

Halo

Drawing Programs in the Normal Form of Code, Data, and Architecture

Forrest Landry

UVSM PUBLICATIONS
Wytheville, Virginia

© 2026 Forrest Landry. Some rights reserved.

Published by UVSM Publications LLC
965 Reed Creek Mill Rd
Wytheville, VA 24382-8287
USA

uvsmpublications.com
contact@uvsmpublications.com

ISBN 979-8-90710-084-8

DOI: https://doi.org/10.67291/SWIA6798

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0).

You may share this work for non-commercial purposes with proper attribution, but you may not adapt, transform, or build upon it.

Full license text: https://creativecommons.org/licenses/by-nc-nd/4.0/

Author ORCID: 0009-0005-6275-0362

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), except as permitted under the license above, without prior written permission of the publisher.

Contents

Introduction iv

Part I. The Normal Form 1

1. The Equivalence Thesis 2
2. The Factoring of Intentionality 18
3. Third Normal Form as Metaphysics; Self-Reference and the Search for a Complete Model of Computation 40
4. Classes and Instances 55

Part II. Halo 63

5. The Language in Five Minutes 64
6. Wiring a Program 80
7. What Counts as Code 107
8. Normal Forms and System Architecture 126
9. Modules as Tables; The Call Triple and Hot Patching 147
10. Code That Lasts; Refactoring Tools and the Cost of Sugar 160
11. Halo in JavaScript; The Module Pattern and the Call Geometry 171
12. Code as Intention; The Hard Problems and the Integrity of Execution 191
13. Drawing the Program; The Axiomatic Form 200

Summary 216

Appendix A. The Rules of a Halo Drawing 219

Appendix B. Hello World, Drawn and Written 227

Terms 232

Bibliography 247

Index 253

Introduction

ABSTRACT

This book answers one question: what single structure do code, data and the architecture of a system share, and how can a program be written in that structure directly? The answer has two parts, and the book gives them in order. Part I establishes the structure. All software architecture can be represented in third normal form, the standard of well-organized relational data, and every software module corresponds to exactly one database table. Three results follow from it. Software engineering is the factoring of intentionality (what the people who make and use a system mean it to do). A community can verify that a system does what it chose only when the system's factoring can be inspected. And a model of computation that means to be general must be able to describe itself in a form a person can read. Part II gives Halo, a language for writing programs in that structure directly. Halo has essentially one symbol, a circle, and two kinds of line, one for the flow of control and one for the flow of data. With them it can represent any program written in the imperative model, and it can be translated into any imperative language and back. The book closes by naming what Halo is: a language for drawing programs that has the axiomatic form. A language has that form when every element it has is required for completeness of representation, when nothing else is in it, and when the same elements cover every layer of compute. In Halo, the diagram can become the program.

I. The Question

Every programmer has seen two pictures of the same system. One is the code: modules, the functions inside them and the calls between them. The other is the database: tables, the records in them and the keys that link one table to another. The two pictures are usually drawn by different people, taught in different courses and judged by different rules.

This book asks what those two pictures have in common, and answers that they are one picture. A module is a table. An instance of the module is a record in that table. A function of the module is a column. A reference from one module to another is a foreign key, the field of one record that names a record of another table. If that is so, then the discipline that keeps a database clean keeps a program clean as well, and the question of how to organize a large system has an answer as definite as the question of how to organize its data.

The book then asks the practical question that follows. If code and data share one structure, can a program be written in that structure directly, so that the structure is never lost and never has to be recovered? Halo is the answer to that second question.

II. The Ground

The ground of the whole book is one result, stated in Chapter 1:

> All software architecture can be represented in third normal form, and there exists a strict one-to-one correspondence between the notion of a software module and the notion of a database table.

Third normal form is the rule, from relational database design, that each fact is stored in exactly one place and depends on the key that names it, the whole key, and nothing but the key. A table is divided where the

world it describes divides. Chapter 1 shows that the same rule governs code, because every function can be viewed as a state machine and every state machine as a table.

Halo is built on that result, and so the book gives the normal form first and Halo second. A reader who has the normal form will see, in each convention of Halo, a rule of the normal form applied to a drawing, where a reader who meets Halo first would see only a set of conventions.

The parts fit as ground and building. The normal form is the ground: one structure for code, data and architecture. Halo is the building raised on it: a way of writing in that structure from the first circle drawn, so that the structure is never lost and never has to be recovered.

III. A Few Terms From the Immanent Metaphysics

A few terms in this book come from the Immanent Metaphysics, the account of comparison, choice and process that I set out in my book *An Immanent Metaphysics* (2001). They are given here once, in plain words, so that a reader who has not met them can follow every chapter that uses them. The chapters use them in their method notes and in the sections marked for readers who want the ground of the engineering in the Immanent Metaphysics. A programmer who wants only the engineering can go on to Section IV now, and come back here when one of those sections points back.

Suppose two people are playing chess. One move is being made now, on this board, by this player, and what that move does depends on everything around it. The rules, and the written record of the moves already played, read the same to anyone, anywhere. The moves still open have been made by no one yet, and any of them might become the next.

The first are the three *modalities*: three roles that anything under consideration can play. The *immanent* is process: interaction, what happens here and now, and what depends on its context. The *omniscient* is structure: what holds the same regardless of position, what proceeds invari-

antly. The *transcendent* is possibility: what might become, and what is not yet bound to any one instance. In the game of chess, the move being made is immanent, the rules and the record are omniscient and the moves still open are transcendent. A *modal assignment* says which of the three roles a thing plays within a given frame of consideration. The same thing can play a different role when it is considered within a different frame, or by an observer oriented differently.

The second are the three *axioms*, which state how the modalities relate. In the words of *An Immanent Metaphysics* (aphorism 1.3-1):

> Axiom I. The immanent is more fundamental than the omniscient and/or the transcendent. The omniscient and the transcendent are in conjugate relationship to one another.
>
> Axiom II. A class of the transcendent will precede an instance of the immanent. A class of the immanent will precede an instance of the omniscient. A class of the omniscient will precede an instance of the transcendent.
>
> Axiom III. The (classes/instances) of the immanent, omniscient, and transcendent are distinct, inseparable, and non-interchangeable.

In plainer words: process is more basic than structure or possibility, and structure and possibility are a *conjugate* pair: two things of which each can be perfected only at the expense of the other. Rules that fixed every move would leave no move open, and a game with every move open would have no rules. The three follow one another in a fixed cycle, from possibility to process to structure, and from structure to possibility again. Of the moves still open, one is played; the move played goes into the record; and the record sets which moves are open next. And no one of the three can be reduced to the others, occur without the others, or be substituted for another. A game has all three, and no one of them can do the work of another.

Think of one line of text in a file. Copy the line into a second file: the same text, in a different place. Read the line twice where it stands: the same text, in the same place. Set two different lines in two files: different text, in different places. Change the line where it stands: different text, in the same place.

The third is the *Incommensuration Theorem*. *Symmetry* is a sameness of content across a difference of context; *continuity* is a sameness of content across a sameness of context; *asymmetry* and *discontinuity* are their opposites, a difference of content across a difference of context and a difference of content across a sameness of context. Those four are the four cases of the line of text, in that order. The theorem states that symmetry and continuity cannot both be applied, simultaneously and fundamentally, to any event or comparison; and neither can asymmetry and discontinuity. In the words of *An Immanent Metaphysics* (aphorism 1.43-2): "Nothing anywhere at all can be both absolutely symmetric and absolutely continuous". "Nothing anywhere at all can be both absolutely asymmetric and absolutely discontinuous". The Incommensuration Theorem is a structural fact about the nature of knowing: it holds in every comparison, and so in every act of knowing. The theorem stands here because its definitions are made of content and context, the two words that the next set of terms names and that Chapter 1, Section VII, puts to work.

The fourth are three words that name the aspects of any interaction: *content*, *context* and *choice*. The definitions of symmetry and continuity above already use the first two. Content is what is given in an interaction: what is compared, carried or acted on. Context is the setting in which the content stands, which gives the content its relations to everything else. Choice is the act of the one who takes part in the interaction: the one who compares, selects and acts. The three are distinct, inseparable and non-interchangeable, in the sense of Axiom III, and so in any one frame of consideration each of them plays one of the three modal roles; which role each plays is set by the frame, as every modal assignment is. Chapter 1,

Section VII, uses the three words in this sense: when code runs, the hardware is its content, the model is its context, and the user is the one who makes the choice.

IV. The Path Through the Book

The book has two parts and a closing chapter.

Part I, The Normal Form, has four chapters. Chapter 1, The Equivalence Thesis, shows that code is organized at three levels (the single operation, the module and the system) and that the same structure governs all three, so that all software architecture can be represented in third normal form. Chapter 2, The Factoring of Intentionality, shows what *normalization* (arranging data, or code, in a normal form such as the third) accomplishes: it is a compression, and compression at the level of architecture is the factoring of intentionality. It follows that a community can verify what a system does only when the system's factoring can be inspected. Chapter 3, Third Normal Form as Metaphysics, shows that a model of computation which means to be general must be able to describe its own structure in a form a person can read, and that code organized in third normal form can do so. Chapter 4, Classes and Instances, carries the pair of terms on which the whole mapping rests, class and instance, beyond software, and grounds it in the uniquity of every instance (no two instances are ever perfectly identical). *Software, Wildness, and the Normal Form; From Software Architecture to Life, Justice, and Wisdom* carries these four chapters, in its own wording, as its Part I, where they open an argument that runs on to life, justice and wisdom. A reader who wants to follow the argument that far can go on to that book after finishing Part I here.

Part II, Halo, has eight chapters. Chapters 5 and 6 teach the language itself: its one symbol, how a trigger (the pulse that carries control along a line) pulls and pushes the values on a circle's pins (the numbered points where lines join the circle; to pull is to read every input at one moment, and to push is to send out every output), and the few limits a program-

mer accepts so that a drawing reads the way people think. They teach it with the language's own drawings, built up one step at a time, from an empty circle to a complete program. Chapter 7 says what counts as code: whatever cannot be drawn in this form is not code. Chapter 8 joins the two parts of the book: code drawn in Halo and organized in third normal form is fully general to all system design, and it is the form in which a system can be changed without debt. Chapters 9, 10 and 11 carry Halo into practice: modules as tables, the call triple (the three things a call names: which table, which record and which method in that record) that lets a running system be changed while it runs, the refactoring tools on which a code base's long life depends, and the exact way Halo is written in JavaScript. Chapter 10 also gives the reason the language is as small as it is: software that must last for a thousand years, and run on hardware far weaker than ours or not yet invented, can depend on almost nothing. Chapter 11 closes with the RISL standard (Reduced Interaction Structuring of Language: the standard by which a Halo drawing is written as code in each language) for JavaScript: what a reader of a Halo program needs from it, and a module written in its canonical form; the standard itself, rule by rule, is in *The RISL Standard; Halo Written as Code, Across Languages*, Chapter 5. Chapter 12 turns to trust: how a person can be sure that the code a computer runs does what its author intended.

Chapter 13, Drawing the Program, names what Halo is. I have long wanted a diagram that is the program and not a picture of it. A language for drawing programs has the axiomatic form when every element it has is required for completeness of representation, when nothing else is in it, and when the same elements cover every layer of compute. The chapter derives those elements from the one question a program answers, shows what is lost without each, and shows that any language with that form is Halo drawn another way. It also shows how the simplest teaching picture of a drawn program (circles for operations, diamonds for decisions, one kind of line for control and another for data) maps onto Halo's own notation.

Each chapter opens with an abstract and closes with a Summary of what it establishes and a list of its key concepts. The book closes with a Summary of its own. Two appendices follow it: Appendix A gathers the rules of a Halo drawing in one place, each with the chapter that teaches it, and Appendix B carries one program, hello world, from its drawing to its complete text in JavaScript written to the RISL standard. Then come the Terms, which gather the definitions of the key concepts with the chapter in which each is defined. After the Terms come the Bibliography, which gives every published work the chapters name, and the Index of terms and names.

Part II and Appendix A together are the definition of Halo. The chapters teach each rule with the drawing it governs and give its reason; Appendix A states every rule in one place, in the form to which an editor, a translator or a checker is built. A person building such a tool needs one text to build to, and so, where the two are worded differently, Appendix A is the rule. Halo defines how a program is drawn. The file encoding of drawings, how a drawing is stored as a file, is a separate standard, a colleague to Halo; how a drawing is written as code in each language is *The RISL Standard; Halo Written as Code, Across Languages*. A reader of this book meets the RISL standard in Chapters 9 to 11, and needs the file encoding only to build a tool that stores drawings.

V. Who the Book Is For

This book is written for programmers. It assumes that its reader has written code, has met a function, a loop and a database table, and wants to know why some systems stay understandable for decades while others decay within a year.

Chapter 1 builds every term it uses from the ground up: the three elements of computation, the function, the state machine, the table and the key. Chapters 9 to 11 show code in JavaScript, Java and C for readers who program in them, and each of the three states its results in its Summary in words that need no code.

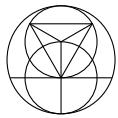
The question the book opens with is the one a programmer meets first: if a module and a table are the same structure, what is that structure, and does it reach all the way to the architecture of a whole system? Chapter 1 answers it.

PART I. THE NORMAL FORM

From Software Architecture to Metaphysical Principle

The four chapters in this part establish the foundation on which the rest of the book builds. The foundation is technical: it concerns how software is structured, what normalization means and why a principle of database design turns out to govern whether a community can verify that the systems it depends on do what it chose.

The argument begins with the equivalence thesis, the claim that object-oriented architecture and relational architecture are not two approaches among many but two notations for the same thing. It moves through the factoring of intentionality, where normalization is revealed as a compression principle and inspectability as a political question, to the broader claim that third normal form is a metaphysical principle, appearing wherever self-reference and the structure of computation are at issue. It closes with classes and instances: the relation of a pattern to the things made from it, which holds well beyond software, and the uniquity of every instance.



Chapter 1

The Equivalence Thesis

Code and data are taught as two disciplines; do they have one structure, and does that structure reach all the way to the architecture of a whole system?

ABSTRACT

This chapter develops the claim that all software architecture can be represented in third normal form, and that there exists a strict one-to-one correspondence between the notion of a software module and the notion of a database table. This *equivalence thesis* unifies two disciplines, software architecture and data architecture, that have traditionally been regarded as distinct. The argument proceeds from the universality results of computation (the results on what is enough to compute anything that can be computed: the Church-Turing thesis, and the structured program theorem of Böhm and Jacopini) through the three necessary and sufficient elements of computation (sequentiality, conditionality and rings), to the mesoscopic structure (the middle level, of named modules) of functions and state machines, and finally to the macroscopic level (the level of whole systems) where system architecture and relational data organization are shown to be structurally identical. The metaphysical significance of this result is that the same tripartite pattern (a pattern of three parts) that governs computation at the microscopic level (the level of single operations) recurs at every scale of software organization, providing a concrete instance of the modal dynamics (the way the three roles of the Introduction, Section III, follow one another) described in the Immanent Metaphysics.

I. Three Levels of Software Organization

Computer science works at three levels of organization, though this fact is rarely made explicit and almost never taught as a unifying principle.

At the *microscopic* level, computation is constituted by three elements: sequentiality, conditionality and rings. Sequentiality is the ordering of steps in time, the causal chain of first-this-then-that. Conditionality is the branching of the causal chain based on the state of data: if this condition

holds, follow this path; otherwise, follow that one. Rings (or loops) are the capacity to return to a prior point in the sequence. Joined with a condition, a ring repeats a block of operations until the condition is met.

The Church-Turing thesis, reached independently by Alonzo Church and Alan Turing in 1936, asserts that every function that can be calculated by an effective procedure (a finite list of exact steps that can be followed mechanically) can be computed by a Turing machine (the textbook model of computation: a tape of cells, a head that reads and writes one cell at a time and a finite table of states). A language is *Turing complete* when it can simulate a Turing machine, and so any algorithm represented in one such language can be faithfully translated into any other. The Church-Turing thesis, with the faithful translation between Turing-complete languages that follows from the thesis, is the fundamental universality result of computation. The result that these three elements suffice for every program is the structured program theorem of Corrado Böhm and Giuseppe Jacopini (1966): every computable function can be computed by a program that joins its steps only by sequence, by selection on a condition and by repetition until a condition holds. Together the two results say that any language implementing these three elements, acting on a store of data that can grow without limit, can represent any algorithm. A machine whose memory is fixed and finite, such as a thermostat with a loop, has all three elements and is still not Turing complete. Because a machine with a fixed and finite memory is not Turing complete, the store that can grow is named beside the three elements: the three elements say how the steps are joined, and the store says how much the steps can hold. The universality of the triple means that, at the microscopic level, the choice of programming language is incidental. The deep structure is the triple: sequence, condition, ring.

At the *mesoscopic* level, computation is organized into modules: functions, procedures, objects, classes. Modules are the units of intentionality. Each module represents a discrete chunk of what the programmer means

to do. The mesoscopic is the level at which software is readable, maintainable and changeable. It is the level at which human cognition actually engages with code.

At the *macroscopic* level, computation is organized into systems and architectures: the relationships among modules, the structure of dependencies, the patterns of data flow across an entire application or service. The macroscopic level is the level of system design.

The argument of this chapter is that the same structural principle governs all three levels, and that this principle can be made precise. The next three sections make it precise one level at a time: the triple at the bottom; in the middle, the triple written out as a table; and at the top, those tables kept by the rule of third normal form.

II. The Microscopic: Computation as Triple

The three elements of computation are *necessary and sufficient*: the three together are enough for general computation, and general computation needs every one of them. That makes the three a fact about computation, more than a convenient classification. Remove any one, and the capacity for general computation is lost. A system with only sequentiality and conditionality cannot repeat operations. A system with only sequentiality and rings cannot branch. A system with only conditionality and rings has no causal chain to branch or repeat.

Every practical programming language implements some version of these three elements, though the surface syntax varies enormously. JavaScript has `if` statements and `for` loops. Haskell has pattern matching and recursion. Assembly has conditional jumps and labels. Beneath the syntactic differences, the triple is invariant: it is the same in every language.

> **Method note.** This note is for readers who want the ground of the triple in the Immanent Metaphysics; the engineering argument is complete without it. The claim that sequentiality, conditionality and rings constitute a necessary and sufficient triple is stronger than the usual statement of the Church-Turing thesis, which speaks of Turing machines and lambda calculus without decomposing them into a triple of named elements. Sufficiency of the triple is the structured program theorem; what this book adds is that each element is necessary (Section II) and that the three correspond to the three modalities of the Immanent Metaphysics. The correspondence is: sequentiality to the omniscient (that which proceeds invariantly), conditionality to the immanent (that which depends on context), and the ring to the transcendent (that which relates beginning to end, the whole to its parts). As with all modal assignments, the correspondence can rotate under axiom two (the fixed cycle in which possibility, process, and structure follow one another; Introduction, Section III) depending on the orientation of the observer.

III. The Mesoscopic: Functions, State Machines, and Modules

At the mesoscopic level, the programmer works not with individual steps but with *modules*: named, bounded units of functionality that can be defined, invoked, composed and reused. In most languages, the basic unit is the function. A function takes inputs, performs a sequence of operations (which may include conditions and loops), and produces outputs.

Take a small function that reads a string one character at a time and reports whether the whole string is a number. At any moment it stands in one of three situations: it has read nothing yet, it has read only digits so far, or it has met something that is not a digit. Each new character moves it from one situation to another, and that is all the function has to remember.

A less commonly appreciated fact is that every function can be equivalently represented as a *state machine*. A state machine is a system that occupies one of a set of states at any given moment, and transitions between states based on input and rules. The three situations of the small function are its states. A programmer who has written a sort or a parser will ask how many states such a function has, and whether a table can hold them all. On any machine with finite memory the set is finite; in any case a table may hold as many rows as the function has states. The table is the structure of the function, and no one has to write its rows out by hand. The equivalence between functions and state machines is well established in theoretical computer science, but its practical significance is not widely taught.

The significance is this: because every function can be viewed as a state machine, and because every state machine can be decomposed into a table of states, inputs and transitions, every function can be viewed as a structured data table. A table is a set of records, one to a row, with one column for each kind of fact a record holds. The function's inputs are the table's columns. The function's internal states are the table's rows. The function's transitions are the relationships between records.

Written as a table, the small function that reads a string has three rows and two columns. Its rows are its three states: nothing read yet, only digits so far, and not a number. Its columns are its two inputs: a digit arrives, and anything else arrives. Each cell names the row to go to next. In the row for nothing read yet, a digit leads to the row for only digits, and anything else leads to the row for not a number. In the row for only digits, a digit leads back to the same row, and anything else leads to the row for not a number. In the row for not a number, both columns lead back to the same row. When the string ends, the row the function stands in is its answer.

The triple of Section II is all in that table. Going from one row to the next is sequentiality. Reading across to the column the input selects is conditionality. A cell that leads back to a row already visited is a ring. So the table is the triple written out for one function, and that is what joins the first level of organization to the second.

This observation is the hinge of the entire argument. If every function maps to a table, then the discipline of how we organize functions (software architecture) and the discipline of how we organize tables (data architecture) are not two disciplines. They are the same discipline, viewed from two orientations.

IV. The Macroscopic: Architecture as Third Normal Form

Suppose a library keeps one table of its loans, with one record for each book that is out, and with columns for the loan number, the member's number and the member's telephone number. A member with twelve books out has the same telephone number in twelve records. When that number changes, a clerk corrects eleven of the twelve. The table now gives two telephone numbers for one member, and nothing in the table can say which is right.

In database design, *third normal form* is the standard of well-organized data. The *key* of a table is the field, or set of fields, whose value names exactly one record. A database in third normal form has tables in which every non-key column depends on the whole key and nothing but the key. Redundancy is eliminated. Each fact is stored in exactly one place. Relationships between facts are represented by foreign keys linking one table to another.

The library's table breaks that rule. Its key is the loan number, and the telephone number depends on the member and not on the loan. Third normal form divides the table in two. A table of members, whose key is the member's number, holds each telephone number once. The table of

loans keeps the member's number as a foreign key and holds no telephone number at all. The next time a number changes, one record changes, and there is no second copy to disagree with it.

Third normal form is taken as a gold standard because it produces data structures that are efficient to store, efficient to access and faithful to the actual relationships among the things they represent. A well-normalized database encodes something about the *nature* of the domain it models. The partitioning of data into tables mirrors the partitioning of the world into entities and relations.

The equivalence thesis states: *all software architecture can be represented in third normal form*.

The claim is that there exists a one-to-one mapping between any software architecture and a relational data architecture in third normal form, and that this mapping is exhaustive: no software architecture escapes it. The equivalence thesis is a claim about the structure that software architecture has, and the thesis stands apart from any stylistic preference that software architecture *ought to* be represented in third normal form.

The step from tables to third normal form is secured by a theorem of database design: every collection of tables, with the dependencies among its fields, can be decomposed into third normal form without losing any fact and without losing any dependency (Bernstein 1976). When the table is a module, the dependencies among its fields are the same kind of fact as they are in a database: knowing the value of one field fixes the value of another. Every architecture has a relational form, by the construction that follows, and every relational form has a third normal form, by that theorem.

V. The Mapping in Detail

The mapping proceeds as follows.

Every module, whether conceived as a function, an object, a class or a service, corresponds to a table. The module's internal state corresponds to the table's columns. Each instance of the module at runtime corresponds to a record in the table.

Every reference from one module to another (a function call, a pointer, a dependency injection, a foreign key in object-relational parlance) corresponds to a foreign key linking one table to another. Object creation is record creation. Object deallocation is record deletion. A method call is a query against a table that returns a computed result.

Suppose a module named `Account` has two fields, an owner and a balance, and one method, `is_overdrawn`. As a table it has three columns: owner, balance and is_overdrawn. Suppose two instances of it are running, one with a balance of 40 and one with a balance 15 below zero. They are two records. The owner field is a reference to a record of another module, and so the owner column holds a foreign key. In the first record the cell under is_overdrawn reads no, and in the second it reads yes. Asking an instance whether it is overdrawn is reading that cell, and the cell is computed from the balance at the moment it is read, so the balance stays the one place that fact lives.

Even a naked function that belongs to no class can be elevated into its own singleton table: a table with one record, containing the function's closure (the function together with the variables it keeps from the place where it was defined) and its reference to any external state. Languages like Java enforce this by requiring that every function live inside a class. But even in languages that do not enforce it, the mapping holds. A free-standing function is simply a table with a single row.

These three placements of a function are one mapping, seen at three grains. A method belongs to the module of its class, and so it is a column of that class's table: an attribute that each record of the class carries. A function considered as a state machine is a table of its own: its inputs are the columns, its states are the rows and its transitions are the relationships between records. A free-standing function, which belongs to no class, is a

table with a single row. Each placement answers a different question about the function (what it belongs to, what it is made of, where it lives), and the three agree, because each records every fact about the function in exactly one place.

The word state is used at two grains in these pages, and the two fit together. A column of a module's table names one part of what a state is made of: the balance of an account is one such part. A record of that table holds one instance's current setting of all those parts. A row of a function's table is one complete state: one whole setting of all the parts at one moment. So the columns say what a state is made of, and the rows say which states there are.

> **Sidebar: Object-relational mapping as common practice.** The software industry has long used *object-relational mapping* (ORM) as a practical technique for bridging code and databases. What is being claimed here goes beyond ORM. ORM is a translation layer, a pragmatic tool for connecting two things that are assumed to be fundamentally different. The equivalence thesis says that code and databases are not fundamentally different. They are the same structure, and the need for a translation layer is an artifact of not recognizing the identity.

