law and many others. The interface to the SQL database is the SQL query, a deterministic amalgam of boolean conditions which define a subset of information, i.e. either an information entity satisfies the condition or it does not. Over the last two decades much time and energy has been spent optimizing and scaling the performances of such stores and adding a continual stream of features such as basic NLP functionalities for text heavy fields.

But when is deterministic lookup an effective interface to information? Outside of inventory, when do people think and work deterministically? When has this been an effective system of exploration and knowledge in the history of human thought? Our internal processing is inherently associative, potentially modelled by Baysian inference or a statistical framework in that spirit. We don't think deterministically, we don't learn this way, we don't discover this way.

If running the query is step 1, then there is only step 1. Such databases undermine the essence of discoverability. With every step forward the next one is a return to running a slightly modified version of your query.

Incorporating new records, not to mention previously unknown types of information, is either impossible, awkward to the point of futility or requires restructuring and reindexing. Just walk yourself through the amount of work it has taken to adapt to the transition of basic inventory, richer text documents, rich text with richer metadata, text messages, images, audio, video in these systems.

Compare that to how we adapt to storing and working with new information. The paradigm is completely different. We do not think about what it is and how to store it. We think about what can be done with it. Text can be read; music can be listened to; video can be watched, etc And many of these things can happen in an enriching context, we can refer to as metadata. New information and information types enrich, they do not inherently undermine existing structure and networks of understanding.

Moreover this is nothing new!

The pioneers of personal computing made many of these same observations in their own time. For example, Douglas Engelbart's NLS had associative indexing built-in at the OS level -- it was *not a software feature*, but a fundamental component of the system itself. Other systems like Sketchpad, Smalltalk, Hypercard, and more demonstrated holistic and effective approaches to some of these core problems, even though their lessons are not mainstream today. But even earlier, 100 years ago now, Vannevar Bush in the seminal *As We May Think* article called for associative trails of information and knowledge.

We propose that database systems exhibit the following:

> Currently unknown and unforeseeable types of information can be stored without undermining existing structures.
>
> Lookup is associative and associations can be made across any and all information in the system. Associative trails in the sense of Bush and Engelbart allow for structured contextual linking of information. These trails are in themselves associative and explorable in the same manner as all data in the system.
>
> Continual discovery and exploration are fundamental to the system API. There are no deadends and only in the very extreme scenario do you return to step 1.

## Bee Hive

At some point in the late 60's/early 70's Alan Kay realized that "if you want to have anything useful, you can't go lower than a computer." IE every object, everything should be some form of a virtual machine, a shell that manages and changes itself as needed. This was one of the ideas that gave rise to the OO *Smalltalk* language, where every object was completely self-contained/described (within its environment) and communicated asynchronously via message passing. Although a VM network version of *Smalltalk*, with each object a full virtual machine, was never realized (at least to our knowledge), this biological metaphor of cell communication gave rise to a very powerful, expressive and stable environment. In many ways *Smalltalk* was not about objects but about messages, so a more appropriate term could have been *Message Oriented.*

We take inspiration from another biological system that offers elements of delegation, hierarchy, stability, self-preservation, and regeneration. This system is the beehive.[1]

The hive system framework:

- All nodes in the systems are Turing equivalent (virtual machines).
- Any node can communicate with any other via a message passing protocol.
- Nodes come in different types: *cleaners, guards, workers*.

[1] Although many ant-descendant insect colonies will share these properties, we have a special liking for bees.

    - cleaners are responsible for system management: "garbage collection," health assessment of nodes etc
    - guards are responsible for system security: communication security, node corruption, etc
    - workers are responsible for various core system processes: storage retrieval, computation, etc
- Prototypical data for each node type is stored in each node, i.e. any node has the basic materials necessary to create another, but not necessarily the means or the right to do so.
- Under certain conditions (duration, need) nodes can "graduate" to others, attaining their attributes (data and processes). For example a cleaner can graduate to a guard or a worker.
- At any given time there is one special node which is the *Queen.*
    - The *Queen* is responsible for life in the system. This is the only node that can make new prototypical nodes.
    - All system-wide processes (example, distributed processes) are handled via the *Queen*.
    - The *Queen* can call for resources to be created (example, more *cleaners* or *workers*).
- Although the *Queen* is responsible for new node creation, it is the workers who decide what type of node it will be (*cleaner, guard, worker* or *Queen*!*)*, i.e. the community, decides with which type of attributes (data and process) to imbue any specific node.
    - This guarantees a central point of computing resource generation, but a network wide distribution of resource allocation.
- The system, community, can "decide" to remove an old *Queen* and replace it with a new one.
    - This can be performed with a voting mechanism or a catastrophic event, such as *Queen* node failure.
    - A *Queen* itself can decree to be replaced.

