A Book Written in Types

Working title

bugabinga

draft

Contents

Preface 3
1 Machine and program 5
2 Information and entropy 6
3 Knowledge 7
4 Software and its forms 8
5 Operating systems 9
6 Learning the craft 11
7 Beauty and mechanical sympathy 12
8 The cycle and its costs 13
9 The practices 14
10 Intent and product engineering 15
Open questions 17
Agents 18
Citations 19
Glossary 20
Prelude reference 21
    Callouts 21
    Defined terms 21
    Output-specific content 21
    Code from real files 22
    Everything else 22

Preface

This book has no overarching thesis, and that is a decision rather than an omission: it is a collection of one programmer's thoughts about software, organized and sharing a vocabulary, and a proposed spine running from encoded knowledge to loss at every hop was considered and declined. Part I lays a foundation – machine and program, information, knowledge, software, the operating system – and Part II is about the craft that sits on it. The foundation is living rather than finished, and is meant to be revised when later material strains it.

Part I: Foundations

Chapter 1

Machine and program

Take “machine” in the abstract sense – a transformation of input into output, not gears – and the distinction between machine and program dissolves entirely; the universal Turing machine makes that vivid. The chapter argues that the distinction survives in practice for two reasons only, economics and culture: physical machines cannot be built as cheaply as digital ones, and we doubled down on the von Neumann model, which builds the split into the architecture and makes state and location first-class. The lambda calculus is the road not taken. The languages descended from Algol and C are not at fault; they are honest reflections of that machine. That no language names time and space first-class is framing here, not thesis, and the industry’s interchangeable use of code, program, software and application is treated as a symptom of the same unfinished business.

Chapter 2

Information and entropy

Two terms travel under the name entropy and this chapter keeps them strictly apart. Thermodynamic entropy is chaos, whose opposite is the order we here call information: we recognise information because it carries patterns that chaos does not. Shannon entropy is surprise, where maximum information is maximum entropy – the meaning the computer-science curriculum uses, and a different one from the physicist's. Compression and coding belong to this chapter because they lead directly to language: a word is a compressed pointer to a cluster of experience.

Chapter 3

Knowledge

Knowledge is categorically different from information, and is first and foremost an experience. The dead-language book test settles it: information is intrinsic, since the pattern in a book is distinguishable from chaos whether or not anyone can read it, while knowledge is relational and needs a reader with the key. That raises a regress – if the key is itself encoded, it needs a key – and the chapter argues that sensation is not the floor of it. The better word is sampling: digital machines sample, and so do humans. What happens after sampling is inside consciousness and cannot currently be defined; the chapter files that as a boundary of the framework rather than papering over it. Decoders need not be human, which is how to think about agents, and the question of whether they truly decode meaning is left unadjudicated.

Chapter 4

Software and its forms

Software is language in the most general sense – a language of description – and it is the compressed form of knowledge, of which the program is the decompressed form. At execution it becomes information rather than knowledge, because the machine transforms patterns without decoding meaning. Text is only the medium, incidental and merely convenient for us. From that the chapter fixes four words the rest of the book uses: *software* for the squishy, informationy parts that get compiled, used interchangeably with source code; *source code* for all authored input, including config, shaders, models and assets; *program* for the compiled artifact; and *process* for the running program, the moment it becomes physical. The line between source code and asset is technical, not philosophical.

Chapter 5

Operating systems

An operating system is a fiction machine: processes, files and virtual memory are inventions that programmers accept as bedrock, and the everyday concept of "a program" is one of its inventions. The chapter argues that much of the machinery layered on top – hypervisors, virtual machines, containers – is heavy-handed work undoing what the operating system imposed, and that a simpler alternative already exists rather than being hypothetical: the operating system as a library, where you import only the stack you need and your artifact becomes the sole program on the machine. Why that lost is trajectory and historical path dependence, not stupidity. Large systems accrete layers and redundancies instead of reinventing themselves into cleaner forms, as biology does, and there is probably wisdom in that; the yearning for simplicity may just be a romantic feeling. The value of the argument is as a thought experiment: the world does not have to be as it is.

Part II: Craft

Chapter 6

Learning the craft

The craft is essentially not taught. The chapter argues that this is not a total failure of universities – theory and research degrees are valuable and should not be dismissed – but a failure of universities pretending to teach craft, when in fact it transmits by mentorship, guidance, osmosis and, heavily, self-teaching. The evidence offered is one self-taught path: an artist drawing comics on paper, then Photoshop and graphics tablets, then Flash animations and the small scripts in its authoring tool, ending in Eclipse writing ActionScript, with no point at which the artist became a programmer. That path is also the argument: drawing is not intuition and feeling but near-algorithmic concepts trained by studying anatomy, movement, perspective and light, and programming has the same shape – clear rules, thought processes, and elaborate abstract constructs built in thought before being put into the machine.

Chapter 7

Beauty and mechanical sympathy

Beauty is a property of the mental model an artifact induces, not of the artifact, which is why the program matters more than the source code and why an obsession with the medium – text – is a narrowing. Simplicity is beautiful not because it compresses well but because it composes well, is easily maintained, and delivers the promises the software set out to keep. Formatting, naming and style guides are not dismissed; their value is collaborative, homogenising software so that many developers, future self included, can work in it over time. The mental model comes from running the artifact, debugging it, interacting with it and reading its documentation as much as from reading its source. Software is interpreted by machines too, and there it has real meaning, so the designs worth admiring satisfy both sides at once. Unix is the worked example: beautifully composable from the human side, while text pipes tax the machine with constant encoding and decoding and many small programs tax it with spawning – a tension that structured pipes and single-binary shells resolve, their drawbacks being about legacy compatibility rather than design.