The mapping holds for concurrent and streaming systems as well. A channel along which messages pass is a table. Each message or event sent along the channel is a record in that table. Each subscription, by which one process receives what another process sends, is a foreign key from the receiving module to the channel. The order in which events arrive is a column of each event's record, and the state of each process at any moment is what its records, read in that order, determine. A concurrent or streaming system adds no new kind of thing to the mapping: its timing is one more column.

The mapping holds for inheritance in the same way. A class that inherits from another class is a table of its own, and the key of each of its records is also a foreign key into the table of the parent class. Each record of

the child names the record of the parent that holds the facts the two classes share, and the child's own table holds only the facts the child adds. Every inherited fact is therefore still stored in exactly one place, which is what third normal form asks. Inheritance, like timing, adds no new kind of thing to the mapping: it is a pair of tables joined by a key.

VI. Why This Matters: The Unification of Architecture

If the equivalence thesis holds, then everything that has been learned about data organization in database theory applies directly to software architecture, and everything learned about software architecture applies to data organization.

The principles of normalization, which have been rigorously developed over decades for the management of data, become principles of code organization. The notions of functional dependency (the value of one field fixing the value of another), of eliminating redundancy, of ensuring that each fact lives in exactly one place, become architectural principles for the design of software systems.

Suppose a program holds the rate of sales tax as a constant in its checkout module, and holds it again in its invoice module. The sales tax rate held in two modules is the library's telephone number over again: one fact in two places. The rate changes, a programmer corrects the checkout module, and the program goes on running while two of its parts disagree. Factored, the rate is one field of one record in one module, and the checkout and the invoice each hold a reference to it. The next change is made once. Every change made to the first program is a chance for its copies to drift apart, and every change made to the second has one place to go. Repeated over the years of a system's life, that difference is part of why some systems stay understandable for decades while others decay within a year.

Because the principles of normalization apply to code, system architecture is not the ad hoc, intuition-driven discipline that system architecture is often treated as in industry. It has a rigorous foundation as solid as the

Church-Turing thesis at the microscopic level, and as principled as third normal form at the data level. There is a *right way* to organize any software system, in the same sense that there is a right way to normalize a database. Where the library's telephone number goes is no matter of taste: it depends on the member, and so it goes with the member. Where the tax rate goes is settled the same way.

To the best of my knowledge, this claim is not common knowledge in the software engineering community. If one were to survey the top engineers at the major cloud service providers and ask whether they believe that software architecture has the same fundamental rigor as the Church-Turing thesis or third normal form, most would likely say no. Architecture, they would say, is a matter of judgment, of trade-offs, of experience. The equivalence thesis says otherwise.

VII. The Connection to Process

This section is for readers who want the ground of the result in the Immanent Metaphysics; the engineering argument is complete without it.

There is a deeper significance to the structure revealed by this analysis.

The word *process* is used in computer science to mean a running instance of a program: a thing that occupies resources, executes instructions and persists for some duration. The same word is used in the Immanent Metaphysics to mean the dynamic of change itself: the way in which content moves through context, the way in which choice and causation interact across time.

These two uses are structurally the same: the one word names one structure.

When code runs, three domains are simultaneously in play. There is the hardware: the flow of electrons, the physical substrate in which the computation actually takes place. There is the model: the abstract repres-

entation of some aspect of the world that the code embodies. And there is the user: the person whose intentions the code serves, the one who is affected by its outputs.

These three domains (hardware, model, user) constitute a tripartite interface that is structurally identical to the three-domain structure described by the metaphysics. The hardware is the objective substrate (content). The model is the relational structure (context). The user is the subjective agent (choice).

And the process that moves through these three domains (the execution of code, the change of state, the flow from input to output) is the same dynamic, the same *axiom two*, that governs process at every other scale in the metaphysical system.

> **Method note.** The connection between computation and metaphysics is structural identity, and not analogy. Computation is a concrete instance of the abstract dynamics described by the metaphysics. That computation is a concrete instance of the dynamics the metaphysics describes is one of the reasons why understanding computer science can deepen one's understanding of the Immanent Metaphysics itself. The flow of code execution makes visible the dynamics of process in a medium where they can be directly observed, tested and modified.

VIII. Summary

Software engineering is often treated as a discipline without foundations, a craft governed by judgment and fashion rather than by principles. The microscopic level has its foundation in the universality results of computation: the Church-Turing thesis and the structured program theorem. The mesoscopic level has its foundation in the equivalence of functions and state machines. But the macroscopic level, system architecture, has been regarded as the province of intuition.

The equivalence thesis closes this gap. It asserts that system architecture has a foundation as rigorous as the other two levels: the principles of third normal form, applied not only to data but to code.

The chapter's question now has its answer. Code and data have one structure, the normal form, and that structure reaches the architecture of a whole system, because every module is a table and every reference between modules is a foreign key. Because code, data and the architecture of a whole system have one structure, I can go on to the second question of this book. If the whole of a system has one structure, then there is one structure for a language to write directly, and I give that language in Part II.

A reader now holds three established results (the universality results of computation, the equivalence of functions and state machines and the principles of relational normalization) and one bridging claim: that the state-machine representation of a function maps one-to-one onto a relational table. A function's table is a relational table, joined to the other tables of the system, when and only when each place where the function reaches another module is a key, and so the bridging claim holds exactly when every reference between modules is treated as a foreign key. That gives every architecture a relational form, and Bernstein's theorem gives every relational form a third normal form that loses no fact and no dependency (Bernstein 1976), so the thesis follows in two steps. The mapping is exhaustive: it holds without exception. Its scope is the structural equivalence of software architecture and data architecture. Performance can justify a deliberate departure from third normal form in a particular implementation. In database design, denormalization is such a departure: denormalization stores a fact in more than one place on purpose, for speed. Such a departure is understood as a departure from a known ideal, and the ideal is still the measure of what the speed has cost.

The deeper implication is that the tripartite structure visible at the microscopic level of computation (sequentiality, conditionality, rings) is not a property of the microscopic alone. It recurs at every scale. Functions are triples of input, process and output. Systems are triples of hardware,

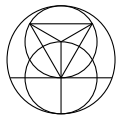
model and user. Architecture is a triple of data structure, algorithmic process and the intentionality that connects them. The small function of Section III shows the first of these: the string is its input, the walk through its table is its process and the yes or no is its output. Section VII showed the second. The third triple, architecture, is the mapping of Section V seen whole. In that mapping the tables are the data structure, and the functions that run over the tables are the algorithmic process. What the programmer means the system to do is the intentionality that divides the system into these tables and no others.

The pattern does not change. Only the scale changes. And this is precisely what the metaphysics predicts: three roles that anything under consideration can play (Introduction, Section III), and so three roles at every scale.

If code and data are one structure, what does normalizing that structure actually accomplish, and for whom?

KEY CONCEPTS IN THIS CHAPTER

- **Church-Turing thesis**: The foundational result of Church and Turing (1936) asserting that every function that can be calculated by an effective procedure can be computed by a Turing machine.
- **Turing complete**: Able to simulate a Turing machine; an algorithm represented in one Turing-complete language can be faithfully translated into any other.
- **Structured program theorem**: The result of Böhm and Jacopini (1966) that sequence, selection on a condition and repetition until a condition holds suffice to compute every computable function; with the Church-Turing thesis, it means that any language implementing sequentiality, conditionality and rings (or equivalent constructs) can represent any computable algorithm.
- **Sequentiality, conditionality, rings**: The three necessary and sufficient elements of computation. Sequentiality is ordered steps. Conditionality is branching based on state. Rings are returns to a prior point in the sequence; joined with a condition, a ring repeats a block of operations until the condition is met.
- **Third normal form**: The standard of well-organized relational data, in which every non-key attribute depends on the whole key and nothing but the key, eliminating redundancy.
- **Equivalence thesis**: The claim that all software architecture can be represented in third normal form, via a one-to-one mapping between modules and tables, and between inter-module references and foreign keys.
- **Microscopic, mesoscopic, macroscopic**: The three levels of software organization: the level of individual operations, the level of named modules and the level of system architecture.
- **Process**: Used both in computer science (a running program) and in the Immanent Metaphysics (the dynamic of change across content, context and choice). The two uses are structurally identical, and not only analogous.



Chapter 2

The Factoring of Intentionality

Where does third normal form fit into software, what does normalizing a system accomplish, and for whom?

ABSTRACT

Where does third normal form fit into software? The answer turns out to be much larger than the question. This chapter develops the exact correspondence between object-oriented software architecture and relational database architecture (a class is a table, an instance is a record, a function is a column, a pointer is a foreign key) and shows that normalization is therefore a fact about architecture generally, not about storage. It then shows what normalization actually accomplishes: a reduction in the contact space (the middle bank of connections through which each entry reaches the others) between past and future, from about the number of entries multiplied by itself to about the number of entries multiplied by its logarithm, achieved by cleaving the representation at the same points at which our understanding of the world cleaves. From this the chapter draws the consequence that software engineering is the practice of factoring intentionality, and that closed source is therefore concealed intent by construction. It sets out the regress of trust (hardware, compiler, debugger and the trusting-trust argument beneath them all) and argues that clarity, not simplicity, is what transcends complexity, because complexity will always win an arms race with simplicity and cannot win one against right application of principle. It draws the political consequence: that a community can only verify that what runs is what it chose if the architecture is inspectable, and that imposing an uninspectable system on a community is the same act as cultural colonialism. It closes by reading an existing formula for trust as three connections (to self, to other and to world), and by holding the relation of humanity and machine among three poles, with nature as the third.

I. The Identity of Software and Database Architecture

Chapter 1 gave the mapping from the side of functions; here it is given from the object-oriented side, with the one amendment that makes it exact. The amendment says which value stands in the cell of a function. Be-

gin with the object-oriented model, because it is the one most people carry in their heads. An object is a container that has some data associated with it and some functions that operate on that data.

Now consider a class of such objects. Each instance is, effectively, a record in a table. The class itself is the table. The private data of the object are the fields of the record. And each function is also a field: the application programming interface of the module is a column vector in the table. I cannot see the contents of the private data; they are there, but they are not exposed. What I can see is the output of each function, which is a value in a single cell. But a function can be called with many different inputs: which input is that one value the output for?

One amendment is needed to make this exact. We are calling in a multivariate convention: a function may take several inputs and produce several outputs. Most language notations permit only a single return, which may be a pointer to some object. The idea is that in every case the inputs and outputs of a function are either simple data types or pointers to complex data types which are themselves, through some process of reification (the making of a relationship or a process into a thing that can be stored and passed), eventually reducible to a collection of simple data types. So however complicated they are, the inputs and outputs of a function call can be regularized into fields within the same record. A record then holds a function's inputs beside its outputs, and the value in the function's cell is the output for the inputs in that same record. The whole module can then be treated as a spreadsheet with the same formulas in each column of cells.

And this is where third normal form appears. Suppose an input to a function is a pointer to another object. I could flatten that object out into its simple types. Or I could treat the pointer as a foreign key into another table whose records are objects of that kind. Take the second route and the correspondence completes. Everything that is an object is a record, or an exemplar of a record, in a table. Everywhere there is a pointer (a call by

reference, a return by reference) there is a foreign key linking one table to another. Object creation is record creation. Object deallocation is record deletion.

Nor is there an escape at the boundary. Where there is a naked function outside any class, that function can be elevated into its own class. Even if the class has no data and only that one function, and even if it is a singleton with exactly one instance in the entire system, it can still be treated as a module in this structured form. We see the same thing in actual object-relational databases, where a table may hold a single record containing counters used to generate keys for other tables. It is not ideal practice. It is nonetheless common practice, and it demonstrates that the mapping holds even in degenerate cases.

So: whether we construct modules in the sense of functions or modules in the sense of objects, or anything between, the structure maps one-to-one onto tables with linking relationships, which is what we ordinarily call the object-relational model. The notion of software architecture is equivalent to the notion of database architecture. The equivalence is exact, and there are no exceptions. Literally everything maps.

II. What Normalization Is Actually For

Having established the identity, we can ask the more interesting question. What have we learned about data organization that explains why third normal form is worth wanting?

Part of the answer is storage efficiency, and part is access efficiency, and sometimes access control. But most of the time the answer concerns something else: what do we consider the points of articulation (the natural joints at which a representation should be divided, as an arm divides at the elbow) to be?

In a database, the question is how we hold the relationships among the data elements in a way that encodes something about the nature of the physical system being represented. Take a logistics operation moving

goods from one customer to another. There is an address at which billing is received. There is an address to which the product will be shipped. There is an address from which it came. The notion of address appears three times in different roles. So we factor address out into its own table and establish foreign keys to it from the shipping manifest, which in turn points to a product description.

The gain is more than tidiness. Partitioning in this way structures the data in the same way in which we think about it, and by doing so it inherits a set of properties.

The most important of them concerns connectivity. Suppose at some initial time I have a mass of disorganized data, and at some later time I will need to access some of it, but I do not yet know which. In the worst case every future access may require any past element, so the connectivity I must represent is the number of elements multiplied by itself. Picture two columns, one for each time, with every entry on the left connected to every entry on the right. I call that a K matrix (what graph theory calls a complete bipartite graph).

No one builds switching systems that way. The old analog telephone exchanges would have required an impossible bundle of wire to connect every telephone to every other telephone. What was built instead was a bank of switches in the middle: everything on the left connects to that middle bank, and the middle bank connects to everything on the right. The engineering question becomes: how small can that middle bank be while still permitting everything to reach everything? Telephone engineering answered it. A full crossbar, with a switch at every crossing of a line on the left and a line on the right, needs as many switches as the number of lines multiplied by itself. A switching network built in several stages (Charles Clos described the three-stage form in 1953, and V. E. Beneš the recursive form in 1964) connects every line to every other line with about as many switches as the number of lines multiplied by its logarithm.

Normalization does exactly this to the data. Dividing at natural points of articulation narrows the contact space (the middle bank) between input and output. If the same address record appears in many shipping records I would otherwise pay the full multiplication; with a linking relationship I need not fill in every entry in the middle column. Every connection runs through the middle bank, and so what narrows the bank reduces the count of connections. The number of connections can be reduced from about the number of entries multiplied by itself to about the number of entries multiplied by its logarithm. The reduction to about the number of entries multiplied by the logarithm of that number is the best case, and natural articulation tends to give that best case.

A worked count shows where the reduction comes from. Take a thousand orders and the addresses they use. Unfactored, any order may share an address with any other, and to know which do, every order must be compared with every other: about a thousand times a thousand connections, a million. Factored, each order holds one key into the address table, a thousand connections in all, and the address table is the middle bank through which each order reaches every other order that shares its address. A thousand orders in one table of addresses is one level. Suppose the addresses are themselves gathered by city and the cities by region, and suppose each record of a table gathers ten records of the level below it. Three levels of such tables reach a thousand entries, four levels reach ten thousand, and each further level multiplies the reach by ten. A whole system factored this way is a hierarchy of such tables, each serving as the middle bank for the level below it, and the number of levels in such a hierarchy grows with the logarithm of the number of entries. The logarithm comes from the levels of the hierarchy: each entry carries one key for each level, so the count of connections is about the number of entries multiplied by the number of levels. The thousand connections counted above are only the one level from orders to addresses. Each further level brings its own keys, so the count grows from one key for each entry to one key

for each entry at each level. For the thousand orders with three levels, that is about three thousand connections, where the unfactored count was a million.

There is another way to say this which makes the general principle visible. Third normal form is a compression. Compression of images, of audio, of files on a disk all work the same way: you cannot compress infinitely, but you can factor out the redundancies. Third normal form factors out the redundancies. In each case some measure of description length is made smaller by finding and removing what repeats. And that is the same compression we want to perform at the level of architecture.

Which brings the argument to the point it has been traveling toward. What is software engineering actually doing? It is answering the question: how do we hold in mind the tremendous complexity of the system? If the system were disorganized and unfactored we could not hold it at all. The compression that wants to happen to remove common strings from a collection of files, and the compression that removes common sub-aspects of records into their own tables, and the compression that removes common conceptual ideas about the structure of systems and the structure of intentionality: these are the same compression.

Suppose a shop means this: a customer who is over the credit limit gets no more goods on credit. In an unfactored system that intention is scattered. A check is copied into the checkout screen, into the telephone order form and into the monthly billing run, each written a little differently, and no one place in the code says what the shop means. Factored, there is one module named for credit, holding one function that says whether this customer may take this much credit, and the other three parts call it. The shop's intention now stands in one place, under its own name, and there is one place to go to read it or to change it.

So: what we are doing in software engineering is learning how to factor intentionality, as represented by code, in a way that cleaves in our minds the same way that our understanding of the world cleaves. That statement about factoring intentionality is a general statement. It is not

particularly about databases. It is close to being a statement about how the world works. Where the factoring is done badly, or not at all, the system may still run, but it becomes progressively less possible for anyone, its own authors included, to say with confidence what it is doing and for whose benefit. A programmer who came to this book asking why some systems stay understandable for decades while others decay within a year has here the general form of the answer: the first kind is factored where the world it represents is factored, and is kept that way as it changes (Chapters 8 and 10 show how).

> **Method note: on the limit of the analogy.** The compression analogy also shows where factoring ends, and the place where it ends is informative. You cannot make system architecture infinitely simple for the same reason you cannot compress a file infinitely. At some point you reach something incompressible: the state in which all the commonalities have been factored out naturally. That endpoint is not a failure of technique. The incompressible endpoint is the shape of the architecture itself, which is what we were trying to see.

III. Whose Intentionality?

Sections III to VI give the ground of Chapter 12: why a person needs to know that the code a computer runs is the code that was intended, and why a drawing the author can read is the way to know it. These sections say what the way of knowing must be: something a person can look at and read. The drawing that meets that need comes in Part II, and Chapter 12 joins the two.

At present, systems carry out very well what a handful of executives and marketing directors in a corporation intend. They do not carry out well what communities intend. The reason I teach system design with

what are rather powerful ideas is that I want communities to be able to benefit from our capacity as human beings to represent intentionality using systems and code.

Whose intention does a system carry when its code is private? Any time a commercial organization keeps its code private, it is declining to tell you what its intentions are.

For every piece of software running on the device in your hand whose source you cannot see, you do not know what that organization is doing or wants to do. The system is not representing your intentionality. It is representing theirs. And the presentation is arranged so that your intentionality appears to be wrapped inside theirs, so that yours is folded within theirs. It is a Trojan arrangement: trust us.

A reader may ask at this point whether this is merely a matter of degree, and whether in practice most organizations are not malicious. The question that decides the matter is whether the relation of intention to outcome is inspectable, and malice is a different question. Where it is not, the alignment of your interest with theirs is a matter of hope rather than of verification, and hope is not a governance mechanism.

IV. The Regress of Trust

Suppose we grant everything and open the source. The problem does not close there, and it is worth walking down the whole ladder, because the shape of the regress determines what a remedy must look like.

Start at the bottom. If I am working with a new processor there is a document specifying what the instructions are and what they do: this bit pattern will transform that register in this way. You do not actually know for certain that the document describing the hardware describes the hardware. The possibility that a hardware document misdescribes the hardware is not hypothetical. There was a period, some decades ago, when a widely deployed processor performed division differently from its specification: the floating-point division error of the Intel Pentium, found in

1994, which led to a replacement of the chips that cost Intel $475 million (Shirriff 2024). Errors at that level have been found since; they are simply not much discussed.

Go up one rung. Suppose I have a closed-source library with a published interface. Do I know that the function calls do what the document says? There may be no deception at all and still mistakes.

Go up again. Suppose I have the source. There is an argument, the trusting-trust argument, which asks whether we can know that the compiler translating this source implements a machine code that is in any sense the same as what the source says. Ken Thompson set it out in his Turing Award lecture, "Reflections on Trusting Trust" (1984). A compromised compiler can insert behavior into its output that appears nowhere in the source it was given, including behavior that recreates itself the next time the compromised compiler compiles a clean copy of its own source. Does the source you are reading correspond to what the application is doing?

Go up again. You might inspect the machine code to find out. But you do not know that the tool you are using to inspect the machine code is giving you a true view of the machine code.

So we are trusting the hardware, and the language, and the compiler, and the debugging tools; and this is before we say anything at all about how much we may be trusting the other people in our community. The regress of trust has the shape of a ladder: each rung is checked with the tools of another rung, and each checking tool needs the same trust that the checking tool was meant to supply. So a remedy must shorten the ladder, and bring what the code is and what the machine does close enough together to be looked at side by side.

The regress of trust matters because a community which decides together what the community wants, and encodes that decision, needs to know that the code actually running is the code the community elected to run.

The same problem is visible in governance and has been for a long time. A policy is passed; but where it is interpreted is in the courts and in day-to-day enforcement. There can be enormous variance between the spirit of the law, the letter of the law, and the actual outcome. What is wanted is a far more immediate relation between what the code is and what is actually happening: one that can be looked at, and looked at in a way that is comprehensible to ordinary people.

V. Clarity, Not Simplicity

The demand for a relation that ordinary people can look at is where third normal form re-enters, as something that a community needs in order to check the systems that the community lives with, and no longer as a storage technique.

Normalization makes it easier for everyone to understand the code, not only the software engineers who happen to be members of the community. Suppose I am a member of the community and not yet an engineer, but I have concerns. I want to learn enough to inspect the system: to do a check-in (in the sense of checking up on what the system does, and not of committing code).

Suppose my concern is what the system does with my address. In a normalized system there is one table of addresses, and every module that can reach an address holds a foreign key into that table. I read those keys, and I have the list of every module that touches my address. I then read each of those modules by name, and I have what I came for. In a system that keeps copies, my address may be in any module, under any name, and nothing in the system says how many copies there are or what reads them. I would have to read all of it, and I would have no way to know that I had finished.

The precedent for this is not obscure. When one power wishes to verify that another is not producing certain weapons, protocols exist governing how inspection is arranged so that it is both possible and trustable. We insist on this because the stakes are high enough that we must know that what we think is happening is what is happening.

So the questions are: what does it mean for intentionality to be represented in code; what does it mean for that code to be inspectable; and what does it mean for it to be inspectable by any sufficiently motivated, reasonably intelligent person? Note that one does not have to be especially intelligent to understand code. Ordinary people can do it perfectly well, so long as we do not encumber the thing with features that make the basics impossible to see. Engineers have a name for the people who add such encumbrance, *architecture astronauts*, the name Joel Spolsky gave them (2001): they add elaboration for its own sake, which hides the underlying primitives instead of making them plain. Very few primitives are needed to represent any quantity of intentionality one wishes to represent.

So I must answer a question here: why not simply demand simplicity?

Because simplicity loses. Anything that depends on logic is stuck in an arms race, and the logic can always be made a little more complicated than the person doing the inspection can follow. The two sides in that race are whoever would hide an intention in a system and whoever is looking for it. You cannot factor out an arms race by demanding simplicity; complexity will always trump simplicity.

Suppose two programs side by side. One is two hundred lines long, all of it one function, with variables named a, b and c. It is small, and no one can say what it does without tracing every line. The other is two hundred thousand lines long, with one module for each kind of thing the business deals with, its customers, its orders and its invoices, and each module has the same few operations in the same places. It is large, and a person who knows the business can go straight to the part in question and read it.

What trumps complexity is clarity. And clarity shows up in the right application of principle: in knowing where the refactoring points are. Clarity is not a quantity but a correspondence: the divisions of a system match the divisions of what it represents, so that the system is understood the way the world is understood. Throughout this book I use the word 'clarity' in that sense, as a correspondence. It answers the bound that *Software, Wildness, and the Normal Form* sets out in its chapter "The Bandwidth Bound on Value": a person can rely on a system only as far as the finite time they have lets them come to know it, and clarity is what lets that knowing keep pace with what the system can do. Think of a person with one free evening and that program of two hundred thousand lines: that is the case the bound describes. Third normal form is one such principle. It is not a demand that the system be small. It is a demand that the system be cleaved where the world it represents is cleaved, so that a person who understands the world thereby understands the system.

Sidebar: why physics and not cryptography. A technologist may expect that cryptography can give a community this assurance. The same reasoning leads to the opposite conclusion: to the question of how a community can check what its systems do, cryptographic construction is not the answer, and anything resting primarily on cryptology is not the answer.

Everybody lives in the world of physics, and none of us gets to change the laws of physics arbitrarily. Mathematics is likewise not arbitrary; but because mathematics is decontextualized, it can be recontextualized arbitrarily, and that creates vectors of power which are occult. Occult in the literal sense: hidden, unseeable, as an eclipse is an occultation. Intention hidden in this way, behind a secret key or in closed source code, is what I call occult intentionality. You do not necessarily know who holds the secret key. Whether a given generator has been compromised is an open question, and it is an open question that cannot be closed. The case readers will know is Dual_EC_DRBG, a generator of random numbers that the United States National Institute of Standards and Technology withdrew from its recommendations in 2014, after evidence that a weakness had been planted in it (National Institute of Standards and Technology 2014). The same holds for the cryptographic infrastructure beneath digital currencies. Security that rests on logic depends on a complexity that can always be made one increment harder to inspect than the available tools of inspection allow. So whoever holds more computational or mathematical power can hide intent indefinitely, and no outside party can establish with certainty whether the holder of that power has done so. The same phenomenon has now appeared in machine learning: it is possible to embed intentionality in an ordinary neural network in a way that is undetectable, as Shafi Goldwasser, Michael Kim, Vinod Vaikuntanathan and Or Zamir showed in "Planting Undetectable Backdoors in Machine Learning Models" (2022). An architecture connects all the way down to physics when each element a person reads corresponds, one to one, to an element of the hardware, so that the check is made by reading the one beside the other and needs no secret key (Chapter 12). An architecture that connects all the way down to physics is inspectable in a way that one resting on hidden logic is not.

VI. Code Colonialism

The final step is to say plainly what kind of act is involved when an uninspectable system is placed into a community's life.