## Actors

We believe the lowest layer of a new computing system should be designed in terms of **Actors**. The Actor Model was first described by Carl Hewitt in the early 70s. Actors are somewhat similar to the original idea of OOP's "objects", in the sense that they are the primitive material of computation and they interact via message passing. What distinguishes Actors from traditional OOP is that each Actor is itself a unit of computation. The minimum required capabilities for any Actor are:

1. It can send messages to other Actors;

2. It can process its own messages and determine what behavior should result;

3. It can create other Actors

What might not be clear from this description, and so should be stated outright, is that Actor messaging is always and completely asynchronous. An Actor does not by default "wait" for a reply from a message it sends, nor does it have to respond to every (or any) message it receives. In fact, there is no concept of a "response," really – just another message send back to the original sender. Because any message can be sent, and it is up to the receiver to determine how to handle it, one can create highly fault-tolerant systems (see Erlang and its OTP).

Additionally, such a system is inherently set up for parallel computation. Each Actor is akin to what we might call a "process" (more on this in a bit). This is a particular advantage in the present computing world, where Moore's Law is a thing of the past and the pervasively networked character of computing is essentially asynchronous.

But for our purposes of rethinking personal computing and user interaction, we don't necessarily care about things like "parallelization" or performance, at least not as first-order concerns.

The real advantage of Actors, then, comes in terms of the metaphor that it (partially) employs. If our lowest layer of the computing system is made up of and described as a collection of Actors and Actor-like things, we can use other metaphors from the world of theater and film (see below). Because, like computing, theater and film are systems that themselves can be used to describe other self-consistent worlds, the a low-layer system of theatrical/film metaphors satisfies one of our requirements:

> As an ensemble of metaphors, is flexible enough to be able to describe the layers that will be "above" it

In an important sense, this lowest layer ensures that "all the world's a stage."

## Example Actors and Metaphors

To make this concrete, we will describe some examples of different types of Actors on a very low level system in this section.

### *`Actor`*

The `Actor` is simply the "base class" (though we need not necessarily have classes per se) of all other kinds of things in this low-layer.

### *`Director`*

A `Director` is a type of Actor that activates other Actors on a given computer processor / core. In other words, it "turns over" computation on the processor to a given Actor in its own collection of known Actors (it yells "Action!"). When an Actor is "activated," it can process the various instructions that correspond to its own behaviors. In computing machine terms, this means executing CPU instructions that correspond to its current behavior. In general terms, this means "acting" on the "script" that currently defines its behavior.

Hopefully you can see the advantage here. A `Director` comes to replace the concept of a "Processor" (or core) at the lowest level. We know what Directors do – they tell Actors what to do, when to start, and they also yell "cut" when things need to stop.

One can also imagine a multi-processor, multi-core, or even multi-machine environment made up of several intercommunicating Directors (think: second-unit directors, etc), each working with different sets of Actors at a given moment. This is easily accomplished because, if you recall, Directors are Actors themselves, and can asynchronously send messages and determine how to respond to messages that they receive.

### *StageManager*

A `StageManager` is an Actor responsible for allocating where Actors and other parts of the set should be and how they should be organized. It is the equivalent of a Memory Management Unit or Memory Management System.

When new Actors are initialized, the `StageManager` knows how to store them into memory. When a `Director` turns over computation to a given Actor, the `StageManager` knows where that Actor is and how to get him to the right place. When messages are sent to Actors, the `StageManager` knows how to find the right Actor and where the message lives in memory.

Here we can imagine several different kinds of stage managers, as well as drop-in replacements for experimenting with different kinds of memory management.

### *CentralCasting*

`CentralCasting` is a kind of Actor that aids in the initialization of Actors, as well as determining when an Actor is "done for the day" and no longer needed. We can think of it, in part, as the equivalent of "garbage collection."

`CentralCasting` works in concert with the `StageManager`.

### *Others*

As we experiment and push these metaphors forward, we expect we will come up with other kinds of base Actors for the system.

## Hardware

In order to realize an initial version of this idea, we propose to implement a small version of this lowest layer (Actors or the Hive) in actual machine language on a CPU whose design we can understand and reason about.

Though the ideal scenario would be to design certain hardware for such a system, we think for the time being we can stick to existing standards. We will use the RISC-V as our baseline instruction set, and that we implement the first version in RISC-V accordingly.

The advantages of RISC-V are:

1. Open ISA – It is well described in documentation and there are no secrets;

2. Comprehensible. The ARMv8 ISA manual is several thousand pages long. The RISC-V ISA manual, with all current extensions, is just a couple of hundred;

3. Open Hardware. There are now and will be open hardware systems that implement RISC-V, including so-called "softcores" that implement RISC-V systems in FPGAs. These will allow us to experiment on real hardware.