Chapter 8

The cycle and its costs

Under the vast day-to-day differences between domains lie three invariants: author, build, ship – to consumers, deliberately, who may be machines. It is not a straight line but a cycle, and problems only become apparent over repeated iterations, which is why every practice in the next chapter exists as a response to a problem the cycle produced. Costs group roughly as authoring over shipping over building, with authoring dominant because human attention, paid in yearly salaries, is the scarce input – and authoring is not typing code but everything that captures product, design and intent, which is why agents cheapening the typing mostly expose how much of the cost was always upstream. Practices move cost between phases, and because the budget is not fixed this is leverage rather than a zero-sum trade; the general test for adopting any practice is to ask what its exchange rate is. Knowing the rate is what observability attacks, and perfect knowledge is both impossible and unnecessary: it is enough to be better than the rest.

Chapter 9

The practices

A practice earns its place in this chapter by being general, which is what admits testing and verification, quality assurance, version control, code review, static analysis and typing, code conventions, dependency management, observability, incident response and hotfixing, and automation – and what cuts formatting into code conventions, infrastructure as code into automation, and feature flags out as a tactic rather than a phase of the pipeline. Each is weighed by the exchange rate of the previous chapter. Testing and verification comes first and carries the most: why to test at all, why test-driven development works in limited cases because exploratory programming has no vision to march towards, why a test is a second and lossy encoding of the same intent, how much testing is enough given the cost of mistakes and the tooling at hand, and how little we can say about whether a test suite is any good.

Chapter 10

Intent and product engineering

Product engineering, requirements engineering, user experience, design and user interviews are one subject: some system by which developers work out with domain experts what to build and why, before anything is said about how to achieve it in code. Tickets and user stories are its artifacts; the specifics of any one workflow are out of scope. The chapter argues that its shape is determined by the distance between engineer and consumer – zero, when building for oneself, collapses into thinking; short, when building for developers, needs only imagination; large distance is a fundamental shift, demanding direct contact or middlemen, and every middleman is another encode-decode hop with loss. That this work is a huge cost driver in most organisations, when it should reduce the cost of authoring, is accidental rather than essential, and the remedy is fewer hops rather than more ceremony.

Appendices

Open questions

What the book leaves open, gathered in one place so that the chapters do not have to pretend to close it: the floor beneath sampling, where consciousness is undefined; what the key to encoded knowledge actually is, and the regress it implies; a language that encodes time and space first-class, deliberately not pursued; whether agents decode meaning, declined rather than answered; a better name than "product engineering" for work that is everyone's; whether authoring stays the dominant cost, which turns on how much of intent capture is irreducibly human attention; whether the yearning for simplicity is romantic; whether awareness of being in an exploratory phase can be taught or only learned; and how to measure the effectiveness of a test suite.

Agents

Large language models are not artificial intelligence in the classical sense of a system that holds a model of the world and adjusts it; they learn, but not dynamically, being pretrained. The recent innovation is the agent: a model given sensors and actuators by a harness, which is a sampling apparatus in exactly the sense the chapter on knowledge gives the word. The claim worth making is that the intelligence is the fine-tuned agent rather than the base model – base models encode knowledge and regurgitate it, and intelligence emerges when a model is fine-tuned to act within an environment, which is also why "thinking models" are better described as having learned how to act than as merely thinking more. The note this is drawn from is parked for later writing.

Citations

Where the book leans on work outside it, chiefly on consciousness, which the chapter on knowledge reaches and then stops at: Annaka Harris on consciousness as possibly fundamental rather than emergent, David Chalmers on the hard problem, Anil Seth, Giulio Tononi on integrated information theory, and Thomas Metzinger.

Glossary

The words this book uses in a fixed sense, collected from the places where the chapters define them: the two entropies kept apart, information against knowledge, and the four words for what a programmer makes – software, source code, program, process. The industry uses code, program, software and application interchangeably; this book does not.

Prelude reference

Everything a chapter can use after `#import "/book/lib/prelude.typ": *`, with its output shown inline. The source of this chapter is the copy-paste source for each helper.

Callouts

`#note[...]`, `#tip[...]`, `#warning[...]` and `#caution[...]` take an optional `title:`; pass `title: ""` to drop the heading entirely.

Note

The default heading is the callout's name.

Custom heading

Any string works as a title.

A callout with no heading.

Caution

For anything that loses data or money.

Defined terms

`#term[...]` marks the place where the book defines a word: a ***functor*** becomes a `<dfn>` on the web and bold italic in print, so a glossary can be generated from the same markup later.

Output-specific content

`#web-only[...]` and `#print-only[...]` include content in one output only.

This sentence only exists in the PDF.

Code from real files

`#snippet(path, tag: none, lang: none)` reads a file under `code/` and quotes one marked region of it. Regions are delimited by `[name]` and `[/name]` inside comments, are dedented, and must exist:

```
#snippet("/code/examples/types.rs", tag: "newtype")
```

Everything else

The rest is plain Typst: headings, lists, `#figure`, `#table`, `#image`, `#link`, `@labels`, `#footnote`, and math. The Typst reference applies unchanged.