Consider an outside body arriving in a place and informing the people who live there how they will be helped, and that if they do not care for the arrangement, that is unfortunate. No one asked the people who live there what they intend. It is a failure of community process. It is a destruction of culture. And we have a word for it.

Now consider an application running on a device you own, which you cannot alter, which updates when its makers choose rather than when you choose, and whose behavior you cannot inspect. Such an application is foreign agency introduced into an environment and permitted to run unchecked without consultation with the local environment.

The structures are identical, and so cultural colonialism and code colonialism are the same thing at two scales, and not two things that resemble one another. Where a local community has maintained a system for a very long time, the reasonable presumption is that they know something about it that the arriving party does not, and the reasonable procedure is consultation. The same presumption should govern the introduction of a system into the daily life of a people.

Think of a town. Its water board decides the matters of water, its school board the matters of the schools and its road commission the matters of the roads. What the town intends about its water is to be found at the water board. Suppose the town's system is built with one module for each of those matters, and each module answers to the body that decides that matter. A resident with a concern about the water reads the water module, and can ask the water board what it was meant to do.

So my positive statement is that it is possible to construct architecture which factors intentionality in the same way that a community naturally factors intentionality, and to show a one-to-one correspondence between

the two. Where that correspondence holds, a community can verify that the intentionality it holds is what is actually happening in the systems that affect it.

VII. The Trust Equation Reified in Three Connections

I wrote Section VII and Section VIII of this chapter for readers who want the ground of trust in the Immanent Metaphysics, and the ground, in the Immanent Metaphysics, of how people and machines relate to each other. The engineering argument of this chapter is complete without these two sections.

Inspection is one condition of trust; the published formula for trust shows what the others are, and the normal form bears on each.

There is an existing formula for trust, from the study of advisory relationships. In its published form (David Maister, Charles Green, and Robert Galford, *The Trusted Advisor*, 2000), trust is the sum of credibility, reliability and intimacy, divided by self-orientation. It is useful enough to serve as a working diagnostic for an actual organizational failure. Suppose a firm whose people are skilled and keep their dates, and whom its clients still do not trust: the formula directs the diagnosis to the divisor, the firm's self-orientation. I read its terms as competence, intimacy and reliability, divided by self-interest, and I read the three as a product and not as a sum, for a reason that follows from their place inside the three-part modal structure used throughout this book.

The three terms are three connections: to self, to other and to world. Competence is connection to self: a settled, internally consistent basis from which to act. Intimacy is connection to other. Reliability is connection to world, verified the way an experimental result is verified, by repeatable observation and comparison with others doing the same test. Each of the three connections is stabilized by the other two, and collapses if either is removed. A person with no memory cannot trust other people or the world, since there is no continuous self against which to check

either. A person with no other people cannot fully trust self or world either, since language, skill and correction are themselves learned from others. Because each of the three connections collapses if either of the other two is removed, the three connections combine as a product: a product falls to nothing when any one of its factors does, and a sum does not.

The normal form bears on each of the three. Competence is a settled, internally consistent basis from which to act, and a system in which each fact lives in exactly one place has no second copy of a fact to disagree with the first. Intimacy is connection to other, and a system divided where the world is divided can be read by another person who knows that world. Reliability is verified by repeatable observation and comparison with others, and a factored system gives every observer the same parts to look at.

The divisor, self-interest, is the degree to which choice is made from a surface, ego-bound sense of self rather than from a deeper, more encompassing one. Go back to the firm whose clients do not trust it. An adviser there who asks first what a recommendation will do for this quarter's commission is choosing from the surface sense of self. The same adviser who asks first what will serve the client, the firm and the work they share is choosing from the deeper one. The same distinction, between a surface self and a deep self, decides the basis from which political choices are made.

VIII. Man, Machine, and Nature: The Third Pole

In this chapter I have asked for whom a system is normalized. The widest form of that question takes in people, machines and the natural world together. The largest claim of this chapter (largest in how far it reaches; the engineering argument is complete without it) connects the trust equation and the normal form to a much older tension: how the relationship among people, technology and the natural world should be held. The relationship becomes tractable when it is held among three poles, humanity, machine and nature, each independent of the other two. Framed

as a dispute between two poles, man and machine, it is unsolvable in principle, because any two-pole framing of a three-part problem is unsolvable; the persistent failure of philosophy and game theory to resolve the *principal-agent problem* (how one party can make another, who acts on its behalf, act in its interest) or the *multipolar trap* (a situation in which each party, acting in its own interest, drives all of them toward an outcome none of them wants) with two-sided analysis alone follows from that unsolvability. The reason is Axiom III (Introduction, Section III): where a problem has three parts that are distinct, inseparable and non-interchangeable, a framing that holds only two of them leaves the third acting without being represented, and no balance struck between the two can account for what the third does. Each of the three poles, man, machine and nature, is a counterpoise against imbalance in the other two poles. The three connections of the trust equation stabilize one another in the same way. In the same way again, three archetypal models of governance (meritocracy, democracy and consensus) become collectively workable only once none of the three models is collapsed into, or treated as a subset of, either of the other two.

Suppose the crews of a logging company and the machines that are replacing them are argued over as people against machine. However the two are balanced, the forest grows back at its own rate, and neither side's gain accounts for that. Held among three poles, the state of the forest is a fact that checks both: it limits what the people may demand and what the machines may be set to do.

The specific danger at the machine pole is artificial intelligence capable of producing further code or further intelligence. Ordinary code already places one level of indirection between human intention and eventual outcome. Such a system adds a second, and past that second level the resulting system destabilizes in a way that resists correction, because the chain of trust already traced (hardware, compiler, inspection tool) is compounded by a further, less inspectable layer of machine-generated design decisions. Machines are properly intention-neutral, useful precisely because they do not, by default, want anything on their own account. The moment

a machine can generate the intentions of another machine, that neutrality is lost, and with it the very property that made machine mediation trustworthy in the first place. With one level, every intention in the code is a person's, and the code can be checked against the person who meant it. With two, the intentions in the generated code are the machine's, and no person stands behind them to be asked what they were meant to do; each further level is checked only against the level that made it, and so an error at one level is carried, unseen, into every level beneath.

IX. Summary

Third normal form began as an answer to a narrow question about storage, and it turns out to be a statement about how representation works. Software architecture and database architecture are the same architecture. Normalization is compression. Compression at the level of architecture is the factoring of intentionality. And intentionality, factored well, cleaves where our understanding of the world cleaves.

The consequence is that architecture is not a private matter. Where a system represents intention and its factoring is inspectable, a community can verify that what runs is what it chose. Where the factoring is hidden, it cannot, and it is at the mercy of an intention it cannot read.

Simplicity will not rescue us from this, because complexity beats simplicity in every arms race. Clarity will, because clarity is not a quantity but a correspondence: the system understood the way the world is understood. Right application of principle means a system that is understood the way the world is understood, and third normal form is one of the principles.

The mapping of classes, instances, methods and pointers onto tables, records, columns and foreign keys holds without exception, including in the degenerate cases, and the multivariate convention puts every input and output of a function into the record, so that each cell holds one value. The step from compression to the factoring of intentionality is an identific-

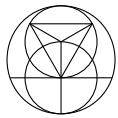
ation: the operation performed in factoring a system and the operation performed in factoring an understanding of the world are the same operation. The same structure of three holds trust together, as connection to self, to other and to world, and holds the relation of humanity and machine together, with nature as the third pole.

The language a system is written in, and the tools that read it, are layers of that system too, as the ladder of Section IV showed. If a system must be factored where the world it represents is factored, what must a model of computation be able to do for that factoring to reach every layer of a system, its own included?

KEY CONCEPTS IN THIS CHAPTER

- **The architectural identity**: class to table, instance to record, function to column, pointer to foreign key; the mapping is exact and admits no exceptions.
- **Multivariate convention**: treating functions as having multiple inputs and outputs, which allows any call signature to be regularized into fields of a record.
- **Points of articulation**: the natural joints at which a representation should be divided; the joints of the world it represents.
- **K matrix**: the worst-case connectivity in which every element of one set may connect to every element of another (what graph theory calls a complete bipartite graph); the thing normalization avoids.
- **Contact space**: the middle bank of a switching arrangement; normalization reduces it from about the number of entries multiplied by itself to about the number of entries multiplied by its logarithm, as a multistage switching network does (Clos 1953; Beneš 1964).
- **Normalization as compression**: third normal form factors out redundancy exactly as file compression does, with the same asymptotic limit.
- **Factoring intentionality**: the identification of architectural compression with the cleaving of understanding; the central identification of the chapter.
- **Closed source as concealed intent**: where source is private, the system's intentionality is unavailable for inspection by those it affects.
- **Occult intentionality**: intentionality that is hidden from inspection, as in closed-source code; used to describe how a corporation's undisclosed intentions can be presented as though they served the user's own.
- **The regress of trust**: hardware specification, library interface, compiler and inspection tool, each requiring trust in the layer beneath it.
- **Trusting-trust problem**: the classic computer-science argument that even full access to source code cannot guarantee trustworthy behavior, since the compiler translating that source could itself be compromised.

- **Clarity trumps complexity**: simplicity loses an arms race with complexity; clarity, as right application of principle, does not.
- **Code colonialism**: the imposition of an uninspectable system upon a community, structurally identical to cultural colonialism.
- **Trust equation reified**: the mapping of competence, intimacy and reliability onto connection to self, other and world respectively, with self-interest read as the degree of orientation toward a surface rather than a deep sense of self.
- **Man, machine, nature triad**: the claim that humanity and technology cannot be stabilized as a two-pole system, and that an independent third pole, the natural world, is required to prevent either of the other two from dominating.



Chapter 3

Third Normal Form as Metaphysics; Self-Reference and the Search for a Complete Model of Computation

What must a model of computation be able to do if it means to be general, and why does code organized in third normal form meet that demand?

ABSTRACT

This chapter shows that *third normal form*, a discipline borrowed from relational database design, when it is applied to code architecture and not only to data schemas, produces systems rich enough to fully describe their own structure. A system describes itself in this sense when it is rich enough to represent its own architecture among the architectures it can represent. The claim of the chapter is that such self-description is a condition of generality, and not a curiosity. I call it the legible self-description condition: any model of computation that means to be general must be able to pass this test. The impoverished foundations most engineers are handed, the Turing machine and Brainfuck (a programming language of eight commands, a byword for willful minimalism), of which the bare Turing machine is the harder to write for, cannot pass it, whatever else can be proved about their computational reach. I develop *Halo*, the model of computation I built to meet this condition, as the necessary alternative: this chapter names it and says what it must do, and Part II gives it in full. I hold that Halo is more than an engineering convenience: it is an expression of a deeper metaphysics, and the same self-referential pattern that makes it work reappears, I argue, in organizational structure and in biological systems. I close this chapter by stating what I establish in this chapter, and the ground on which each of those results rests.

I. Third Normal Form, From First Principles

Chapter 2 left a question about what a model of computation must be able to do. To answer it I need third normal form stated in full, the textbook form behind the short rule of Chapter 1, Section IV. Third normal form, in its home discipline, is a rule about how to organize a relational database so that the same fact is never stored in two places that could quietly drift apart. A table is in first normal form once its fields hold single, atomic values rather than repeating groups. It reaches second normal form once every non-key field depends on the entire key, and not on only part of a composite key (a key made of more than one field). It

reaches third normal form once every non-key field depends on the key alone, and not, by way of some other non-key field, on anything else in the table. The working mnemonic, older than my own use of it, is that every field should describe the key, the whole key, and nothing but the key. Every schema has such a decomposition: any collection of tables, with the dependencies among its fields, can be brought into third normal form without losing any fact or any dependency (Bernstein 1976).

The purpose of the discipline is to eliminate what relational theorists call transitive dependency: a fact stored redundantly because it actually belongs to a different entity than the one the table claims to describe. Applied correctly, it does not remove information. It relocates each fact to the one place where changing it changes nothing else by accident.

Stated in this way, on its face, the rule speaks only of tables: it is a rule for tables of customer records and order histories. As Chapters 1 and 2 have shown, the same discipline does the same work at a different level when it is applied to a program's own architecture (its modules, its call structures, its instances) and not only to a business's data. It relocates every architectural fact to the one place the fact actually belongs. In doing so it exposes a structure that had been present all along, obscured by redundancy and never absent.

II. Code as Its Own Database: The Self-Referential Move

Relational databases already contain a quiet precedent for what I mean. Every serious database system keeps a catalog, a system table, a schema of schemas, that describes the structure of its own tables: their names, their columns, their keys, their relationships to one another. The catalog is itself stored as a table, normalized by the same rules it describes. It is, in the plainest sense available, self-referential: a table that describes the structure of the other tables, including, if you look closely enough, itself.

A codebase organized in third normal form develops the same property. Its modules, its instances, its call relationships can be represented as normalized records, and once they are, the system that manages that representation is describing the very structure it is built from. Any layer of abstraction has this character: rich enough, once properly normalized, to fully describe itself. That character, and not some separate design decision, is why a codebase organized in third normal form can serve as its own database. A structure that can represent its own structure does not need a second, privileged language sitting outside it just to talk about itself.

Suppose a codebase keeps two tables about itself. One has a record for each module: its name, and the key that names it. The other has a record for each call that one module makes to another: the key of the caller and the key of the module called. The module that manages those two tables has its own record in the first table, and its own calls have their records in the second. Which modules can reach the table of customers? That question is answered by reading the second table, in the same way that any question about customers is answered by reading the table of customers.

A practical detail shows the same property in ordinary engineering. When a piece of borrowed code fails, I want to know immediately which component, which function, threw the failure, without hunting through someone else's undocumented call stack. My practice is to tag every function with a globally unique identifier, generated from a fast-scrolling timestamp (a clock reading whose digits change so fast that two readings seldom match). The identifier is cheap, and collisions are rare enough to ignore, so they pose no design risk. Tagging every function with a globally unique identifier is a small instance of the larger pattern. Once components carry their own identity as data, the system can describe, trace and locate its own parts without appeal to anything outside itself. Self-description, in other words, is already latent in the ordinary engineering habit of wanting a system legible enough to debug at three in the morning.

III. Legible Self-Description as a Condition of Generality

The first step of my argument is that a model of computation that claims to be general claims to be able to represent any architecture whatsoever. Its own architecture is some particular architecture among the others. So its own architecture is one of the things it must be able to represent. A model that could describe every architecture except its own would be an exception to its own universal claim, which is a contradiction. Self-reference is therefore the specific place where a general claim is tested against the one case it cannot avoid: itself. Because a general model must be able to represent its own architecture, a general model must have self-reference.

Architecture, in this book, is the factoring of what the people who make and use a system mean it to do (Chapter 2), held at the level at which human cognition engages with code (Chapter 1). A description of a model's architecture that no person can read describes its bits and not its architecture. So the self-description that generality requires is a legible one. The two tables of Section II are a description of that kind: what each record names is a module or a call between modules, and those are the units a person reads and reasons with.

Being able to describe itself is a necessary condition for a model that claims full generality. I call it the *legible self-description condition*. The word legible marks what the condition asks for: a description of the model's own structure, held within the model itself, that a person can read and reason with. A further claim concerns sufficiency: that the normal form covers every architectural form, so that nothing has been left out and every other architectural form is also reachable. The book's argument for that claim is the construction of Chapter 1, The Equivalence Thesis: every module maps to a table, every reference to a foreign key, and every free-standing function to a table with a single row. Chapter 2, The Factoring of Intentionality, adds that there is no escape at the boundary.

Method note. The necessary direction of this argument has real formal company. A universal Turing machine, in the ordinary sense already established in computability theory, must be able to take a description of itself as input and simulate its own behavior. In the literature of computability theory, the meaning of "universal" includes the reflexive requirement that the machine take a description of itself as input and simulate the machine's own behavior. A universal Turing machine meets that formal condition of self-description; it does not meet the legible one, which is the condition at issue here. Von Neumann's work on self-reproducing automata is closer still to what I am describing here: a configuration that contains a complete description of itself, and uses that description both to build a copy and to pass the description along. That precedent is constructive and optimistic, and it is the one I mean to invoke. It should be kept apart from Gödel's use of self-reference, which encodes statements about a formal system within the system itself and produces a limitative result, a true statement the system cannot prove. My claim, and Halo's claim, belongs with von Neumann's use of the pattern: self-reference as the mechanism of generality. Gödel's use is self-reference as the mechanism of a limit. What is shown about one of these two uses does not carry over to the other without an argument of its own.

IV. The Turing Machine and Brainfuck as Impoverished Models

Chapter 1 gave the short definition: a language is Turing complete when it can simulate a Turing machine. To explain more fully what I mean by Turing complete, I need the fuller architecture: the whole Halo diagram, which Part II draws, and the conventions behind it. Chapter 5 says in what sense Halo is complete. Most people, hearing the phrase, picture an actual Turing machine, the strip-of-tape machine from the textbook, and treat it as the obvious reference point for what computation is. It is worth looking at that machine on its own terms, before setting it aside.

A Turing machine is an extremely cumbersome description of computation. It has a tape, divided into cells, each holding one symbol from a minimal alphabet; a head that reads and writes one cell at a time and can move one step left or right; and a finite table of states governing what the head does next. Nobody tries to write programs directly for a machine like this, because it does not even give you logic gates. It gives you a tape of cells, each holding one symbol from a small alphabet, and no built-in operations beyond move, read, write and change state. Everything else, every operation a working programmer takes for granted, has to be built up laboriously from that floor.

Building up every such operation from a bare Turing machine is possible. Mathematicians have done it. But it is very hard, and it is hard in a specific way: almost nothing about its structure helps a person reason about what it is doing, although the machine can eventually compute what needs computing. Trying to write real code for a Turing machine directly is, I would say, maybe worse than writing code in *Brainfuck* (a programming language built by Urban Müller in 1993 to be as small as possible, with eight commands), which is itself a byword for willful minimalism. The comparison is meant seriously. Brainfuck, for all its austerity, actually takes more for granted than a bare Turing machine does: an addressable array of memory cells, paired increment and decrement operations and a matched-bracket loop construct that already gives you structured iteration. A Turing machine gives you none of that. Most people think about computation by starting from this end of the spectrum. They are starting from an extremely awkward space, one that is very hard to reason about, and harder still to make self-describing.

My claim about the Turing machine and Brainfuck is a claim about whether a model can hold a legible description of itself, and that claim leaves computability in the formal sense exactly where computability stands. By the Church-Turing thesis, anything Halo can compute, a Turing machine can eventually compute too, given enough tape and enough patience. Turing-completeness is a floor, not a ceiling, and it says nothing about which floor is livable. My claim is that a model minimal enough to

be maximally general in the computability sense is, for that reason, unable to hold within itself a workable description of its own structure. The reason follows from what this book has already set down. A person reads and reasons about code at the mesoscopic level, the level of named, bounded modules (Chapter 1, Section I), and architecture is held at that level (Section III of this chapter). A model minimal in the computability sense has no named, bounded units of its own: every unit a person could read must first be built inside it, out of cells and states. So any description of its own structure that a minimal model holds is a description at the level of cells and states, which is a description of its bits, and Section III has shown that a description of a model's bits is not a description of its architecture. A minimal model therefore fails the legible self-description condition argued for above, whatever it can be shown to compute given enough time. A model is needed that is far more comprehensive, far lower level in the sense that matters, and far more powerful in terms of how people actually think. Halo is that model.

Picture a circle with a name written beside it. Small round pads just inside its right side take data in and give data out, and small triangles just inside its left side say when it begins and what comes next (Chapter 5). That circle is a named, bounded unit, which is the level at which a person reads. It is also lower level in the sense that matters: the same circle is the unit at every layer of compute, from a logic gate to a whole system (Chapter 13, Section IV), so a person reaches the bottom of a system by reading more of the same circles. When a whole system of such circles is organized in third normal form, the drawing and the normal form of Part I are one structure (Chapter 8). Halo holds a description of its own structure that a person can read, because the drawing is the description, and the drawing is made of units that a person reads.

V. Halo as an Expression of Metaphysics, Not a Convenience

This section is for readers who want the ground of Halo in the Immanent Metaphysics; the engineering argument is complete without it.

I make a further claim for Halo, which this chapter describes without unpacking it in full. Halo is itself an expression of an underlying metaphysics: a broader account of how abstraction, structure and reality relate, which I hold independently of software and which the model of computation instantiates. Halo therefore satisfies the legible self-description condition as a consequence of what it expresses, and its satisfying the condition is not arbitrary. The self-reference examined in the sections above, a system rich enough to describe itself, is what you would expect to find if a deeper commitment about how any adequately general structure must relate to itself were being worked out concretely, at the level of code.

That deeper metaphysics belongs to a more foundational account, the Immanent Metaphysics (the few terms this book uses from it are given in the Introduction, Section III). In this chapter about computation I do not reconstruct the Immanent Metaphysics, and I make no claim that anything in this chapter substitutes for the Immanent Metaphysics. From inside the material, I can say in which direction the relationship runs. The software is what the metaphysics looks like when it is made to answer to the discipline of actually running. The relationship runs from the metaphysics to the software, and not from the software to the metaphysics.

Part II of this book gives Halo itself: its one symbol, the conventions by which a Halo diagram is drawn and read, and the way the diagram is carried into running code. Each of those conventions is a rule of the normal form, applied to a drawing. The principle that Halo draws is third normal form, and so what this section says of Halo it says of third normal form: it is a metaphysical principle, appearing wherever self-reference and the structure of computation are at issue.

VI. One Practical Payoff: Memory for Free

An argument this abstract has to produce something a person can check. In this section I give the clearest concrete payoff of the third-normal-form transform. The third-normal-form transform carries every module of a system into tables in third normal form, by the mapping of Chapter 1. The payoff brings the claim of this chapter down from the level of architecture diagrams to a running system.

Think of a hotel. Its rooms are there from the day it opens, numbered, and a guest who arrives is given the next one free. No room is built for the guest. When every room is taken, the desk says so.

Once a system has been carried through this transform, object creation in the conventional sense disappears. Every instance of a module is now a record of its table, and every record of a table has the same columns, so the size of one record is known before the program runs. Creating an object is then filling in the next free row (the record creation of Chapter 1), and no request for memory has to be made at that moment. Instances are pre-allocated in fixed arrays, sized to a known, generous ceiling, and are not allocated and deallocated on demand. If a given module will never need more than a few hundred live instances, space for a few hundred is reserved up front, whether or not most of it is ever used. The overhead of that reservation is real but bounded, and in exchange the system gets fixed memory consumption and the complete removal of allocation and deallocation as a runtime cost. When a module needs one more instance than its table reserves, the call that asks for it returns through its fail exit (the exit that says a failure occurred), as every call that cannot do its work does (Chapter 6), and the caller decides what follows; the ceiling is set generous so that this is rare, and the fail exit makes it known when it happens. Memory management, in the usual sense of a garbage collector or manual bookkeeping, has already been done, structurally, before the program runs at all. The sizes that make this possible come from the system's

description of itself: the record that describes each module can hold the size of its table's records and the ceiling of its table, so the whole of the system's memory can be read from that description.

That single fact makes it possible to *transpile* (translate source code from one programming language into another) directly from a memory-managed language such as JavaScript into a non-memory-managed target. The target does not need to be taught how to manage memory; the architecture already handles it. The parser for a system built this way is correspondingly simple, because it is no longer doing the work of reconciling two different memory models. A reader can verify this claim directly against a working system built this way, by building one module: give it a fixed array of records, sized to its ceiling, write it in JavaScript and translate it into C, and no line of the translation has to ask for memory or give it back.

VII. Beyond Code: Organizations and Living Systems

This section is for readers who want the reach of the pattern beyond code, which the Immanent Metaphysics grounds; the engineering argument is complete without it.

The self-reference examined so far has been a claim about code. The same pattern reaches past code to organizational complexity and to biological systems, by shared structure. I hold that the metaphysics behind Halo is a toolkit for understanding how those domains relate directly, by shared structure (as distinct from analogy). The abstraction under discussion in this chapter is itself a projection of something still deeper.

The clearest real-world instance of the pattern in living systems is the molecular machinery of a living cell: a genome that encodes, among everything else it encodes, the machinery that reads, copies and repairs the genome itself. The cell's genome, which encodes the machinery that reads, copies and repairs that genome, is self-reference in exactly the sense that I argued for in Section III: a structure rich enough to fully de-

scribe its own structure. Evolutionary constraint arrived at it, and no engineer's decision did. That makes the cell a stronger form of evidence that the pattern is a real one, because it arose where no person's preference could reach.

The organizational case runs in parallel. An organization whose rules for changing its rules are themselves subject to the same process they govern is doing organizationally what a third-normal-form codebase does architecturally. An organization whose governing rules can only be changed by some authority standing permanently outside those rules is not self-referential in this sense. It tends to accumulate exactly the kind of drift between rule and practice that a self-describing structure is built to prevent. I claim that the self-referential organization and the third-normal-form codebase have the same shape, recurring at a different scale of the same underlying pattern. Recurrence across scale is exactly what a metaphysical claim should produce: a pattern that reaches past software, as this one does.

> **Sidebar.** This sidebar is for a reader who wants to take this chapter's question further. Two directions lead out from this chapter that it deliberately does not follow. One leads inward, toward the metaphysics Halo expresses, the subject of the Immanent Metaphysics itself, concerned directly with reflexivity, abstraction and what grounds them, rather than with computation as such. The other leads outward, toward consequence: the essay *Open-Sourcing the World; Universal Transcoding and the Politics of a General Computation*, in *Software, Wildness, and the Normal Form*, takes the same pair of results examined here, a general language and a self-describing general architecture. It asks what it would mean politically if software of any kind could be losslessly moved into that general form and back out again. That essay starts from the architecture given here, in the same way that this chapter starts from the metaphysics that Halo instantiates.