### *A Note on RV, Actors, and the Internet*

Another advantage of starting from the ground-up is that we can make assumptions about the "natural environment" of the computing landscape that, perhaps, the designers of teletype-based systems half a century ago could not have.

One of these assumptions is the pervasiveness of TCP/IP as a standard. In the 70s, 80s, and even early 90s, it was not clear these protocols would be the standard for the "world internetwork." Today, it goes without saying. Therefore we should design our lowest-layer with TCP/IP in mind.

One possible way of doing this is to say: the addresses that something like `StageManager` uses to locate Actors can be specified in terms of IPv6 addresses. The "local" block of addresses can correspond to Actors on the local machine, while the "remote" addresses are references to Actors somewhere else on the Internet. Because Actors are asynchronous message passing systems, we need not worry at the metaphor layer about the boundary between local/remote. We could even implement TCP/IP in an FPGA co-processor, specify it as some kind of Actor (`TransportationManager` or something), and handle the receipt and sending of messages using such an Actor. It is also worth noting that the RISC-V spec has an extension for integers/registers of 128-bits, which happens to correspond with the length of IPv6 addresses.

### *Proposed Steps*

Initial proposed steps:

1. Design an initial suite of Actors/metaphors for the "lowest-level";

2. Implement the RISC-V ISA and a simulator in an environment that is dynamic enough for us to work in quickly and easily;

3. Attempt to implement our design in terms of RISC-V instructions, and test in our simulator

4. Re-evaluate and repeat

## Reading and Viewing List

The reading and viewing list is by no means comprehensive. It represents an entry point in the vast literature surrounding computing, technology, learning and related topic. Given that this is a story of humans and technology, organizing these into categorical sections is rather futile, but we have tried to give some "top-level" names for ease of browsing. Arguably this is a mistake and the works here should simply be absorbed in no specific order and organized as seen fit in our receptive minds.

### Computing

As We May Think, Vannevar Bush (1945)

Man-Computer Symbiosis, JCR Licklider (1960)

Sketchpad (PhD Thesis), Ivan Sutherland (1963)
Sketchpad Demo, Ivan Sutherland (1963)

GRAIL (1969)
wiki and report papers there in
demo

Project Summary Report, Doug Engelbart (1962)
Mother of All Demos, Douglas Engelbart (1968)

Spacewar, Rolling Stones, Stewart Brand (1972)

Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age, Michael Hiltzik (1999)

The Dream Machine, M. Waldrop (2002)

What the Dormouse Said, John Markoff (2005)

"Microelectronics and the Personal Computer," Scientific American, Alan Kay, 1977
"Computer Software," Scientific American, Alan Kay, 1984
User Interface, A Personal View, Alan Kay (1989)
The Early History of Smalltalk, Alan Kay (1993)

The Evolution of Smalltalk, Dan Ingalls
Yesterday's Computer of Tomorrow: The Xerox Alto, Dan Ingalls (2017)

Hypercard, Bill Atkinson et al (1987)

Viewpoints Research Institute, STEPS final report (2012)

## Media, Technology, Learning and Thinking

Personal Dynamic Media, Adele Goldberg and Alan Kay (1977)
The Future of Reading Depends on the Future of Learning Difficult to Learn Things, Alan Kay (2013)

Mindstorms, S. Papert (1980)

The Pattern on the Stone, Daniel Hillis (1998)

The Society of Mind, M. Minsky (1986)

Inventing the principle, Bret Victor (2012)
*Media for Thinking the Unthinkable*, Bret Victor (2013)

A Small Matter of Programming, Bonnie Nardi (1993)

The Second Self, S. Turkle (1984)

Understanding Media, M. McLuhan

Cybernetics: Or Control and Communication in the Animal and the Machine, N. Wiener (1948)
Human Use of Human Beings, N. Wiener (1950)

Computer Power and Human Reason: From Judgment to Calculation, Joseph Weizenbaum (1976)

The Aims of Education, Alfred North Whitehead (1929)

## Technology, Science and Civilization

The Printing Revolution in Early Modern Europe, Elizabeth Eisenstein (1993)

The Information Age and the Printing Press, James Dewar (1998)

Orality and Literacy, Walter Ong

Understanding Reading: A Psycholinguistic Analysis of Reading and Learning to Read, Frank Smith (1971)

The Act of Creation, Arthur Koestler (1964)
The Sleepwalkers, Arthur Koestler (1959)

Tacit Dimension, Michael Polanyi (1966)

Sciences of the Artificial, H. Simon (1968)

The Structure of Scientific Revolutions, T. Kuhn (1962)

Eclipse of Reason, M. Horkheimer (1947)

The Machine in the Garden, L. Marx (1964)

Science and the Modern World, Alfred North Whitehead (1925)

The Question Concerning Technology, Martin Heidegger (1954)

Novum Organum, Francis Bacon (1620)