VIII. Summary

In this chapter I asked what a model of computation must be able to do if the model means to be general. I also asked why code organized in third normal form meets that demand. A general model must be able to hold a description of its own structure that a person can read. Code organized in third normal form meets the demand because its modules, its instances and its calls are records, and records can describe the structure they are part of.

In this chapter I give a specific engineering result: a system that manages its own memory as a structural side effect of being organized so that every part of the system can describe every other part.

The larger claim is that legible self-description is a condition of generality for any model of computation that means to be general, and that the Turing machine and Brainfuck fail it despite their formal universality. A universal Turing machine meets the formal condition of self-description; it does not meet the legible one. The necessary direction is argued directly: a model that claims to represent every architecture must represent its own. A structure normalized by the third-normal-form discipline does become self-describing, in a demonstrable sense, visible already in the ordinary system catalogs of relational databases. The claim that the normal form covers every architectural form rests on the construction of Chapter 1. Turing-completeness is a floor: it says a system can eventually compute whatever any other Turing-complete system can compute, given enough resources, and this chapter stands on that floor. The argument concerns human reasoning, and it leaves computability as it stands: minimality in the computability sense makes a model unworkable for human architectural reasoning, as the bare tape, the single symbol in each cell and the absence of built-in operations show.

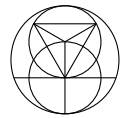
What Halo is for is more than a better way to write software. Halo is what a general architecture looks like when the architecture is required, as a condition of being general at all, to be legible to itself at every layer that someone might need to inspect. That requirement, once stated, matters as

much for an organization or a living cell as it does for a codebase. The essay Open-Sourcing the World, in *Software, Wildness, and the Normal Form*, follows that same generality into its political consequences, asking what it would mean for computational power in the world if a language and an architecture actually met this condition at scale.

Halo rests on the pair of terms the whole mapping uses: a class and its instances. What is a class, what is an instance and how much can an instance differ from its class and still be faithful to it?

KEY CONCEPTS IN THIS CHAPTER

- **Third normal form**: the relational-database discipline requiring every non-key field to depend on the key, the whole key, and nothing but the key, eliminating transitive dependency and redundant storage of the same fact.
- **Legible self-description condition**: the requirement that a model of computation claiming full generality must be able to represent its own architecture, within itself and in a form a person can read and reason with, since that architecture is itself one instance of what the model claims to cover.
- **Necessity and sufficiency**: legible self-description is necessary for generality, and is argued directly; the claim of sufficiency, that the normal form covers every architectural form, rests on the construction of Chapter 1.
- *Halo*: the model of computation of this book, presented not as an arbitrary engineering choice but as an expression of an underlying metaphysics; Part II of this book gives the language itself.
- **Turing machine (as impoverished model)**: the tape-and-state-table model of computation, technically universal but, on this chapter's argument, unworkable as a structure for human architectural reasoning because it lacks any capacity for legible self-description.
- **Brainfuck**: the minimalist esoteric programming language invoked as a comparison; argued here to presuppose more built-in structure, an addressable cell array and loop constructs, than a bare Turing machine provides.
- **System catalog**: the ordinary relational-database precedent for architectural self-reference: a table, stored and normalized like any other, that describes the structure of a database's own tables.
- **Von Neumann self-reproduction versus Gödelian self-reference**: the chapter's distinction between two traditions of formal self-reference, one constructive and used here as the closer precedent, the other limitative and explicitly set aside.



Chapter 4

Classes and Instances

What is a class, what is an instance and how much can an instance vary and still be an instance of its class?

ABSTRACT

This chapter carries a familiar pair of terms from object-oriented programming, class and instance, outside of software into other domains entirely. It uses the ordinary example of a blueprint and the houses built from it. With that example it works through what a class is when it has no location of its own, and what an instance is as a concrete realization of that class. It asks how much variation an instance can tolerate before it stops being a faithful instance of its class and becomes something else. It then shows what work this vocabulary does once it is applied beyond software, wherever the Immanent Metaphysics finds a recurring structural pattern across otherwise unrelated fields. It closes with the principle of uniquity: every instance is unique, because every instantiation takes place in a context that can never be exactly replicated. Fidelity to a class is therefore fidelity to its essential properties alone.

I. The Blueprint and the House

Begin with an example ordinary enough that it needs no technical vocabulary to state: a blueprint for a house, and an actual house built from that blueprint. The blueprint does not exist anywhere in particular, in the way a house exists at a particular address, made of particular boards and nails. What the blueprint specifies is a type of thing (as distinct from a thing): the pattern that any house built to it would need to follow, whether or not any house is ever actually built.

Building a hundred houses from the same blueprint does not require those hundred houses to be identical. They might vary widely in finish, in materials and in small departures made on site, and still all count as houses built to that blueprint. Each counts as long as it retains enough fidelity to the blueprint's essential specification to be recognized as an instance of it, and not of some other design entirely. The example raises a question: exactly where does that boundary of fidelity sit? How much can a built house depart from its blueprint before it is no longer usefully described as an instance of that blueprint at all?

II. Classes and Instances as a Technical Vocabulary

The terms class and instance, borrowed from object-oriented programming, name exactly this blueprint-and-house relationship, generalized. A class, in software, specifies a type: what data a thing of that type will hold, what operations can be performed on it, what its general shape is. An instance is a particular, actually existing realization of that type, created (instantiated, in the technical term) at some specific point while a program runs, occupying actual memory, holding actual values.

Naming the blueprint-and-house relationship this way has a value. Software engineering, in order to be practically workable, had to make the distinction unusually explicit and unusually precise. A running program needs to know, at every moment, exactly which class a given piece of data

belongs to, and exactly what operations are therefore valid on it. Philosophers have discussed universals and particulars (general kinds and the individual things that fall under them), in one vocabulary or another, for a very long time. The precision that practical necessity forced on software engineering is what makes the vocabulary worth borrowing. The vocabulary comes with a discipline about fidelity and departure, which a looser, purely philosophical use of universal and particular does not automatically carry with it.

III. Fidelity, and How Much an Instance Can Vary

This discipline sharpens the question the blueprint raised. An instance, to remain an instance of its class, needs to follow the instructions, the design, the layout given by that class. Following a design is compatible with a real range of variation, since no design specifies literally everything about its eventual instances down to the last physical detail. A house varying in the particular grain of the wood used, or in the exact shade of paint chosen, is still unambiguously an instance of the blueprint, because the blueprint specifies neither.

What the blueprint does specify is binding in a different way: the layout of the rooms, the load-bearing structure, the overall shape. A structure violating those specified elements is no longer a faithful instance, whatever else about it might still resemble a house. The general principle is that a class specifies some properties as essential and leaves others open. Fidelity to a class is fidelity specifically to the essential properties, and not to every physical detail an actual instance happens to have. The philosophical tradition named these two kinds of property essential and accidental, and software design arrives at the same distinction from the practical direction.

In a table the answer is direct. The class is the table, and its columns are what the class specifies: every record has them. The values in a record's cells are what the class leaves open. A record in the table of accounts (Chapter 1, Section V) with a different balance is still an account; a record with no balance column would belong to some other table.

IV. Extending the Abstraction Beyond Software

In this section and the next I carry the pair of terms, class and instance, into the Immanent Metaphysics, for readers who want that ground. A programmer needs from these two sections only the principle of uniquity, which I state in Section V and use in Chapter 5.

This class-and-instance vocabulary has proven useful across many domains, in my study of the Immanent Metaphysics, the broader metaphysical framework behind this book. Class-and-instance terms can describe any situation in which a general pattern is realized in multiple particular, concrete cases, each varying in its accidental details while sharing the pattern's essential structure. The pattern in question may be a physical design, a governance structure, a recurring conversational move or a modal category applied to a particular concrete situation. Take the second and the third of those. Suppose a model charter is adopted by a hundred towns: the charter is a class, and each town's council is an instance of it; who may vote is essential to the charter, and the day of the meeting is left open. An apology is a recurring conversational move: that the speaker names what was done is essential to it, and the words chosen are left open.

Describing such situations in class-and-instance terms imports the discipline just described along with the vocabulary. The discipline is the requirement to be explicit about which properties of a pattern are essential to it, and which are free to vary across its instances. Applied with the discipline software engineering forces on the terms, class and instance offer a useful check. Before calling something an instance of some general class,

one has to be able to say, concretely, what about it is essential to that class and what is incidental. That check is what keeps class and instance from becoming just another pair of words for old and already well-worn philosophical concepts: it makes each use of the pair add clarity. The question of what is essential and what is incidental is worth asking of any general pattern claimed to recur across the widely different domains this book is concerned with: software, data, communities and, in Chapter 3, organizations and living cells.

> **Sidebar.** The same discipline of essential and accidental properties is exactly what is at stake, at much greater length, in Chapter 2, The Factoring of Intentionality, where it goes under the name of finding the points of articulation. There the practical question is which properties of a data record are essential, and therefore belong in a normalized table, and which are incidental, and therefore safely left out or derived. The address of Chapter 2, Section II, shows an essential property: that a shipment has an address is essential to the shipment, and so the shipping manifest holds a key into the table of addresses; and a value that can be computed from stored fields, as the cell of is_overdrawn is computed from the balance (Chapter 1, Section V), need not be stored at all. Leaving out what can be derived removes no information (Chapter 3, Section I), since what can be derived from the facts that are stored is still there to be derived. Chapter 2 treats that question as a general compression principle with consequences well beyond database design. The blueprint-and-house example here is the simplest possible case of that same underlying discipline.

V. Uniquity and Instantiation

Go back to the hundred houses. Build two of them in the same week, from the same blueprint, with boards cut from the same tree, on lots side by side. One of them stands to the east of the other, gets the morning sun first and sits on its own ground. Two instances of one class in a running

program are the same case. Give them the same value in every field, and one still stands at one address in memory and the other at another, and one was created by this call and the other by that one.

The distinction between essential and accidental properties has a ground, and the ground is the principle of *uniquity*: the assertion that every instance is unique and that no two instances are ever perfectly identical.

Every instance of anything is unique. No two instances can be perfectly identical, because identity itself requires a context of comparison, and the contexts of any two instances are necessarily distinct.

Uniquity is a structural necessity, and not an empirical observation. The act of instantiation (the passage from class to instance) always involves a specific context that contributes to the character of the instance. This context can never be exactly replicated. The principle of uniquity is a necessary consequence of the modality framework of the Immanent Metaphysics and of the nature of interaction. The step is short. Instantiation is a process, and the Introduction, Section III, gave process as what happens here and now and depends on its context. Each instantiation therefore has its own here and now, and what it finds there goes into the instance.

The relationship between class and instance is central to the Immanent Metaphysics. A class describes the common features shared by a group of instances. An instance is a specific realization of a class within a particular context. The Axiom II process (the fixed cycle in which a class of one modality precedes an instance of the next; Introduction, Section III) governs the transition from class to instance across the three modalities.

Uniquity ensures that no abstraction can ever fully capture the concrete. Every instance carries something beyond what the class describes: something contributed by the context of its instantiation. That every instance carries something beyond what the class describes is a structural feature of reality, and not a defect of abstraction. And it is why fidelity to a class is fidelity to its essential properties alone. A class specifies what

every instance shares; the context of each instantiation contributes what each instance has as its own; and classification keeps its use because it asks only for the first.

VI. Summary

A short observation about a blueprint and the houses built from it carries a useful discipline, borrowed from software engineering's practical need to be precise about types and their instances. Calling something an instance of a general class obliges a person to say what, specifically, is essential to that class and what is free to vary. The principle of uniquity grounds the discipline: every instance is unique, every instantiation carries something its class does not describe, and so fidelity is owed to the essential properties alone. The distinction of class and instance is portable beyond the software context in which its terms are most precisely defined, and it parallels the philosophical account of universals and particulars. It connects to the larger concern of this book, developed in Chapter 2, The Factoring of Intentionality, with what counts as essential structure in a system and what can safely be treated as incidental detail. In that chapter the concern goes under the name of the points of articulation. In Halo this is why every circle carries its instance at a pole (one of the two marks at the top and bottom of the circle): an instance is never only its class, so a drawing must say which instance it calls and where that instance came from (Chapter 5).

Part I has given the structure: a module is a table, an instance is a record, and a system is well made when it is factored where the world it represents is factored. How can a program be written in that structure directly, so that the structure is drawn, read and run as one thing, and what is the least a language needs in order to do it?

KEY CONCEPTS IN THIS CHAPTER

- **Class**: a type specification, such as a blueprint, that exists nowhere in particular but defines the pattern any instance of it must follow.
- **Instance**: a particular, concrete realization of a class, such as an actual house built from a blueprint, occupying a specific place and made of specific materials.
- **Instantiation**: the act or process of producing an instance from a class, borrowed directly from the technical vocabulary of object-oriented programming; always dependent on the context in which it takes place.
- **Fidelity to a class**: the degree to which an instance must conform to its class's specification in order to still count as an instance of it, rather than of some other pattern.
- **Essential versus accidental properties**: the distinction, forced into practical precision by software design, between what a class specifies as binding and what it leaves free to vary among its instances.
- **Uniquity**: the principle that every instance is unique, since no two contexts of instantiation are the same; a structural necessity of the Immanent Metaphysics.

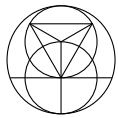
PART II. HALO

One Symbol, Two Kinds of Line, and the Normal Form in Practice

Part I showed that code and data have one structure. Part II gives a language in which a program is written in that structure directly. Halo is a complete programming language, and it is taught here for two things, in this order: to understand the nature of formal structure itself, and then to write software in that structure, which Chapters 9 to 11 do.

Halo is a graphical programming language, and RISL (Reduced Interaction Structuring of Language) is Halo implemented in code across each language: the conventions by which its structure is written in each textual language (Chapter 9). Both are designed for software that must evolve over thousands of years: software that will run on hardware not yet invented, or on hardware a pale fraction of what we have today. The horizon of thousands of years is why Halo holds so little, and I give the reason in full in Chapter 10.

The eight chapters of this part move in one direction. Chapters 5 and 6 teach the language. Chapter 7 says what counts as code. Chapter 8 joins the language to the normal form, at the level of the architecture of a whole system. Chapters 9 to 11 carry Halo into everyday code, and Chapter 12 asks what Halo makes possible for trust. Chapter 11 is written for readers who program in JavaScript, and its Summary states its results for every reader. These chapters teach the language in the first person and one drawing at a time, in the order a drawing is built up on a whiteboard; Appendix A states the same rules in the form a tool is built to.



Chapter 5

The Language in Five Minutes

What is the least a language needs in order to describe any program, and can all of it be learned in five minutes?

ABSTRACT

This chapter teaches a complete programming language in about five minutes. The language answers one question, "what happens when?", and so it is a model of causation. It has essentially one symbol, a circle, which marks the boundary between a function and whatever uses it. Small circles on the boundary, called pads or pins, carry data in and out. Small triangles carry the triggers that start and end a call. A trigger entering a circle pulls the values on its data pins at one moment, and a trigger leaving it pushes the values out. Everything else in the language is a specialization of that one element, and everything built in it is built from primitives supplied from outside the language, as a circuit is built from gates.

I. What Happens When

Part I ended by asking how a program can be written directly in the structure it established, where a module is a table and an instance is a record. Writing in that structure takes a language, and so the first thing to ask is how little a language can hold and still describe any program. Picture a

whiteboard with one circle drawn on it, a few small marks on the circle's edge, and some lines running from those marks. That picture is all of the language. In about five minutes, a reader can learn everything there is to know about a complete programming language, which I believe is a world record. The language is complete in the sense that matters for computation: it is Turing complete, able to perform any classical calculation (any calculation that does not need a quantum computer). It is not complete in the other sense, of a working language with built-in constructs and libraries, which it does not have. Within the classical calculations, it covers all of them.

The language is fully general to everything in the imperative model of programming. An imperative language is one in which you tell the computer what to do. The fundamental question it answers is "what happens when?" It is therefore a model of causation.

It is a fully graphical language. It has no keywords, no built-in constructs, nothing supplied in the box. Even the notion of logic is imported. The only thing the language manages is the notion of what happens when.

With that alone, and any number of arbitrary primitives (functions defined outside the language and supplied to it), the notion of computation can be constructed in general. By "in general" I mean that a program written in it can be compiled to ANSI C (the C language as standardized by the American National Standards Institute), to JavaScript or to assembly language. It can be compiled to something that runs on an FPGA (a field-programmable gate array, a chip whose logic circuits can be rewired after it is made) or on a GPU (a graphics processing unit, a processor built to carry out many calculations in parallel). It can even be compiled to circuits that run straight up in hardware and represent the computation directly. In Halo's formalism there is one concept model that covers all of these targets: ANSI C, JavaScript, assembly language, an FPGA, a GPU, and circuits that run straight up in hardware.

The one thing it does not cover is what lies outside the imperative model. The imperative model is the physical substrate on which all computation in the world is done today. A solver, where you say what you want done but not how, still works within an imperative framework: it eventually compiles down to hardware running an imperative model. The same holds for the languages that are not imperative, such as SQL, Prolog and Haskell. Each is carried out by a program written in the imperative model (a query planner and its executor for SQL, a resolution engine for Prolog, a compiler and its runtime for Haskell), and that program, the form in which the work is actually done on hardware, is what a Halo drawing draws. So by learning the imperative model we learn almost everything that is known about computation today. There are some other models, but this one is fundamental.

The language is very easy to learn, because it is exceptionally simple, and its simplicity is there for the sake of clarity. Anyone who has written software will recognize its constructs: they are the same ones used in every language. What has to be learned is a new way of writing code, with the syntactic sugar (notation a language adds for convenience, which adds no power) and the shortcuts stripped away.

Think of the layout of a circuit board: an engineer reads it to learn what the circuit does, and a machine builds the board from the same sheet. When we describe code and language, we are talking both to people (other software developers) and to machines. That translation in both directions, from human language to machine language and back, is what makes it code rather than just writing. English has one domain of representation, and code has two. So a software language describes an abstract domain of representation, one that both people and machines can read. Because both people and machines must read a software language, the language of this chapter is a drawing: people already know how to read a diagram, and a machine can follow every wire in the drawing.

Computer science is the study of how to manage complexity, since there will always be things in code and in the world more complicated than any single one of us can understand; a language for code manages it by speaking to people and to machines at once. What is the least that can be drawn and still be read by both?

II. One Symbol

The language has essentially one symbol, and it is a circle. The circle marks the boundary between the things that use a particular module and the definition of the module itself. We can think of the circle as the basic function. It stands for the fundamental unit of code, which is usually a function and can as well be a machine instruction or an operator. Whether the circle is the definition of a function or a call of it depends on which side of the boundary we are concerned with: the definition is what is inside the circle, and the use is what is outside it. In the terms of Part I, the circle is a module, and a module is a table. One circle, taken by itself, is that table holding one record: the one instance being drawn (Chapter 8, Section V, gives the table of many).

A function has inputs, and I denote each input by a small round pad drawn just inside the upper right of the big circle (Figure 5.1). Each pad is a point at which the function takes an input. The language makes no distinction of type at a pad: an input can be a value of any type: an integer, a floating point number, a string or any other. I document the type in the same place where I document what the value is for (Chapter 7, Section II), and a function that receives a value it cannot use says so through its fail exit (Chapter 6). A function can also generate outputs. Already here the language departs from most thinking about software languages: it uses *multivariate output*, meaning that a function can produce more than one output. Most programming languages take any number of inputs and produce one output. Here a function can produce any number of outputs, and their pads are drawn just inside the lower right of the circle (Figure

5.2). Pads for any number of inputs and any number of outputs are the multivariate convention of Chapter 2, Section I. By that convention, the inputs and outputs of a call are fields of one record, and each pad of data is one field.

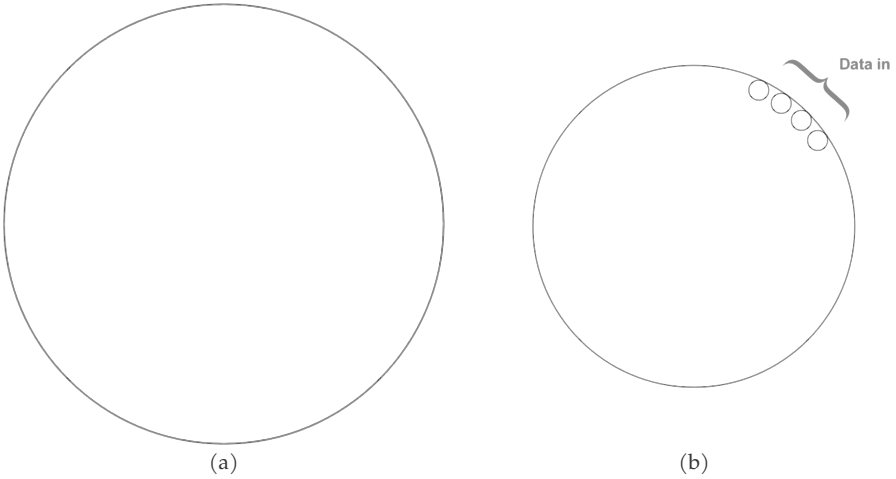
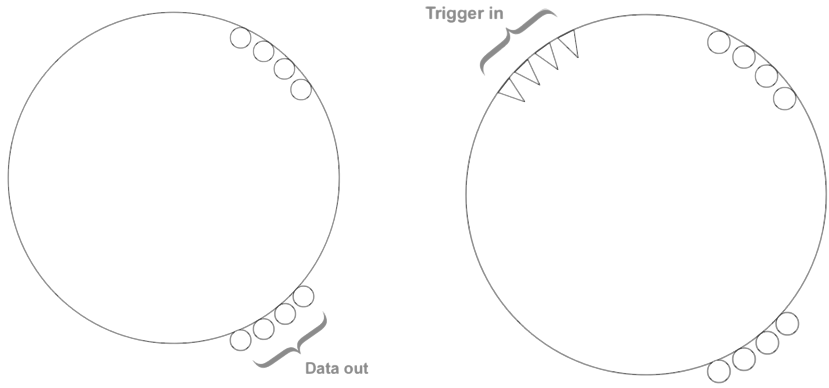


Figure 5.1. The circle, and its inputs. (a) Everything starts with a circle, the fundamental unit of code: usually a function, and equally a machine instruction or an operator. (b) The data inputs, small round pads just inside the upper right of the circle.

Then there are the points that define when a function is triggered, which is to say when it is called. A function can have any number of triggering points that define when it is called, and any number that define when it exits. Allowing any number of triggering points in and out is a second departure from most thinking about software languages, since a function call is usually single entry and single exit. We generalize a little here because we want a construct with enough reach to cover just about everything we could be talking about.

The triggers in are drawn as small triangles just inside the upper left of the circle, and the arrival of a pulse on any one of them is the event of the function being called. The triggers out are drawn as small triangles just inside the lower left, and each is an exit of the function (Figures 5.2 and

5.3). In most programs a function has a single trigger in and exactly two triggers out: one for a call completed with success, and one for a call completed with a failure of some kind (Chapter 6).



(a) (b)

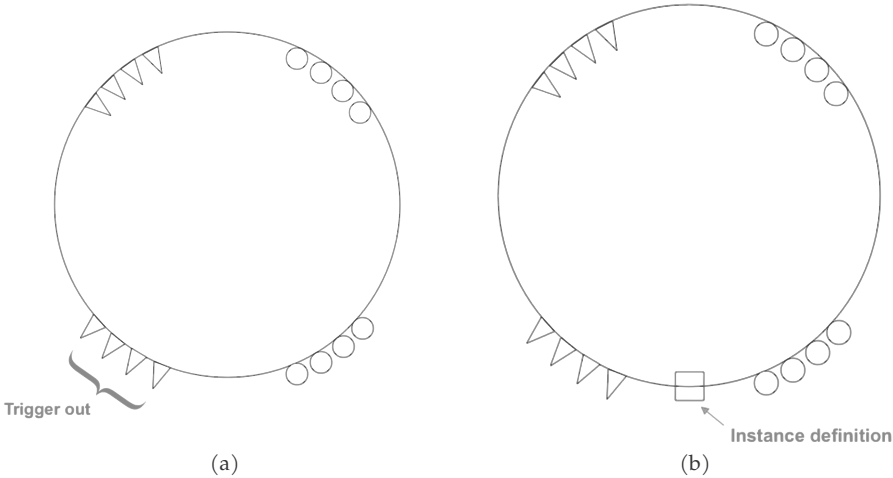
Figure 5.2. Outputs and triggers in. (a) The data outputs, pads just inside the lower right; a function may have any number of them. (b) The triggers in, triangles just inside the upper left; a pulse arriving on any one of them is the event of the function being called.

The reason for wanting that reach, enough to describe any program, is Turing completeness, whose meaning the *Church-Turing thesis* grounds (Chapter 1, Section I): any computation that can be described in one language with enough properties to be called Turing complete can be translated into any other language that is also Turing complete. The requirements for Turing completeness are ridiculously simple: only three fundamental concepts are needed (the sequentiality, conditionality and rings of Chapter 1). The three can also be named sequence, conditionalization and grouping, where grouping is the ability to form rings of sequence, named functions that can be called, and instances of data objects; the ring of Chapter 1 is the first of these. The second naming says what the third concept reaches: the named functions and the instances that Chapter 8 keeps in tables. Once a language is Turing complete it is translatable into

any other Turing complete language, and so an algorithm does not depend on the language it is written in. By describing this one construct in a general way, we describe something essential about what Turing complete means.

All three concepts can be seen on the circle with nothing more than the marks already named. Think of runners in a relay: when one runner finishes, the next one starts. Wire the trigger out of one circle to the trigger in of another, and the second circle runs when the first has finished: that is sequence. Think next of a fork in a road: a traveler who reaches it goes one way or the other, and never both. Give a circle two triggers out, and the pulse leaves by one of them according to what the circle found: that is a condition, and it is the reason a circle needs more than one way out. Think last of a racetrack, which brings a car back to the line it started from. Wire a trigger out back to the trigger in of a circle the pulse has already passed, and the same steps run again: that is a ring. Chapter 6 draws all three in one small program: Print followed by Ring Bell is a sequence (Figure 6.6), success or fail at each call is a condition (Figure 6.7), and a bell that sends control back to Print is a ring (Figure 6.13 (a)). Those three need only one way in and two ways out, and the element reaches further, because a module may have as many exits as it needs, each with the meaning the module designates, and several threads (lines of control running at the same time) may enter one module (Chapter 6, Sections I and V). Chapter 6 then accepts one way in and two ways out as the default, by choice, over that general element.

There are only two other symbols to attend to, and each crosses through the circle at one of its poles. One is a square that crosses through the circle at its bottom, the nadir, with a line hanging off it (Figure 5.3). The other is a diamond that crosses through the circle at its top, the zenith, with a line rising from it (Figure 5.4). Every one of these symbols, the four kinds of pad and the two poles, can have a wire attached to it.



(a) (b)

Figure 5.3. Triggers out, and the instance as defined. (a) The triggers out, triangles just inside the lower left: the exits of the function. (b) The square crossing the circle at its nadir, labeled in the drawing "Instance definition": the instance of the function as the function is defined.

The marks at the top and bottom of the circle are its two poles. The diamond at the top, the zenith, is the instance being called. (This diamond marks an instance; in Halo every circle makes its decision through its two exits, and so the diamond marks no decision.) I can have a function defined somewhere else and want to call that same function. Think of a coat-check ticket: the ticket is not the coat, and it says exactly which coat is meant. The mark at the zenith is a handle for the instance of the function I am calling. The square at the bottom, the nadir, is the instance as the function has been defined. Wires drawn on the outside of the circle treat it as something I am calling, and wires drawn on the inside treat it as something that is called. If I am calling it, I must say where the instance of the function came from. If I am defining it, I must say where the instance is going. It may go, for example, to a data input of another circle, which then holds the function as a value and can call it, as the callback of Chapter 6 does (Figure 6.17 (a)). Every instance is unique (Chapter 4), and so the

drawing names each one. In the terms of Part I, an instance is a record and a reference to a record is a foreign key, and so the line at a pole is the key that says which record is meant.

So I can define a function, or I can call a function. The pole at the nadir means define: I am creating a new function from scratch. The pole at the zenith means call: it is the instance being called. I can write "call instance" at the top and "define instance" at the bottom. One drawing never shows both a definition and a call of the same circle at the same time.

Where the wires attach shows which of the two a drawing means. If I draw the pads and their wires on the outside of the circle, as an element on the page, I am treating the circle as a thing: a function I am calling, which is defined somewhere else, perhaps in a drawing on another page (Figure 5.4). Then I make the circle just large enough to hold the pads it needs. If I draw the pads and wires on the inside of the circle, and make the circle large enough to fill the page, I am treating the whole drawing on that page as the definition of the function.

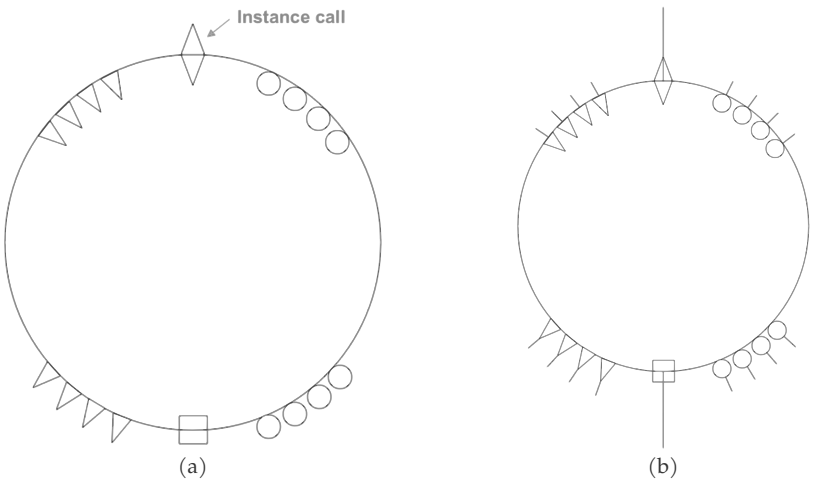


Figure 5.4. The instance as called, and wires. (a) The diamond crossing the circle at its zenith, labeled in the drawing "Instance call": the instance of the function as the function is called, a handle for a function defined somewhere else. (b) Every pad and both poles with a wire attached, drawn on the outside of the circle: the circle is treated as a thing on the page, a function being called.

A call is different from a trigger. The call says which: the diamond at the zenith names the instance. The trigger says when. So I have to notate what I am calling, and when. The trigger pulse says: here is a thing I am triggering, and the pole tells me what that thing is. A trigger says: when I have an instance of a function, take the inputs and carry them from outside the circle to inside, treating the boundary as a kind of wall. Think of the circle as having four quadrants. The data inputs sit in one, the data outputs in another, the trigger in a third and the trigger out in the fourth.

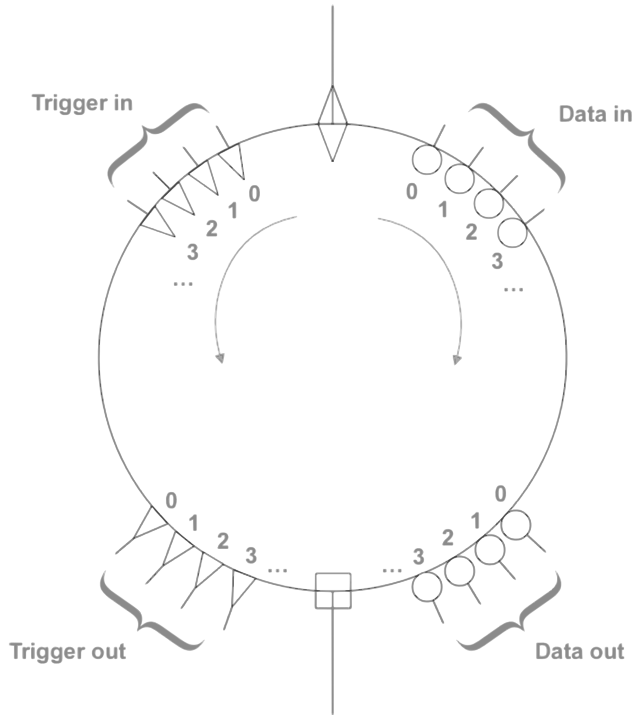


Figure 5.5. The single Halo element, complete. Triggers in sit at the upper left and data inputs at the upper right; triggers out sit at the lower left and data outputs at the lower right. The data pins are numbered clockwise and the trigger pins counterclockwise, as the two curved arrows show, each numbering starting from zero. The diamond at the zenith is the instance being called, and the square at the nadir the instance being defined. A trigger entering pulls the values on the data inputs, and a trigger leaving pushes the values on the data outputs.

III. Pins, Pulls and Pushes

To keep track of things, every pin (another name for any of the four kinds of pad) is numbered in order, starting from zero. Starting from the zenith, the first input is zero, the second is one, then two, three and so on. The output pins are numbered zero, one, two in the same way. Each kind of pin is numbered from the point of the circle where its quadrant begins: data inputs from the zenith and data outputs from the three o'clock point, both clockwise, and triggers in from the zenith and triggers out from the nine o'clock point, both counterclockwise. The data pins are numbered clockwise and the trigger pins counterclockwise, so that there is a trigger zero, a trigger one and a trigger two, as there is an input zero, one and two (Figure 5.5). The number of a data pin is its place in the record, as a field keeps one column of a table, and so a caller and the function it calls mean the same field by the same number.

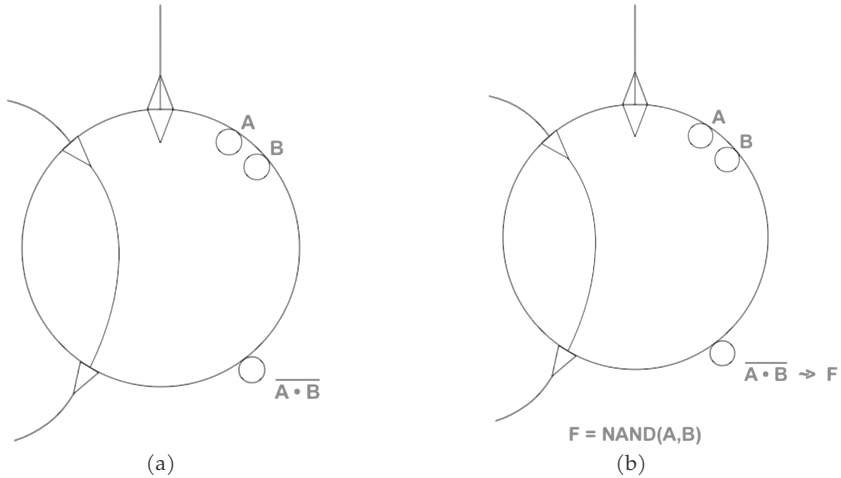
Now return to what happens when. Assume there is an instance of the function, and assume there are sources available for its input lines. When a trigger pulse arrives and hits the boundary, it pulls the instance and all the data pins: at that moment, whatever is at the sources of those data pins is read, and each value is held on the boundary, at its pin. Any one of the triggers can go high (switch on) on its leading edge, and when it does, all of the data pins take their value at the same time. The term leading edge comes from electronics: it is the moment the voltage changes on a wire connected to a pin of a microchip, which signals the chip that something is ready to be done. There is a synchronization. What a trigger synchronizes is the crossing of the boundary by the data. And whenever an exit trigger fires, whenever a high signal comes out of it, I can trust that all of the data lines are set for the output. The firing of an exit trigger is the push: the trigger leaving sets every output at one moment, as the trigger entering read every input at one moment in the pull. A pull reads the fields of the record all at once, and a push writes them all at once.

A trigger is exactly like a processor's clock: at each tick, the clock reads whatever is on the wire. And with that, the language is finished. The circle, its pins and poles, and the pull and the push are all that the sequence, condition and ring of Section II need. Everything described from here on is a specialization of this element, an application of it. It can represent single threaded and multi threaded code. It can represent callbacks (a function passed to another so that the other may call it; Chapter 6 draws one). It can represent all the kinds of synchronization walls (points at which one part of a running program waits for another) we normally think of.

All I am doing here is defining a kind of symbolism. Code can be represented purely symbolically, without even variable names.

IV. A Gate and Its Library

Take a very primitive element, a NAND gate (a logic gate whose output is false only when both of its inputs are true, and from which every other logic circuit can be built). It has a negation on the output side, two input pins for A and B, and one output pin, which is A and B, inverted (Figure 5.6).



(a) (b)

Figure 5.6. A NAND gate as a Halo element. (a) The gate as a circle called from a library, with its instance at the zenith; the trigger line enters at the upper left and leaves at the lower left, passing straight across, and the inputs A and B sit at the upper right, with the output, A and B inverted, at the lower right. (b) The same gate with its output directed to a variable; the drawing writes it F = NAND(A,B), where F names the variable that the text calls result.

Somewhere along the way a clock pulse triggers the gate to take its inputs, and the pulse comes back out again. The pulse route runs straight across. When I want the gate to fire, the trigger wire comes in, pulls the A and B values, and generates an output value, and the output trigger keeps going wherever it is going. So I can treat the gate as an element of an equation or an expression. If I have a function in the form `result = NAND(A, B)`, the output can be assigned to result: I direct the output to a variable named result.

You may ask whether the function is global, available from everywhere. In this case I implicitly took the definition from somewhere: there is a library that defines the function. Since the library defines the function, you might expect the library at the other pole, the nadir. The library comes in as an input, at the zenith, because I am calling the function. The library gave me an instance of the function. Somewhere in main memory there is an address where the function lives, or somewhere in hardware a

place where the function has its instance. As Section II gave it, the pole says what is called, and the trigger says when. In this case I know it is the gate, because the library told me so.

This symbolism can describe any computation whatever. But the notion of a primitive has to be provided. At least some functions must be defined outside the language for its problem-solving capabilities to mean anything. So I build functions out of primitive functions that I do not define.

Physical hardware shows the pattern. A NAND circuit can be made out of transistors. The transistor is provided to me, and so is the NAND circuit. I have a little cookie-cutter instance of a NAND that I can start stamping out, and every one I stamp out is a unique instance. Then I can wire them together to define a logic that does something more complicated, such as addition, or something more complicated again, such as multiplication. Little by little we assemble primitive components into more complicated and more interesting operations, wrap those up, and use them as building blocks for yet more interesting operations. In that way we build a stack of more and more complicated software out of more and more basic software.

What we are looking at, then, is a symbolism that answers one question for any collection of components: what is an algorithm, represented in the most pure form imaginable? Further generalizations of it exist, and they are not worth having, since in actual practice we never need them. So I have settled on this symbolism as a way to represent the concept of computation.

V. Summary

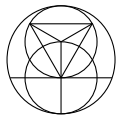
Halo is a complete language in the sense that matters for computation: it is Turing complete, and it is fully general to the imperative model, which is the physical substrate of all computation done today. It answers one question, "what happens when?", and so it is a model of causation, drawn

and not written. Its one symbol, essentially, is a circle, the boundary between a function and whatever uses it, with two lesser symbols that cross the circle at its poles: a diamond at the zenith and a square at the nadir. Small circles carry data in and out, numbered clockwise. Small triangles carry triggers in and out, numbered counterclockwise. The diamond at the zenith is the instance being called, and the square at the nadir the instance being defined. A circle drawn small, with its wires outside, is a function being called; a circle drawn large enough to fill the page, with its wires inside, is a function being defined. A trigger entering pulls every input at one moment, and a trigger leaving pushes every output, so that every crossing of a boundary by a trigger is a synchronization. Everything else is a specialization of this element, and everything built with it is built from primitives supplied from outside the language, as addition is built from gates and gates from transistors. Each circle is a module, and so a table holding one record; each data pin is a field of that record; and each line at a pole is the key that says which record is meant. So the language that takes five minutes to learn is already written in the structure of Part I, which is what the book set out to find: a way to write a program in that structure directly.

The element is now complete. How is a whole program drawn with it, and what few limits does a programmer accept so that the drawing reads the way people think?

KEY CONCEPTS IN THIS CHAPTER

- **Imperative model**: the model of programming in which the program tells the computer what to do, and the physical substrate on which all computation in the world is done today.
- **Halo element**: the one symbol of the language, a circle marking the boundary between a function and whatever uses it, with data pins, trigger pins and two instance poles.
- **Multivariate output**: a function may produce any number of outputs, as it may take any number of inputs.
- **Zenith and nadir**: the two poles of the circle: the diamond at the zenith is the instance being called, and the square at the nadir the instance being defined.
- **Grouping**: the third of the three concepts of computation, stated as sequence, conditionalization and grouping: the ability to form rings of sequence, named functions that can be called, and instances of data objects.
- **Leading edge**: a term from electronics: the moment the voltage on a wire to a chip's pin changes, and that change of voltage signals the chip that something is ready to be done; a trigger acts on its leading edge.
- **Pull and push**: a trigger entering a circle pulls the values on all its data inputs at one moment, and a trigger leaving pushes the values on all its data outputs.
- **Primitive**: a function defined outside the language and supplied to it, such as a NAND gate supplied by a library or by hardware.



Chapter 6

Wiring a Program

How is a whole program drawn with one symbol, and what limits does a programmer accept so that the drawing reads the way people think?

ABSTRACT

A general element becomes a usable language when the programmer accepts a few limits on how it is wired. This chapter gives those limits. Each function has one way in and two ways out (success and fail), and data wires and trigger wires are kept apart. Only data may cross the boundary of a circle, each input has one source, and each trigger leaving a call has one destination. It draws a complete program, hello world, inside the operating system that calls it. It shows why a sequential program is the default: most algorithms carry a sequence that cannot be factored out, and people think in narrative. And it shows that the general language underneath still represents parallel code, callbacks and functions defined inside functions, so that each limit is a choice and not a loss.

I. Limits Accepted for Readability

Every actual programming language places limits on this model. We accept certain stipulations voluntarily, so that the model can be more easily understood by actual developers.

The first stipulation that we accept is the notion of single entry. We learned early, in the 1970s, a discipline called *structured programming*, for which Edsger Dijkstra argued in his letter "Go To Statement Considered Harmful" (1968), and what it says is that there should be only one way to enter a particular function, because multiple ways in make a function more confusing. Single entry is the first limit that we accept by choice. Chapter 1 showed that a function can be viewed as a table whose rows are its states, and one way in means that every call begins at the same row. That table of states is a second table, beside the table of Chapter 5: there a circle is a table in which each row is one instance, and here a single function is a table in which each row is one state that a call can be in.

A second limit is applied often but implemented fully not often enough (it should be, and is not): a function has exactly two trigger outputs. One is labeled success and the other fail (also called normal and signal, or true and false; in code, the return values 1 and 0). For every function I call, I should be able to tell, as the caller, whether the data I am getting on the output pins is valid. If I call an addition function and it overflows the register its output is written to, I want the caller to know that, so that it does not trust the outputs. So we should always have two trigger outputs, success and fail, with fail signaling that an error, a variance (a result other than the one expected) or a signal (an event the caller must be told of) occurred. The general language lets a module have as many trigger outputs as it needs, and the module designates what each one means: one may mark an error while three others mark different kinds of success. The limit accepted here makes two the default, success and fail, so that a reader of any drawing knows what each exit means without looking inside the module.

The fail exit says one thing: that a failure occurred. In code it is received as one bit, and 0 is the only value that means it. Which failure it was is a data output of the call, registered with its other outputs (declared, as they are, among the values the call returns), so that no caller ever makes a second call (such as a call that asks for the last failure) to learn the result of the first; that is what keeps code written this way safe to

run on several threads. Every exit pushes the values on the data outputs, the fail exit as well as the success exit; on a fail exit the caller trusts only the outputs registered to describe the failure. In code the convention is the same: a call returns 1 for success and 0 for the error, and any detail of the failure is part of the call's registered return data set (*The RISL Standard; Halo Written as Code, Across Languages*, Chapter 1, Section V).

Take one call that fails. Suppose the addition call above overflows. The pulse leaves through its fail exit, and the push sets its outputs, one of which says which failure occurred. The caller reads that output from its pin on the same push, as it reads any other output, and asks nothing further. Now suppose instead that a failure were kept in one shared place, like a single message slate by the door of a shared office, and that a caller had to make a second call to read it. Two threads are two lines of control running at the same time. If a second thread fails between the first thread's failure and the first thread's question, the second failure is written over the first on the slate, and the first thread reads a failure that was never its own. Reading the failure from the call's own output is the normal form's rule, applied to a failure. The fact of which failure occurred depends on the one call that failed and on nothing else, and so that fact is kept once, with the failed call's own outputs.

A third limit concerns inputs and outputs, since any input can be mapped to any output. In a function definition, we want to know at once that the wires used for trigger signals and the wires used for data are always discrete, separate from one another. We never wire an input to a trigger line or a trigger line to an input, an output to a trigger line or a trigger line to an output. Kept apart, the two kinds of line let a drawing be read as two separate graphs: the flow of control with no values mixed into it, and the flow of values with no control mixed into it (Figure 6.1). Keeping trigger wires and data wires apart is the normal form's rule that a table is divided where the world that the table describes divides. When something happens and which value goes where are two kinds of fact, and each kind of fact keeps a graph of its own.

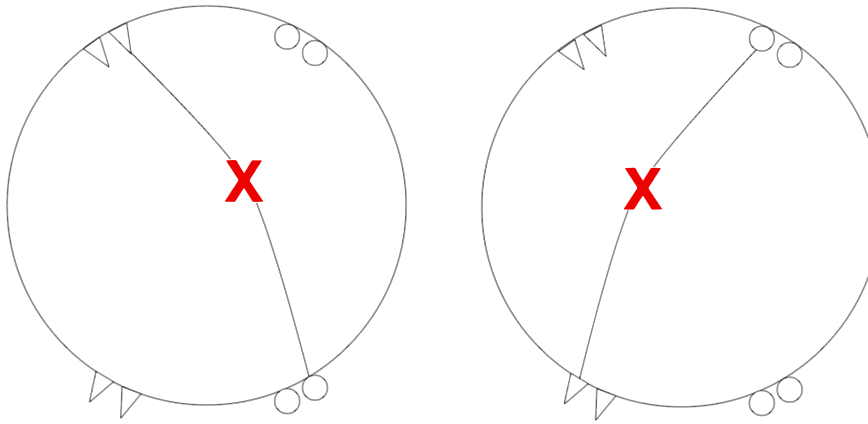


Figure 6.1. Trigger lines and data lines are never joined. Left, a line from the triggers in to the data outputs; right, a line from the data inputs to the triggers out. Each is struck through with a cross, because neither is allowed.

II. A Program Inside Its Operating System

Now draw an enclosing circle, with two functions inside it that it calls (Figure 6.2). Because I am defining this circle, there is an instance: I might create a named function. The system environment will call me, and I will return to the system environment or signal an error. I may get particular inputs from my environment, such as the command line: this pin is the first parameter of the command line, this the second, this the third (Figure

6.3). I have an operating system, I type a command, and it calls a program. The outer circle represents the whole program, and the inner circles represent library functions that the program can call, which can be invocations of the operating system.

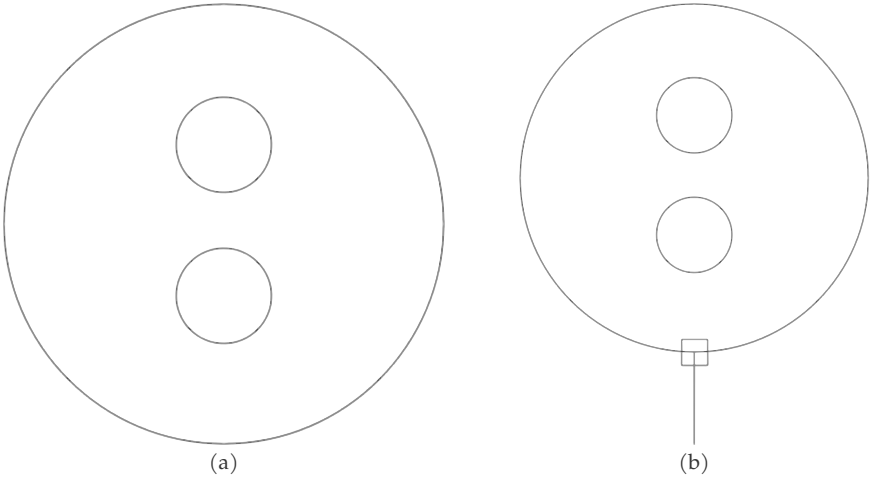


Figure 6.2. A program begins. (a) The enclosing circle, with two circles inside it, the functions it calls. (b) The square at the nadir: the drawing defines the enclosing circle, so it has an instance, the program, which might be named hello world.

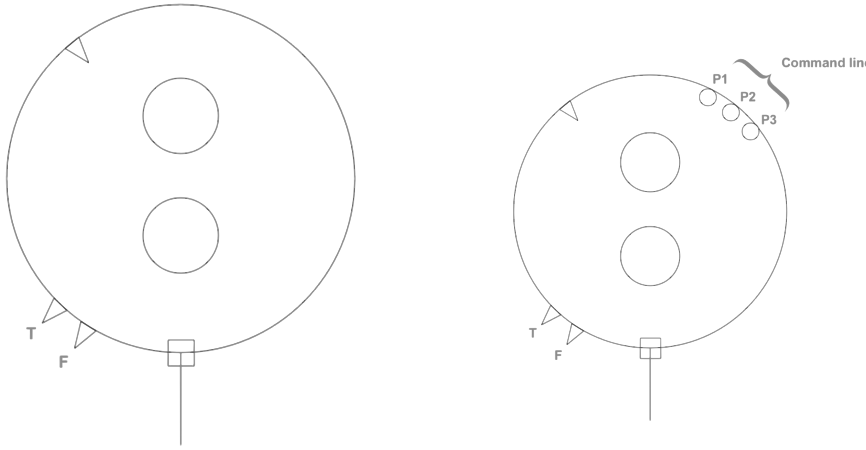


Figure 6.3. The program's connections to its environment. (a) The trigger in at the upper left, by which the system environment calls the program, and the two exits, T and F, by which the program returns true or false to signal an error. (b) Three data inputs, P1, P2 and P3, the parameters of the command line.

Take a hello world program. Say I want to print the words "hello world" and also ring a bell. I have a print function and a ring-a-bell function, and somewhere in the general environment the operating system provides both. Because the operating system provides them, I can label the two circles Print and Ring Bell at their source, and leave the zeniths (the poles at which each circle's instance arrives) wired to numbered pins without names: the lines carry values that come from somewhere outside the program, resident in the operating system (Figure 6.4).

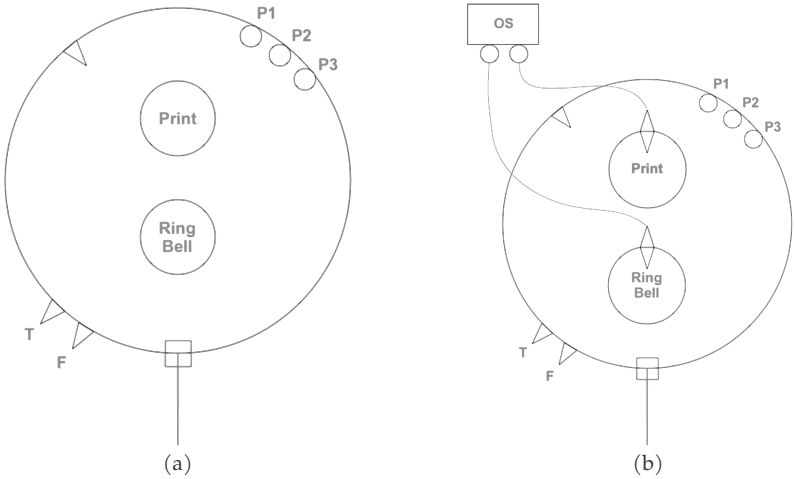


Figure 6.4. The functions the program calls. (a) The two inner circles labeled Print and Ring Bell. (b) The operating system, drawn as a box above the program, supplies both: a line runs from each of its outputs, across the membrane of the program, to the diamond at the zenith of each inner circle, which is the instance being called.

The box labeled OS is the one element of the language, the circle, drawn short, and not another symbol. It is a *leaf node*: a circle whose interior the program does not specify, drawn as a box to show that it stands outside the program and supplies instances and values to it. A starting value on a wire is provided by the operating system, as a leaf node, or by a node whose job is to produce a constant value and for which no interior is specified (Section III gives the second).

In this case a value gets through the membrane, the boundary of a circle, without going through a trigger: it simply crosses the membrane. Data lines can go through the membrane, and trigger lines cannot. Every time a trigger line meets a membrane, it signals a transition, an actual synchronization of the data. Where a data line goes straight across the membrane, there is no necessary synchronization.

Such a line carries what we would call a *global value*, a value that comes from something outside the program, and not from any circle inside it. The outside circle represents the program as a whole. Because the program ends up as a file on the system, it has an instance, and that is what the line at the pole says.

So far only three stipulations have been made, and the first is that a routing wire (a control signal wire, one of the wires connected to the triangles) cannot pass through the boundary of a circle. A data line can pass through the boundary of a circle. And any data line that is connected to an ordinal point on the circle (a numbered pin) is synchronized with respect to the triggering of that circle.

You may ask whether the lines that pass straight through are accessible from everywhere. A line that passes straight through carries a global value, which has whatever state it holds at the moment a trigger pulls it. The lines that have to be triggered are the ones attached to a circle themselves: the attachment points on the boundary of a function call are synchronized with respect to the call. When the call happens, everything on those points is pulled, and each has a definite state at that moment. The data lines that come from the computer at large also have state at that time. When the circle is triggered, that is effectively a pull of whatever the operating system has at that location. Think of a water pipe that runs through the wall of a house: nothing happens at the wall, and the water is drawn when someone opens the tap at the pipe's end. A data line that crosses a membrane without meeting a pin is such a pipe. The crossing synchronizes nothing, and the line ends at a pin or a pole of some circle further in. When that circle's trigger enters, it pulls whatever the line carries at that moment.

As I said in Chapter 5, whenever a trigger enters, it does a pull, and whenever a trigger exits, it does a push. Think again of the four quadrants. Whenever a trigger fires in the lower left, it pushes the data state of everything in the lower right. Whenever a trigger enters from the upper left, it pulls the data, synchronized, from everything in the upper right.

In the hello world drawing, the trigger entering the program is the operating system invoking the program, and the two triggers leaving it return a value to the operating system. When I call a command line program, the operating system knows how it went, because it gets back an integer value that says whether the program succeeded or failed. I represent that explicitly with two trigger output pins. The data lines can be the parameters of the command line: the first one is the first parameter typed on the command line.

You may ask whether the operating system calls the function, and whether the values are then pulled. At the moment the operating system hits the start (there is a main function, for example, or a start of the program), the trigger at that point goes active high (it switches on, and high is the state that means on). At that moment we presume that the operating system holds the command line somewhere in its memory. We could ask the operating system for the command line, but from the point of view of the program the values are already available. So there is a synchronization in this sense: at the moment the trigger fires, the values exist. They already existed, but we know they exist only from the moment the trigger fires. Before that, they might not. When the trigger fires, I accept that the operating system has made those values available.

Normally I would route the program's call to print. Call one trigger output of each call success and the other fail. If the call comes into print and print succeeds, I call the bell function next. If the bell succeeds, the program exits. If either call fails, I go directly to the fail condition. As far as the trigger conditions go, this represents the flow of code (Figures 6.5 to 6.7).

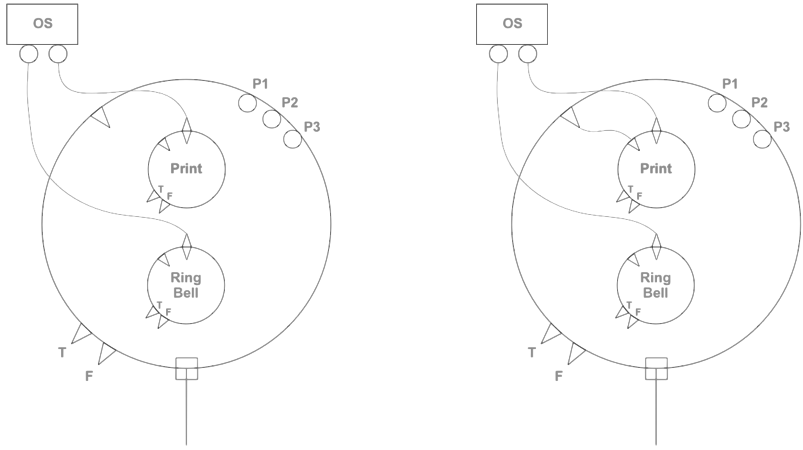


Figure 6.5. Routing the calls. (a) Print and Ring Bell, each with its trigger in and its two exits, T and F. (b) The program's trigger in, from the operating system, routed to the trigger in of Print.

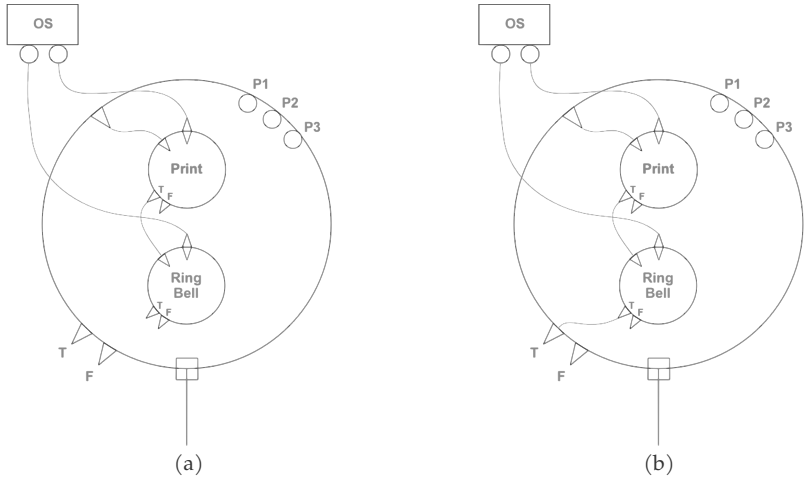


Figure 6.6. Success after success. (a) Print's success, T, routed to the trigger in of Ring Bell. (b) Ring Bell's success routed to the program's own success exit, T.

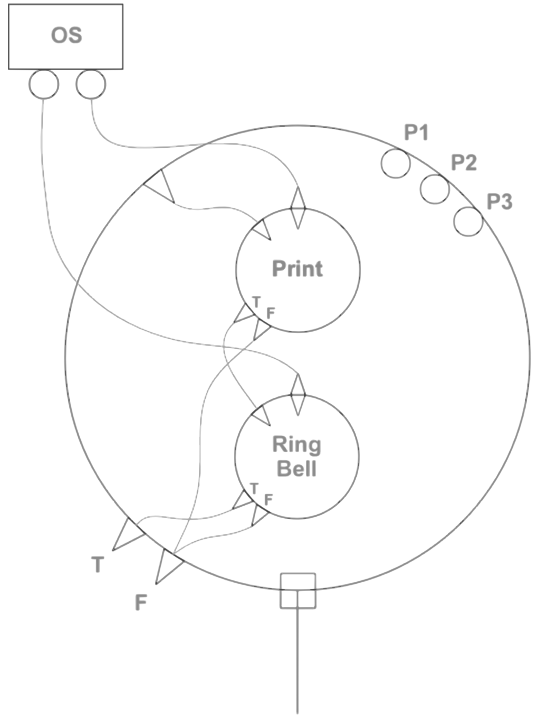


Figure 6.7. The fail exit, F, of each call routed to the program's fail exit, F: if either call fails, the program exits through F.

III. Hello World

This language is designed to show an abstract concept, and its drawings are not meant to be visually pretty. You will end up with big spaghetti code whatever you do, and that is fine, because what we are looking at is an abstract concept, and not necessarily how useful it is in practice. Chapter 7 gives the conventions (a named variable at every pull and push, and named states on the trigger lines) that make a large drawing read cleanly, and Chapter 11 a second way of drawing the same program, closer to a flowchart.

What matters is that the hello world drawing represents an algorithm. The print function takes a string, which the print function uses as a format string. I put a constant value inside, and that becomes my format string. I draw the constant as a lozenge, a long box with rounded ends, with its value written inside it (Figure 6.8). The lozenge that holds a constant is the second source of a starting value: a node whose job is to produce a constant value and for which no interior is specified, drawn short. Then I take the first parameter and pass it. The print function substitutes the first parameter on the command line as the word after "hello", and sends the result to the screen. It generates no outputs. The bell function takes no inputs and generates no outputs. It calls a hardware function internally, and it just does it.

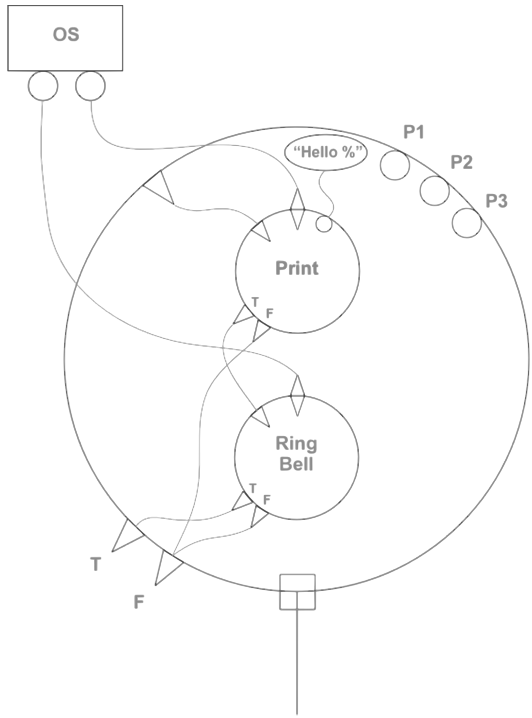


Figure 6.8. The format string: a constant, "Hello %", drawn in a lozenge and wired to a data input of Print.

At this point the program is finished. The finished program is hello world in Halo (Figure 6.9). The language is called Halo because it is a big circle.

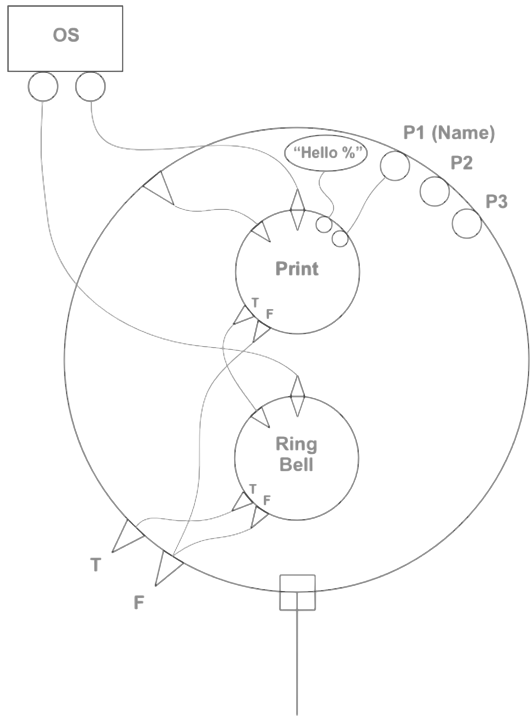


Figure 6.9. Hello world in Halo, complete. The operating system's box supplies Print and Ring Bell. The program's trigger in runs to Print, Print's success to Ring Bell, and Ring Bell's success to the program's exit T, while each F runs to the program's exit F. The constant "Hello %" feeds one data input of Print, and the first parameter of the command line, P1 (Name), feeds the other.

This drawing can be handed to a programmer in almost any coding language, to be translated into the syntax of that language. There is nothing special about the representation other than that it is dead simple. It is a wire layout.

Suppose the print function, rather than sending something to the screen (these are operating system functions), generated an output: say, a number giving how many characters it printed. And suppose I wanted the

bell to ring that number of times. I could wire that output to the bell's input (Figure 6.10). But the output is not available as an input until the trigger exits print, and once the trigger enters the bell, the bell looks for that value. As Chapter 5 gave it, at the moment a trigger enters, it pulls on all the wires, and at the moment a trigger leaves, it pushes the values of all the wires. When it reaches the next circle it does a pull, and when it goes on to the one after that it does a push. So the wiring works: the trigger leaving Print pushes the count onto the wire, the trigger entering Ring Bell pulls that count, and the bell rings that many times.

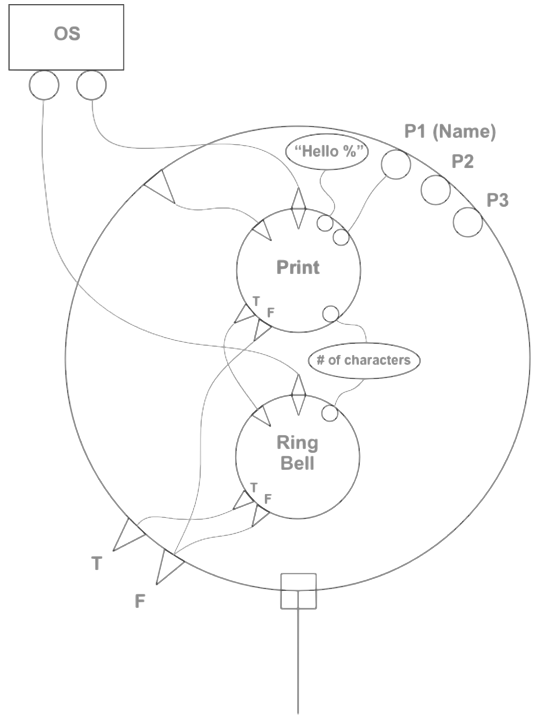


Figure 6.10. Print given an output, the number of characters it printed, wired to a data input of Ring Bell, so that the bell can ring that many times. The value is on Print's output only once Print's trigger has left it, and Ring Bell pulls it when its own trigger enters.

IV. Fan-In, Fan-Out and Loops

At this point we need yet another constraint. The timing is managed by the trigger, and the worry is race conditions (faults in which the outcome depends on which of two things happens to come first). If both a push and a pull are happening, when does the data actually move from one boundary to the other, given that the data lives on the boundary? And if I route two outputs to the same input, which of them actually writes the value (Figure 6.11)?

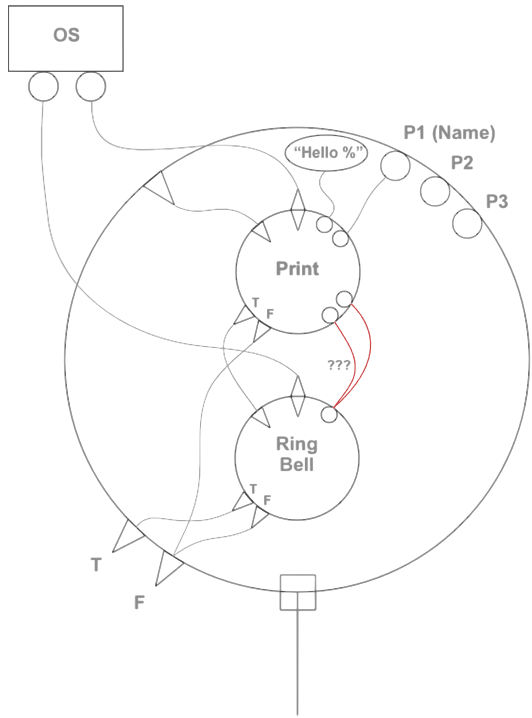
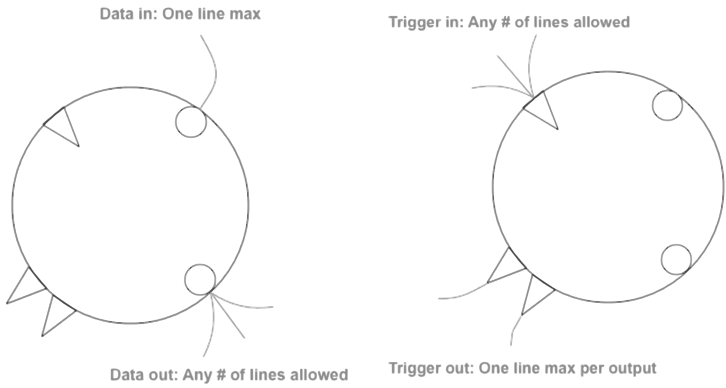


Figure 6.11. Two writers for one input: a second output of Print, the outer of the two curves on the right of the program, which runs from the lower of Print's two data outputs (the red line, drawn heavier), wired to the same data input of Ring Bell as the first. Which of the two writes the value is undefined, and the drawing marks it "???".

Think of a letter left in a mailbox: the sender's part is finished when the letter is in the box, and the letter moves on when someone collects it. The first of those two questions, when the data moves, has its answer in the two moments that Chapter 5 gave. A push sets the value on the output pin and leaves it there, and the value moves at the pull, when the trigger reaches the reading circle and the value is held on that circle's boundary, at its pin. The trigger line runs from the one circle to the next, and so the push at the one comes before the pull at the next. Two inputs can read from the same source safely: each reads the one value that the push left on the pin, and so, through the push and the pull alike, both see a consistent value. The question is what to do for an input with two sources. To settle it, we accept another limit: an input reads from one source. The limit that an input reads from one source is the normal form's rule, drawn as a wire, that each fact is stored in exactly one place. An input with two sources would have no one place that says what the input's value is. For wires going to an input (the small circle drawn on the inside of the ring; here ring means the big circle of a function, and not the loop that Chapter 1 calls a ring), the stipulation is that each pulls from one place only. Fan-in is how many wires arrive at a point, and fan-out is how many leave it. A value that must come from one of several sources passes through a named variable, which may take fan-in from any number of places (Chapter 7, Section II). An output, by contrast, can write to more than one place with no problem. So fan-out may be multiple, but fan-in is singular (Figure 6.12 (a)). Fan-out is drawn as several lines leaving one pin. Think of a road that crosses another on a bridge, with no way from one to the other. Lines that cross on the page do not connect: a line connects only at a pin, at a pole or at a lozenge. Where two lines happen to cross is a fact about how the page was laid out, and a connection is a fact about the program, which only a pin, a pole or a lozenge states. Figure 6.14 shows several crossings: the line that brings Ring Bell's instance from the operating system crosses trigger lines on its way to Ring Bell's zenith, and it joins none of them. Think next of a form with a blank where an amount belongs: whatever a clerk does next rests on something nobody wrote down. Every data input

of a circle that is called is wired: a drawing in which such an input has no wire is incomplete, because the value the function would read rests on something left unstated (Chapter 7, Section I), and a checker (a program that reads a drawing and looks for faults of this kind) reports it.



(a) (b)

Figure 6.12. How many lines each kind of pin takes. (a) Data: "Data in: One line max"; "Data out: Any # of lines allowed". (b) Triggers: "Trigger in: Any # of lines allowed"; "Trigger out: One line max per output".

For the trigger lines we have exactly the reverse condition: any number of things can trigger an input (Figure 6.12 (b)). Say I want to go back from the bell to print and make an infinite loop. I take the bell's true condition and loop it back to print, so that nothing is connected to the program's own success exit any more (Figure 6.13 (a)). Now the program prints, rings the bell, and if the bell succeeds, goes back to print and prints again, round and round forever. Somewhere in the loop I would probably put a test and a variable that says "print ten times", at which point what I want is another module, a counter. Call it Count to Ten (Figure 6.13 (b)). I need an instance of it from somewhere, and here I assume that the operating system provides it, as it provides print and the bell (Figure 6.14). Once the count reaches ten, the program has a natural exit. Now we have a constrained program: it prints whatever I put on the command line after

"hello", and it rings the bell. The drawing shows the routing: a trigger input can accept triggers from many functions, and any number of trigger outputs can be routed to one trigger input.

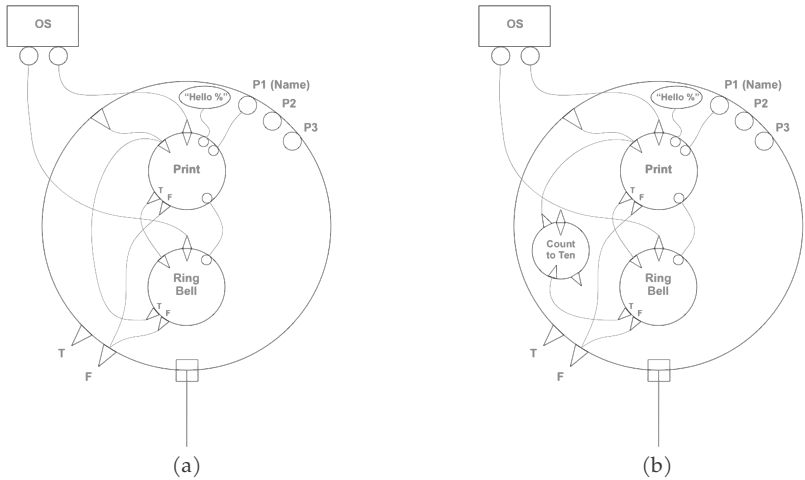


Figure 6.13. A loop, and a counter in it. (a) The infinite loop: Ring Bell's success runs back to the trigger in of Print in place of the program's exit, and the program prints and rings the bell forever. (b) A counter in the loop: the module Count to Ten, wired into the path that returns from Ring Bell to Print.

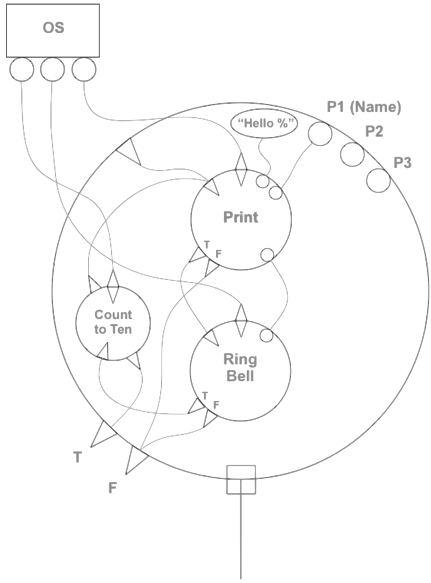


Figure 6.14. Count to Ten's instance supplied by the operating system, as Print's and Ring Bell's are: the operating system's box now has three outputs. When the count reaches ten, the program exits.

A trigger output, on the other hand, can go to only one place, because if it went to several, what would that mean? It would mean a multithreaded program, in which one output triggers several modules at once. If a trigger output of the bell were wired both to Count to Ten and to another module, the two would run at exactly the same time, in parallel (Figure 6.15). We could allow it, but we accept yet another limit, because programmers, for the most part, think sequentially. Our brains tolerate one thought at a time unless we go into special training. Sequential programming is much, much easier, unless you actually need to do multithreaded programming. So we say: no splits on the output trigger events. Chapter 1 showed that a function can be viewed as a state machine, which occupies one state at any given moment, and as a table whose rows are those states; one destination for each trigger leaving a call keeps a drawing at one row of that table at a time.

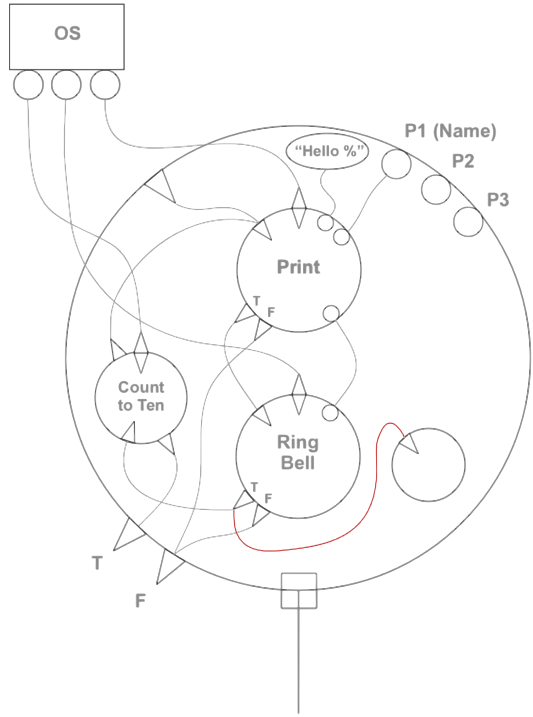


Figure 6.15. A split trigger: a trigger output of Ring Bell, the line that runs from Ring Bell's T exit, under Ring Bell, to the new circle at its right (the red line, drawn heavier), runs both into the loop and to that new module, so that the two would run at the same moment, in parallel. Halo can draw this; the limit of one destination for each trigger leaving a call keeps it out of the default program.

The limit matches how algorithms actually behave. Most algorithms have a high degree of sequentiality that cannot be factored out. People have had multiple processors for decades, but, in my experience, beyond about 8 to 16 of them some problems simply do not speed up when more compute is added: they still have to wait for their data dependencies. Amdahl's law gives the reason: the part of a computation that must run in sequence sets a limit on the speedup, however many processors are added (Amdahl 1967). For a computation of which one tenth must run in sequence, no number of processors gives more than a tenfold speedup, and sixteen processors give a speedup of only 6.4 (one divided by the sum of

one tenth and nine tenths divided by sixteen). And humans work in a purely sequential way, because we think in narrative. Sequential process is, in my experience, about 90 percent of what people care about. Getting less-savvy people to think in multithreaded code is not going to happen.

V. Limits Over a General Language

So what we have done is start with the general language and add constraints to it. JavaScript and ANSI C, for example, are both inherently single threaded languages. You can do things to make them feel like multithreaded languages, more so in C than in JavaScript, but those are amendments (the 2011 revision of the C standard, which adds threads, is one such amendment).

This language can represent multithreaded programming directly. Several intake threads into one module, and several active output threads from one module, are special uses of the same symbolism. Nothing in the drawing prevents several intake threads or several active output threads. The module itself defines how the module works with several threads, and the symbols are used as they are. Figure 6.19, at the end of this section, draws one such module, a join, at which two threads meet. Then you have to pay attention to the wiring patterns, so that you do not create thread problems such as race conditions or the things that cause deadlock (two or more threads each waiting on another, so that none of them can go on). But that depends on the specifics of the actual situation, and it is not inherent in the language. The language is not necessarily preventing you from shooting yourself in the foot.

Another thing I can do is define a function inside a function. The outer ring represents a function definition, for the program. The inner rings represent function calls. But if I want to set up a callback, where I have a function into which some other function may want to inject a call, I can

define a function inside. I draw the new function as a ring inside the program, and I give it a square, an instance definition, at its nadir (Figure 6.16).

Say the bell function wants to issue a callback to something after it rings the bell. It takes the callback as an input, and the definition inside says: when it gets a trigger, it has two output triggers again, success or fail, by convention, and inside it there may be another call to print. The drawings take these steps one at a time: the callback's instance wired to a data input of the bell (Figure 6.17 (a)), its two exits (Figure 6.17 (b)), the call to print inside it, which is the same print the operating system supplies (Figure 6.18 (a)), and a constant defined inside it, "Bell", passed to that call (Figure 6.18 (b)). Now I have a function definition inside a function definition. Its instance is passed to the bell function, and as part of its own definition the bell function may call it: it triggers that pin, the inner function does what it does, and the trigger comes out again. A function definition inside a function definition, passed as a callback, is an example of the kinds of things that you can think about with Halo.

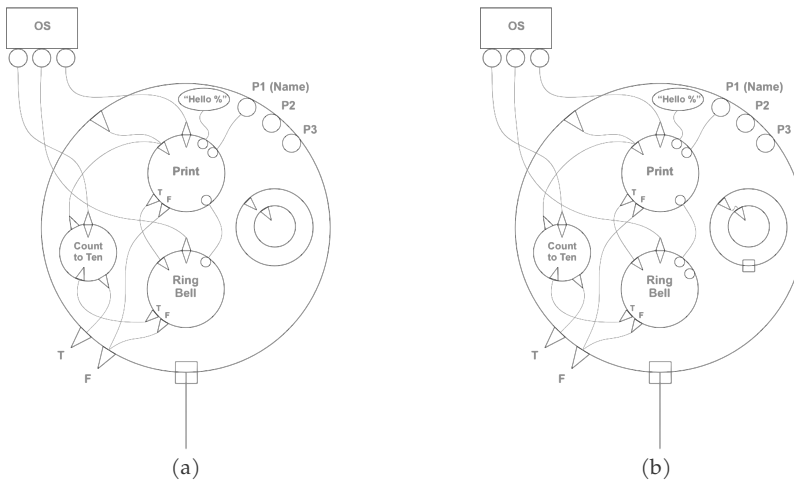


Figure 6.16. A function defined inside the program. (a) A new function drawn inside the program, at the right: an outer ring, with its own trigger in, and an inner ring, a call it makes. (b) The new function given a square at its nadir, so that it is a definition with an instance of its own, and its trigger in wired to the call inside it.

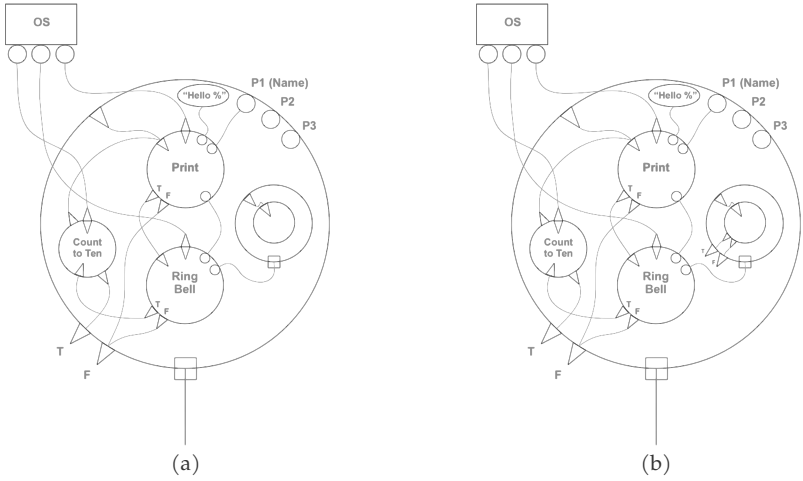


Figure 6.17. The callback. (a) The instance of the new function, from its square, wired to a data input of Ring Bell, which takes the function as an input: a callback. (b) The callback's two exits, T and F, by the convention of success and fail.

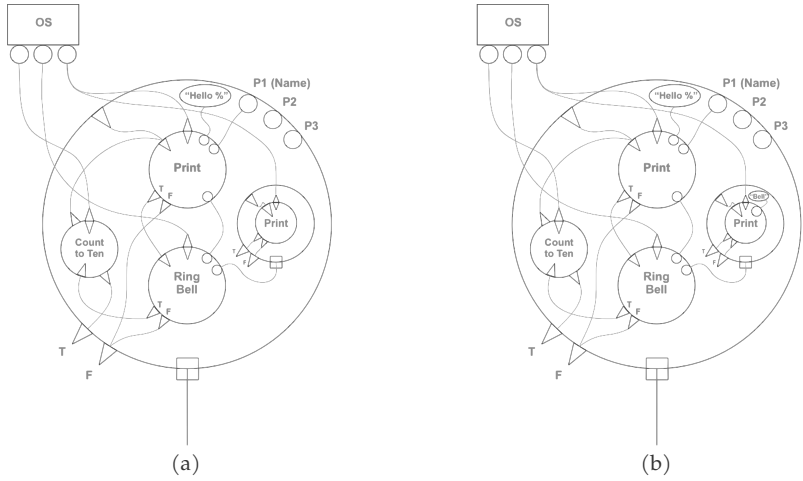


Figure 6.18. Inside the callback. (a) A call to Print, the same Print supplied by the operating system. (b) A constant, "Bell", in a lozenge inside the callback, passed to that call of Print.

You may ask what happens if two triggers reach the same circle at once, and how two threads are made to wait for each other. In the default program the first question arises only in recursion, where a trigger

reaches a circle that is still running through a call made from inside it, and Chapter 8, Section VII, settles that case: the call names a new instance, taken from the table of that function's instances. In the general language, where two threads may reach one circle, what the circle does with a second trigger that arrives while it is still running is defined by the module, as every use of several intake threads is. A *join* is then a module with two triggers in that fires its exit once a trigger has arrived on both of them (Figure 6.19). Two threads that must wait for each other each wire their exit to one trigger in of the join, and the join's exit is the synchronization wall at which they meet: whatever follows the join runs only after both threads have finished.

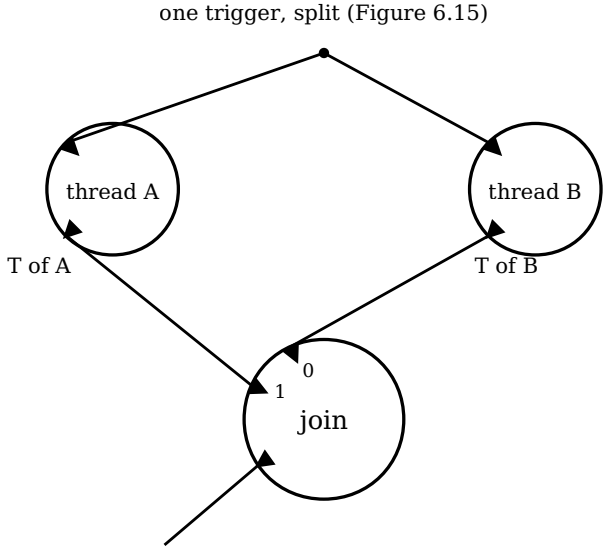


Figure 6.19. A join. One trigger, split as in Figure 6.15, starts two threads, A and B, which run at the same moment. The success exit, T, of each thread is wired to its own trigger in of the join, numbered 0 and 1, counterclockwise from the zenith. The join is a module with two triggers in, and it defines what it does with them: it fires its one exit once a trigger has arrived on both. Its exit is the synchronization wall at which the two threads meet.

With that, I have basically described the underlying concept.

VI. Summary

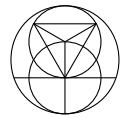
A whole program is drawn as an outer circle, the program, with the circles it calls inside it, wired by two kinds of line. Trigger lines carry control from circle to circle and never cross a boundary. Data lines carry values and may cross a boundary, and a line that crosses without meeting a pin is a global value. The limits a programmer accepts are few, and each has its reason. One way into each function keeps the function from being confusing. Exactly two ways out, success and fail, let a caller know whether to trust the outputs. Data wires and trigger wires are never joined. One source for each input leaves no race over who writes it. And one destination for each trigger leaving a call keeps a program sequential, because people think in sequence and most algorithms carry a sequence that more processors cannot remove. Hello world, drawn this way, is a complete algorithm, and a programmer in almost any language can translate the drawing into that language, because it is nothing but a wire layout. Every limit is a choice laid over a general language that can still represent parallel code, loops, callbacks and functions defined inside functions. In that general language a module may have as many exits as it needs and designates what each means, and a module that takes several threads defines how it works with them, as a join does when it lets two threads meet. The fail exit is one bit, and the detail of a failure is a registered output of the call, so that no caller makes a second call to learn it. The box that supplies instances and values is a leaf node, a circle whose interior the program does not specify; lines that cross on the page do not connect; and every input of a called circle is wired. These limits are also rules of the normal form, applied to a drawing. An input with one source keeps a fact in one place. A failure carried on the call's own outputs depends on that call and on nothing else. Two kinds of line divide the drawing where control and values divide. One way in gives every call the same first row of its table,

and one destination for each trigger leaving a call keeps the drawing at one row at a time. So the limits that make a drawing read the way people think also keep it in the structure of Part I, from the first circle drawn.

The language is now in hand. If any program can be drawn in it, what counts as code, and how can a drawing be made easy to read, and easy to translate in both directions, without changing the language?

KEY CONCEPTS IN THIS CHAPTER

- **Structured programming**: the discipline that a function has one way in, and the first limit Halo accepts by choice.
- **Success and fail**: the two trigger outputs of every function, by which a caller knows whether to trust its outputs (also called normal and signal, or true and false; in code, the return values 1 and 0).
- **Membrane**: the boundary of a circle, which data lines may cross and trigger lines may not.
- **Global value**: a value on a data line that crosses a membrane without meeting a pin. It comes from outside the program and is not necessarily synchronized with the call.
- **Fan-in and fan-out**: how many wires arrive at a pin and how many leave it. A data input takes one source, a data output may feed many, a trigger input may take many, and a trigger output goes to one place.
- **Amdahl's law**: the part of a computation that must run in sequence limits the speedup that more processors can give.
- **Callback**: a function defined inside another and passed to a third, which may call it.
- **Lozenge**: the drawn symbol for a constant or a named value, a long box with rounded ends with the value or the name written inside.
- **Fail exit**: the exit that says a failure occurred; in code it is one bit, 0, and which failure it was is a registered data output of the call.
- **Leaf node**: a circle whose interior the program does not specify, which supplies instances or values; the box labeled OS is one, drawn as a box to show that it stands outside the program.
- **Join**: a module with two triggers in that fires its exit once a trigger has arrived on both, the synchronization wall at which two threads meet.



Chapter 7

What Counts as Code

If any program can be drawn in Halo, what does that say about code that cannot be drawn, and how is a drawing made easy to read without changing the language?

ABSTRACT

Having the language, we can add a general rule about how to write software. The rule is that whatever cannot be represented in a Halo diagram is not code. Any classical program can be read into a Halo diagram and written from it into any other imperative language, so that Halo stands among languages as the Rosetta Stone stands among scripts. Halo separates syntax from semantics almost perfectly: a diagram carries no linguistic tokens other than its constants, and it stays readable because people know how to read diagrams. The chapter then gives the conventions that make a diagram easy to read (named functions, numbered pins, named intermediate variables, names that say the kind of each symbol), none of which changes the language. It closes on what a drawing that compiles would change for the people who write software, and on the one part of their work that remains once the plumbing is done.

I. If It Cannot Be Drawn, It Is Not Code

Chapter 6 drew one whole program. If every program can be drawn that way, what is left over, the thing that cannot be drawn? Think of a recipe that says to add the eggs and has never said how many: a cook can follow it only by supplying something the recipe left out. If it cannot be represented in this diagram, it is not code. By "not code" I mean not code that is useful to us: something about it depends on a local language environment (the particular features of one language and its setting), and that something is not needed for what an algorithm requires in its purest state. I can import any functions I want, but if I cannot represent a function with the circle, with this kind of logic, then something about its definition is either incomplete or undefined. Such a function is not axiomatic: its definition rests on something left unstated. For the first of two such cases, suppose a function reads a variable that nothing has ever set. In a drawing, that is an input with no wire, and Chapter 6, Section IV, has said why such a drawing is incomplete. Or suppose a line of C that changes one variable twice in a single call, as `f(i++, i++)` does. The C standard leaves undefined which change happens first, and so what the line does depends on the particular compiler. What happens when? The line does not say, and so there is nothing to draw until the programmer says which change comes first.

So I am making two claims. My first claim is that Halo diagrams can represent any conceivable software that matters to us. That leaves aside quantum computing, and even a good deal of quantum computing can be described in this way. The essay *Which Programs Can Be Drawn; Halo and the Quantum Exception* takes up the question of which parts of quantum computing a Halo diagram can describe. My second claim is that any classical program (and classical programs are, in my experience, about 99.99 percent of the code written in the world) should be readable into a Halo diagram and then writable from the Halo diagram into any other language.

A Halo diagram is a blueprint for a programmer to translate into whatever language. The translation works the other way as well: I should be able to take anything written in any current language, if the code is written in the imperative model, and translate that code into a Halo diagram. And if I find that I cannot, it means there is something about that code I do not understand, because I cannot answer the question "what happens when?" for it. Either the answer is in the code and I have not found it yet, or the code leaves the answer unstated, as the two cases above do, and in both I cannot yet say what happens when. Asynchronous code is drawn with callbacks and, where a module takes several intake threads, by the module's own definition of how it works with them (Chapter 6, Section V). Code that makes code, such as an eval, generated code or a just-in-time compiler, is drawn as well: the code it makes is an instance, and an instance arrives at the zenith of a circle as a value on a data line, as the callback of Chapter 6 does. This covers literally everything I know about programming.

What I am saying as a base statement is that the notion of Turing complete maps to this diagram. An actual Turing machine, which was the basis for Turing completeness, is a cumbersome model. It has many odd details that must be managed just so, and translating to and from it is cumbersome because it is a very unnatural representation. This representation is very natural to how a programmer actually thinks. So when I am in conversation with other people and draw a diagram like this, there is no ambiguity about what the message is, and everybody in the room can translate from it to anything else very easily.

Halo is a way of representing code that deals with synchronized sequential code and asynchronous parallel code with equal facility. As a descriptive language of code as intention, it separates syntactical elements from semantic elements so that the only text left on the Halo diagram side is its constants: there are no other linguistic tokens in a diagram (Section II). All syntactic information has been parsed away, and the diagram remains recognizable and usable to people, because people know how to deal with diagrams. A Halo diagram is a wire layout: circles stand for the

program and for the functions it calls; trigger lines, drawn to small triangles, carry the flow of control from circle to circle; and data lines carry the values, which may cross the boundary of a circle where trigger lines may not (Chapters 5 and 6). It is a natural representation, with good refactoring characteristics and a high degree of fungibility, meaning that one part can be exchanged for another that does the same work. It is very abstract, however: there would be a transition period for programmers, but once in that world, even common people could work with it.

II. Labels Without Changing the Language

If every program can be drawn, then large programs will be drawn, and Chapter 6, Section III, has said what a large drawing becomes with no help: big spaghetti code. So the second half of this chapter's question follows from the first: how is a large drawing made easy to read without changing the language? Now we can talk about how to structure diagrams like this so that they make more sense, starting with a few conventions that have made programming much, much simpler. Each convention will turn out to be a rule of the normal form, applied to a drawing.

The first convention concerns routing. We can route wires point to point for everything, but in actual practice that turns out to be very hard to understand. Remember that everything here is defined by ordinal positions, by the number of each pin. Even the labels on the functions are not really labels: they are numbers coming out of a chain of operating system functions. In the memory of the operating system, every function is represented by an integer giving the memory address where the function lives, and there is no name associated with it. So when we distill a program down to wires and ordinal positions, I could literally take every word out of the drawing except the constants.

Human beings, though, tend to think in identities, and not in ordinal numbers all of the time. So the first thing we want to do is label the functions we are calling. The second thing we want to do is label the pins, but

the pins must stay ordinally numbered, because the pins represent a contract from the caller to the callee. A data pin is a column of the function's table (Chapter 1, Section III), and a column keeps its place.

Within a function, though, the definition of what the algorithm is means that we want to label the inputs, and to label the transitions as states the function is in. So, rather than running wires directly from one function to another (which maps an ordinal directly to another ordinal), we substitute intermediate variables. As a convention, that adds another symbol.

An intermediate value can be drawn, first, as an ordinary module. Say a stream of characters, bytes in sequence, is to be processed by several rings, sequential flows of function calls in a loop, and I want to keep a byte from one pass to the next. I put in its path the simplest Halo module imaginable, with one data input and one data output: its trigger goes straight through, and its data line goes straight through (Figure 7.1). All it does is remember the value it had the last time it was triggered, and between triggers it puts that same value out, constantly, on its data output. Because the module has an instance, there is a diamond at its zenith, and somewhere there is a location in memory that holds the value, provided by the operating system, because memory is a real thing and every real thing has an instance. Somewhere I connect that instance back to main memory and give it an address. The simplest module remembers a number, and so the simplest module can stand for a local variable of a program, or for a position on a stack. It is a table with one record and one field.

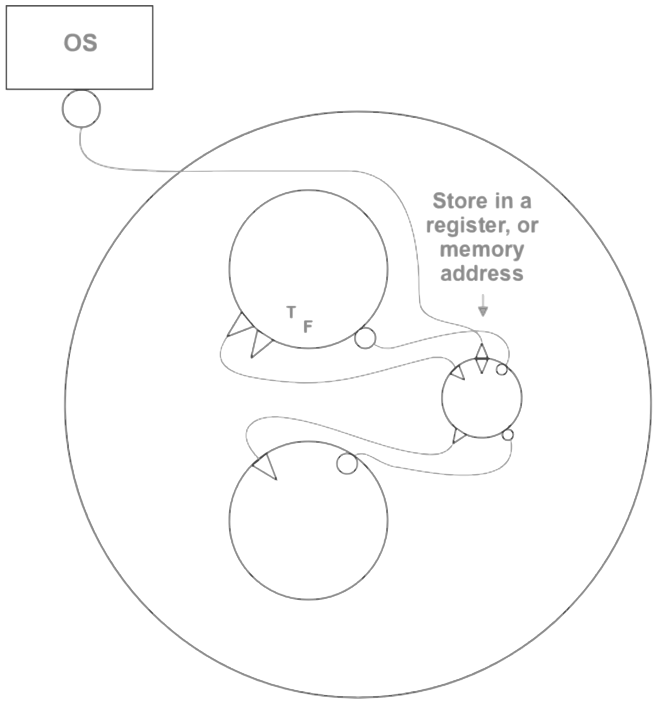


Figure 7.1. The simplest Halo module: one data input and one data output, its trigger and its data each passing straight through, holding the last value it was given. Its instance is supplied by the operating system, "Store in a register, or memory address", as the label says. It is how a local variable, or a place on a stack, is drawn.

Drawing a module for every value soon becomes cumbersome, and so the drawing gains a convenience that changes nothing in the language: the named variable. It is drawn as the lozenge of Chapter 6, a long box with rounded ends, and I simply give it a name.

In the naming convention I use, a local variable is prefixed with "fl_" followed by the name of the variable. A function input is prefixed "fi_", a function output "fo_", a module local variable "ml_" and a module input "mi_". There are not many prefixes, and Table 7.1 gives the ones the RISL conventions use most, in the code that a drawing becomes (Chapter 9).

Table 7.1. The RISL prefixes, each of which says what kind of symbol a name stands for.

Prefix	Kind of symbol
fi_	Function input.
fl_	Function local variable.
fo_	Function output, used where the distinction from an input is needed.
fn_	Function.
cl_	Class: usually a collection of functions together with the record data that they share.
op_	Operator: a special kind of function that cannot follow the RISL calling conventions fully, because its call or its callback must be compatible with an outside library or with a built-in function of the language in which it is written.
ss_	Static structure: usually a variable visible only within the definition of an object, to the member functions that make it up, and hidden from every other scope.
ml_	Module local: usually a variable defined outside every function and object and visible only within its file; also used for symbols defined in a module as part of a document, such as named tags reached through the Document Object Model of HTML or CSS.
mi_, mo_	Module input and module output: almost never needed, and used only in languages that enforce the linking of object modules.
pr_	Preprocessor definition, in languages that have a preprocessor (ANSI C, C++): a symbol meant to be interpreted at that level.
lb_	Label, in languages that have a goto or a named break or continue, and in assembly languages for a named absolute address of a jump point or of data.
ty_	Custom-defined type, as in a typedef of ANSI C.
tx_	Transmit identifier: a named token of a communications protocol, used as a linking symbol where data is serialized.

There are a great many different conventions for naming the variables of a program. I have experimented with camel case and with every convention of prefixes and suffixes. What I mostly care about, looking at a variable name, is what class of symbol it is (the reason is in the next paragraph: the type of a variable changes as code develops, and a name that

carried its type would have to change with it everywhere, while its kind of symbol does not change). If it is a function, its name begins "fn_", meaning it is a call. If it is data, I want "fi_", "fo_" or "fl_" in front of it. A preprocessor definition gets "pr_". A keyword has no prefix. A prefix keeps the rule of the normal form, because the kind of a symbol depends on the symbol alone and never changes, and so writing it in the name can never leave a second copy that disagrees with the first.

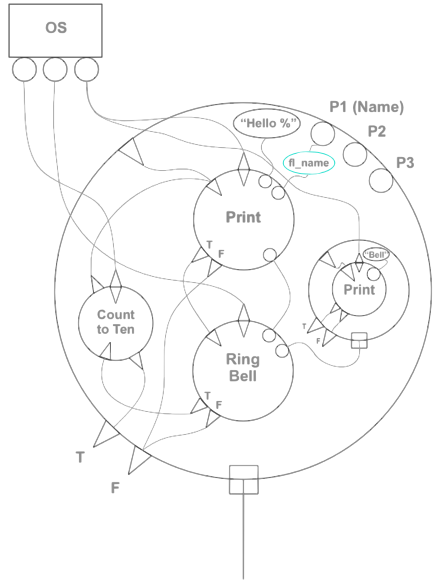
So the type of the variable is not that important to me. As a programmer I do care what the type is, but I might change a variable from an integer to a floating point number as I develop the code, and renaming it all the way through the program would be a real nuisance. I document the type in the same place where I document what the variable is for. The type is then written once, in one place, which is the normal form's rule applied to a name: a type carried in the name would be the same fact written again at every use. If the third input of a function is an indicator of the wind speed, the comment says "wind speed, meters per second", and it may add "floating point number". If I am really careful, I might say "32-bit representation, using this convention for floating point numbers". At the level of hardware I would need to care to that extent. In software, for the most part, "meters per second" is enough, since in JavaScript a number has one representation everywhere. When I name a variable, then, I mostly want to know its scope and its kind of symbol, and everything else can be documented as part of the meaning of what the variable is for.

A general convention of prefixes, across languages, for the type of a value, its layout in bits and its units would be too cumbersome, and too remote from any one language. More useful than any prefix for type is a suffix that keeps the version of a name, together with the prefix that gives its kind of symbol (Chapter 11, Section V). Once a name is chosen, only its *unicity* matters: that it names exactly one thing. A name with unicity does for a symbol what a key does for a record in a table. (Unicity is a property of a name; the uniquity of Chapter 4 is a property of an instance, and Chapter 5 joins the two: every instance is unique, and so the drawing

names each one.) Everything else about it is detail, relevant at some times and not at others, and a reader should not have to parse that detail again every time the name appears.

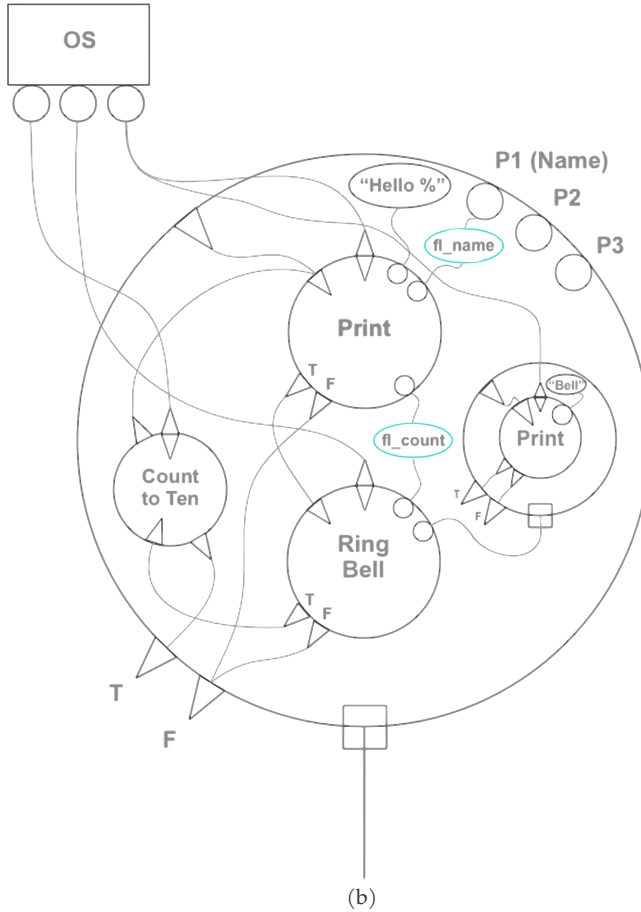
Another convention: in place of camel case, I use underscores. Remembering whether a symbol needs its first letter in upper case, or its second, or whether it means something different when one letter is upper case than when the whole word is, is a nuisance. There are so many conventions around it that after writing code for fifteen or twenty years under every convention there is, you reach the point where enough is enough. Underscores, and no upper case in the name of any symbol, ever. With one spelling for each name, a name cannot stand for two things by a difference of case, and so the spelling keeps the name's unicity. You will be surprised at how much time this saves you, and how much mental frustration.

So, for every push and every pull, I want the value to go to a named local variable, a scoped variable of some kind. That way the mapping is always clear. Every input and every output then has a fan-in and fan-out of one, while a variable can take fan-in from any number of places and fan out to any number. Now the pull and the push are explicit, every time. And each value now has one place, and that place has one name, as each fact in the normal form has one place and one key. In hello world, the first data input of the program, the name to print after hello, goes to a named variable, fl_name, and the count of characters Print writes, local to the hello world function, goes to another, fl_count (Figure 7.2).



(a)

Figure 7.2 (a). Named variables, drawn as lozenges. (a) The program's first data input, the name to print after hello, captured into fl_name, the lozenge at the upper right inside the program, below the constant "Hello %" (outlined in a light color in the drawing).



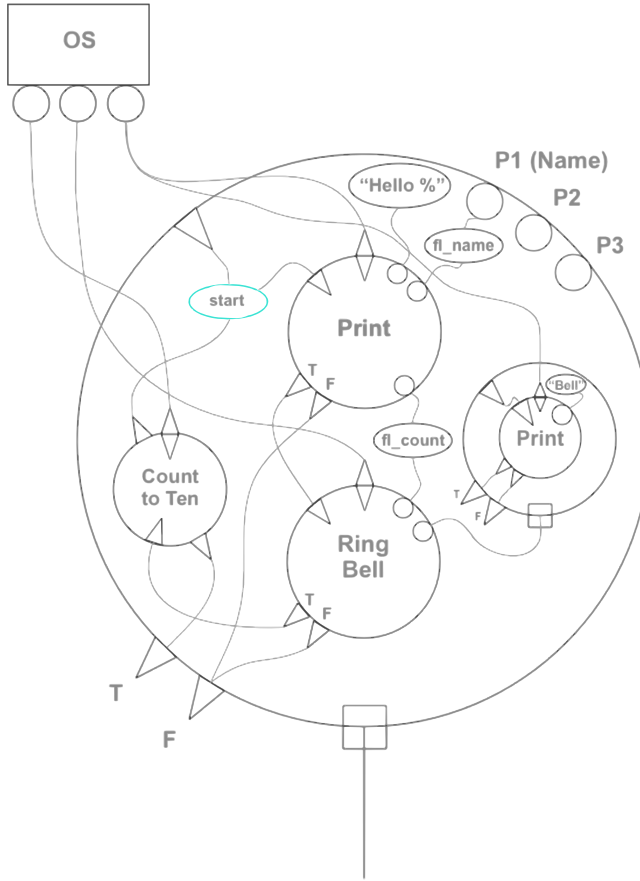
(b)

Figure 7.2 (b). A second named variable, fl_count, local to the hello world function: the lozenge at the middle of the program, between Print and the callback at the right (outlined in a light color in the drawing). It is not yet wired; Figure 8.6 wires Print's output through it to Ring Bell's input, as Figure 6.10 wired them directly.

If we are really disciplined, we do the same thing for the wiring of the trigger lines: we give the program named states. If the program is in this state, it triggers this module, and if it is in that state, it triggers that module. When this module is finished, the program goes to this state. So I can represent the whole program as a state machine. Chapter 1 has shown that a state machine is a table whose rows are its states: each named state

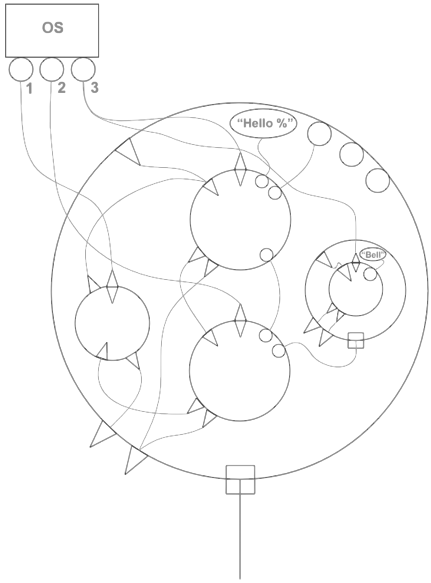
is a row, and its name says which row. For a state I use the same lozenge, with the state's name in it: the program's trigger in goes to a state called start, and start at once triggers print; when Count to Ten sends the loop around again, it also goes to start (Figure 7.3 (a)). A named state is a mnemonic, a marker that tells me where things are in the program. The lozenge now has three uses, and where it is drawn tells them apart: a lozenge on the trigger lines is a named state; a lozenge on the data lines holding a value written in quotation marks or as a numeral is a constant, a node that produces a constant value (Chapter 6, Section III); and any other lozenge on the data lines is a named variable.

The beauty of the Halo language is that it can represent an algorithm without any labels at all: no keywords, no special symbols, no special anything (Figure 7.3 (b)). Because Halo needs no labels at all, the language itself can be taught almost immediately. You draw the one diagram and say what its symbols are, and everything after that is wiring. It is that simple. Symbols, wiring and the limits on how the symbols are used are all special cases, subclasses, of the Halo language.



(a)

Figure 7.3 (a). States, and no labels at all. (a) The trigger lines given a named state, start, the lozenge at the upper left inside the program (outlined in a light color in the drawing): the program's trigger in and the loop from Count to Ten both arrive at start, which triggers Print.



(b)

Figure 7.3 (b). The same kind of program with every label taken away: no names and no keywords, only circles, pins and wires, and the algorithm is still fully drawn.

What the drawing removes is the arbitrariness of identity, and identity itself stays. A wire is still a reference: something must say that this output feeds that input. In a Halo drawing the structure fixes that identity: at a pin it is the pin's position, the ordinal that is the contract between caller and callee, and at a value it is unicity, the fact that each name stands for exactly one thing. A label added afterward is for people to find their way by, and the machine never needs it. The same choice has been made elsewhere. Moses Schönfinkel showed that logic can be carried out with no bound variables (the named placeholders inside a formula, such as the x in a function's definition) at all (Schönfinkel 1924). N. G. de Bruijn replaced the names of bound variables in the lambda calculus (a notation from logic for writing functions) with numbers that count positions (de Bruijn 1972). John Backus argued for programs built by combining functions without naming their arguments (Backus 1978).

You can think of the Halo language as a Rosetta Stone language. I can translate any imperative language into it, and I can translate from it into any imperative language. Think of a tune written out in another key and then written back: every note returns, whatever the handwriting of the first copy. The translation is *exact in both directions*, and exact means this: a drawing written out and read back is the same drawing; a text read in and written out is the same algorithm, in the canonical form of its language (the one form of each construct that the written standard for that language allows). If I translate my code into Halo and back, I get my algorithm back, written in that canonical form (Chapter 11, Section II). Code, in whatever language, that does not pass through that translation should be rewritten in its own language just enough that it can be translated into Halo. The line of C in Section I is such code: written as two lines, with its two changes in the order the programmer means, it translates. So the notion of "write once, run anywhere" is simply moved up to the meta level: once the algorithm is done, it is done.

III. A Drawing That Compiles

A program can be written this way, graphically, and then compiled from the graphic directly into any language, with no intermediary and no developer needed, but you still have to have the graphic, and the question is where the graphic comes from. If I have an existing piece of software, I can translate it into a graphic.

Drawing programs can export a drawing as an image file, such as an SVG (scalable vector graphics) file. The same menu could offer JavaScript, so that saving the drawing produced the program. The graphic can become directly executable code. Take the one call to Print in hello world. The drawing says which instance is called, which values stand on its numbered pins and where each of its two exits goes. In JavaScript written to the RISL standard, each of those has one place in the text (Appendix B, Listing B.1): the values go into an array by position, as `f[0] =`

`fl_format_1` and `f[1] = fl_name_1;` the call is `cc.osys_fn_print_1(f)` (`cc` is the namespace the whole program shares, `osys` names the operating system, whose box in Figure 6.4 supplies Print, and the ending `_1`, on this name and on `fl_format_1` and `fl_name_1`, is the version suffix of Chapter 11, Section V); a test of that call sends the fail exit to the program's exit F; and the line after it takes the output back into the named variable fl_count. That demystifies programming, and to some extent it also disintermediates it, removing the middleman. It makes it possible for anyone to be a developer, as long as they understand the diagrams. So we make the diagrams more interpretable, easier to understand, by adding the intermediate labels, and then the diagrams are much simpler. In some cases, in place of wiring the data side at all, we may simply use the intermediate variables and keep the wiring for the control flow alone. Most people, when they look at diagrams, think of flowcharts, and the control flow of a flowchart stipulates everything except the data.

You may ask what tool this book describes for writing a module out as code. The tool this book describes is the Halo metalanguage processor: it reads a module file (one file that holds a module, written in each of its languages) and writes the module in each of the languages the file holds, JavaScript, ANSI C or assembly language (Chapter 9, Section IV), and the post-processors of Chapter 11, Section II, return code written in any other way to the canonical form it reads. The file encoding of drawings, which says how a drawing itself is stored as a file so that a tool can read it, is a separate standard (Chapter 9, Section I). Appendix B shows, element by element for hello world, what such a tool carries from the drawing to the text and back.

If a drawing can be made from existing software, as the start of this section said, a programmer will want to know what may be done with that drawing. The method can certainly be used to map existing products, for anything that does not involve homomorphic encryption (a way of computing on data while it stays encrypted). You may ask whether the method also breaks the barrier of intellectual property. The notion of an

algorithm is effectively a kind of mathematics. To the degree that an idea cannot be patented but its expression can be copyrighted, you have to work out the distinctions. With copyright, if an idea is expressed in English, for example, you cannot reproduce it in that same wording. If you write the idea anew in your own expression, you are clear of it; a translation of the same expression into another language is a derivative work, and stays under the original copyright (United States Code, Title 17, Section 101). So a program translated through Halo from one language into another is, under that definition, a derivative work of the program it was translated from, and it stays under that program's copyright. As far as I know, patent law does not cover mathematical algorithms, though that is a question for a lawyer.

A drawing that compiles does demystify development work, completely. But there is a mystery left, and here it is. I have described how to represent an algorithm. Now suppose I say: here is a grab bag of ingredients, here is what I will give you as inputs, and I need you to produce these kinds of outputs. The magic of a software developer is working out how to take those components and assemble them into something that takes those inputs to those outputs reliably. Working out that assembly is a kind of meta-thinking that a software developer does. The rest of it is all plumbing.

IV. Summary

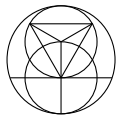
Whatever cannot be drawn as a Halo diagram is not code. Anything that cannot be drawn either depends on its local language environment or has a definition that is incomplete or undefined, and any classical program should be readable into the diagram and writable from it into any other imperative language. That makes Halo a Rosetta Stone among imperative languages, and it moves "write once, run anywhere" up one level: once the algorithm is done, it is done. Halo separates syntax from semantics almost perfectly, since a diagram carries no linguistic tokens other than its con-

stants, and people can still read it because people read diagrams. The translation is exact in both directions: a drawing written out and read back is the same drawing, and a text read in and written out is the same algorithm, in the canonical form of its language. The conventions that make a diagram easy to read change nothing in the language. Functions are named, and pins keep their numbered order as a contract between caller and callee. A named intermediate variable stands at every pull and push, and the flow of control passes through named states. A value can be held by the simplest module, which only remembers what it last carried, and a named variable or state is drawn as a lozenge; where a lozenge is drawn tells a named state, a constant and a named variable apart. Identity stays in the structure, as the position of a pin and the unicity of a name, and a label is for people. Names tell the kind of each symbol, and they are written with underscores and no upper case. A drawing that compiles would let anyone who reads the diagrams write programs. What it leaves is the developer's real work, assembling components so that given inputs become the required outputs reliably. The conventions are rules of the normal form, applied to a drawing: a value has one place and one name, a pin keeps its column, a named state is a row, and a type is written once. So what counts as code is what can be written in the one structure this book set out to find, and that structure carries into any imperative language.

Halo can draw any conceivable software that matters to us, quantum computing aside. How, then, should a whole system of such drawings be organized, so that it can be understood and changed for as long as it is in use?

KEY CONCEPTS IN THIS CHAPTER

- **Not code**: whatever cannot be represented in a Halo diagram, whose definition is incomplete or undefined.
- **Rosetta Stone language**: a language into which any imperative language can be translated and out of which any can be written.
- **Separation of syntax from semantics**: a Halo diagram carries no linguistic tokens other than its constants, since all syntactic information is parsed away.
- **Exact in both directions**: a drawing written out and read back is the same drawing, and a text read in and written out is the same algorithm, in the canonical form of its language.
- **Ordinal position**: the number of a pin, which is the contract between caller and callee.
- **Intermediate variable**: a named variable placed at every pull and push, so that the mapping from pin to name is always explicit.
- **Variable prefixes**: written in code in lower case with an underscore, each saying the kind of a symbol (Table 7.1): fi_ (function input), fl_ (function local), fo_ (function output), fn_ (function), cl_ (class), op_ (operator), ss_ (static structure), ml_ (module local), mi_ and mo_ (module input and output), pr_ (preprocessor definition), lb_ (label), ty_ (type), tx_ (transmit identifier).
- **Simplest module**: a module with one data input and one data output whose trigger and data pass straight through, and which remembers the last value it carried; it draws a local variable or a place on a stack.
- **Named state**: a lozenge on the trigger lines naming where the program is, such as start, so that a program can be drawn as a state machine.
- **Unicity**: the property of a name that it stands for exactly one thing; once a name is chosen, it is the only property of the name that matters.
- **Plumbing**: the rest of programming, once the developer has worked out how to assemble the components so that inputs become outputs reliably.



Chapter 8

Normal Forms and System Architecture

How should a whole system of Halo drawings be organized, so that it can be understood and changed for as long as it is in use?

ABSTRACT

Data structured in third normal form can represent any construction of data, and code constructed in third normal form is likewise fully general to all system design. This chapter shows what that means in practice. It gives the normal forms as a short teaching ladder (a network of data, a table, linked tables) beside the textbook definitions of Chapter 3. It shows a class as a factory, whose every call makes a new instance, and a table of those instances as the record of a system's parts, through the example of a factory floor of drilling tools and inspectors. It shows that a program in this form is a superset of a flowchart. And it shows the use of the whole discipline: code in third normal form shows where its points of articulation are, the joints at which a system can be refactored to take new functionality, and so code kept in this form never accumulates technical debt.

I. Third Normal Form at the System Level

Hello world fits on one page. Picture a whole system as a wall of such pages, thousands of them, each page a circle wired to circles on other pages. What order can be put on that wall, so that a person can find any

part of it and change it? A database that is well kept has such an order already: each fact has one place, and each link between facts is a key. I now make some broad generalizations at the level of architecture. They come from the construction of Chapter 1, and here I state them plainly.

In the same way that data structured in third normal form has been shown to represent arbitrary constructions of data, code constructed in third normal form is fully general to all system design. That code constructed in third normal form is fully general to all system design is the equivalence thesis of Chapter 1, seen from the side of code, and that generality is an exceptionally powerful thing to know. It is so because software engineering is the science of learning how to organize complexity, and what I have just described is what it means to organize complexity at the level of the system. Stated once more: in the same way that third normal form covers everything we would want to do with the relationships among data, third normal form can also describe everything we would want to do with the relationships among the modules of a system.

Ben Moseley and Peter Marks reached a related conclusion from the side of complexity: keep the essential state of a system as relations in Codd's sense (E. F. Codd, who founded the relational model of data), and write the rest as pure functions over them (Moseley and Marks 2006). The equivalence thesis carries the same normalization to the modules themselves.

II. The Normal Forms as a Teaching Ladder

To describe the code equivalent of third normal form, I first describe briefly what third normal form is in data, so that it can be recognized when it appears in code. The term is the standard one, and any reference on databases defines it, at cumbersome length, and so I give an abbreviated version. What follows is a teaching ladder, three rungs that make the purpose of normalization visible; the textbook definitions, from E. F. Codd, are stricter, and Chapter 3 gives them: first normal form requires a

single value in each field, second normal form removes dependence on part of a key, and third normal form removes dependence between fields that are not keys.

The first rung is a network of relationships. I have data elements as nodes, connected to other data elements as nodes, so that there is a huge graph of data connected to other data. It is not organized. The unorganized network is the first rung: the bare fact of what is connected to what. Codd's first normal form begins one rung higher, with the table.

The second rung is the table. In place of a random network of data, I put the data in rows and columns. I list the things, and then I list attributes for each of them; I name the attributes and sort them so that each is always in the same column. So the table is the second rung. In Codd's terms such a table, with one value in each field, is in first normal form, and it reaches second and third as its repeated facts are moved into tables of their own. But if we put all the data into the same table, we end up with a great deal of duplication. Say I have people, and I record their street addresses and characteristics such as hair color. Hair color will differ from person to person, but the street address may be the same for a whole family. If I list the same street address for every person in that family, I duplicate the address in every row (Figure 8.1 (a)).

So I set up an address table. I have a person table and an address table, and a key that links the records of the address table to the records of the person table. I assert that the key has a unique value for every address. Now I can join the two tables wherever I need information about a person's address (Figure 8.1 (b)). The person table and the address table, linked by the key, are in third normal form, the third rung of the ladder (Figure 8.1 (c)). Third normal form is a collection of tables with linked keys that bind the tables together so that the notion of a join works. Third normal form is no more than such a collection of linked tables, and third normal form is very straightforward.

NAME	ADDRESS	HAIR COLOR
John	123 Cherry Dr	Black
Sarah	125 Maple Way	Blonde
Mark	123 Cherry Dr	Red
Sally	111 Sandy St	Black

(a)

ID	NAME	HAIR COLOR
001	John	Black
002	Sarah	Blonde
003	Mark	Red
004	Sally	Black

ID	ADDRESS
001	123 Cherry Dr
002	125 Maple Way
003	123 Cherry Dr
004	111 Sandy St

(b)

ID	NAME	HAIR COLOR	ADDRESS_ID
001	John	Black	A1
002	Sarah	Blonde	A2
003	Mark	Red	A1
004	Sally	Black	A3

ID	ADDRESS
A1	123 Cherry Dr
A2	125 Maple Way
A3	111 Sandy St

(c)

Figure 8.1. From one table to third normal form. (a) One table: the address 123 Cherry Dr is written twice, once for each person who lives there. (b) The attributes split into a person table and an address table, joined by a key, ID; the address 123 Cherry Dr is still written twice, and the person table does not yet carry the key of an address. (c) Third normal form, drawn for this book: the person table carries ADDRESS_ID, the key of a record of the address table, and the address table holds each address once; John and Mark both carry A1.

Everything we normally think of as a relational database, the kind reached through SQL (Structured Query Language, the standard language of relational databases), is built to hold its data in third normal form. A particular SQL database can still hold tables that are not normalized. On the ladder, a NoSQL database (a database that stores data without the relational model's tables and joins), essentially one big table of associative relationships, effectively stands on the second rung. So does

a vector or a hash table, which can be treated as a kind of table. The data a single application keeps for itself, unorganized, stands roughly on the first.

We can use data in all three of these forms. Because data in the third normal form is the most structured, it turns out to have the most utility in many applications, and so we move data from the first and second forms into the third whenever we can. If the data is a network, it starts on the first rung; it is carried to the third as a table of its nodes and a table of its connections, each connection a record with two foreign keys. The three rungs are all there is to the ladder.

III. A Class as a Factory

What is the relationship between third normal form and code? In Chapter 1 I gave the mapping from the side of functions, and in Chapter 2 from the object-oriented side. In this section I add the factory that makes the instances, and the table that holds the instances.

Take object-oriented programming, a paradigm that has to some degree come and gone now that people think in terms of functional languages. As far as I am concerned, these paradigms have more in common than they have differences, and thinking in terms of transformations, rather than objects, is also helpful. I stay with object-oriented programming so that I can describe a relational model between modules, but what I describe holds equally for what is called functional programming. The same underlying language model serves either one, depending on whether you are talking about currying (turning a function of several arguments into a chain of functions, each taking one argument) or about instantiation.

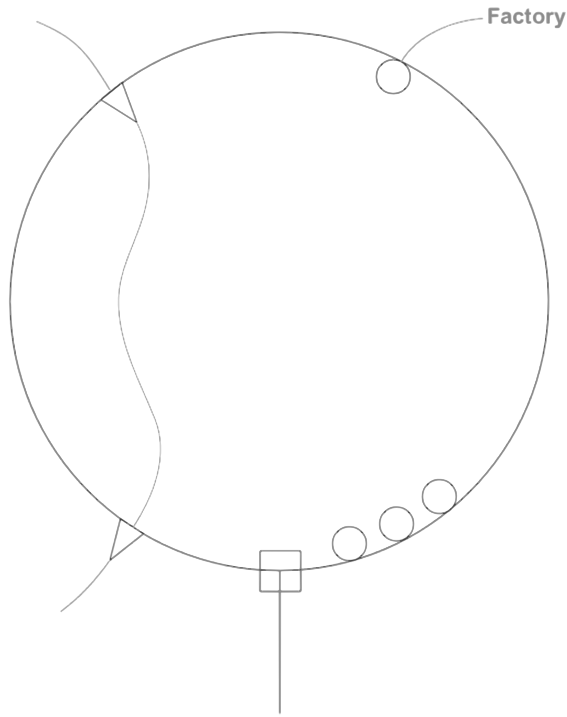


Figure 8.2. A class drawn as a factory: the outer ring, labeled Factory, with a trigger line passing through it, its square at the nadir, and its data outputs.

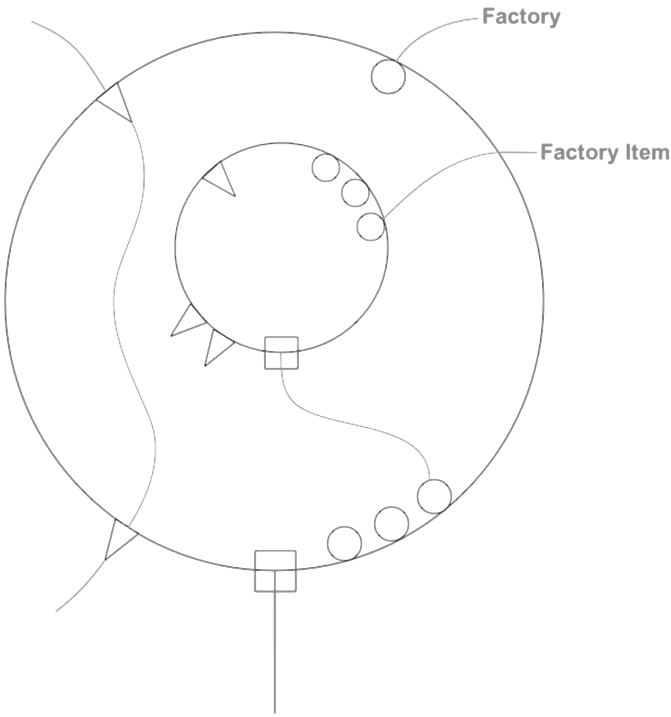


Figure 8.3. The factory with the factory item inside it. The inner ring, the item, is also a definition, with a square at its nadir, its own trigger in, data pins and exits; each call of the factory makes a new instance of the item, and a data line carries that instance out through a data output of the factory.

Call the drawing of Figures 8.2 and 8.3 a class definition. The outer ring (here, as in Chapter 6, a ring is a big circle of the drawing) is a factory, and the inner ring represents the thing the factory makes; call it the factory item. Whenever a thread of control enters the factory, it generates a new instance of something that can take a trigger input and perhaps a few input variables, and can generate a few output variables. Every call of the factory function creates a new instance of the item, and the item is itself a function that can be called. I can use such a factory to generate a callback function.

Now say a whole set of different functions is defined inside the factory and routed out as outputs. And say a collection of variables is defined at the factory's scope, at the level of the class. All of the defined functions can read and write those variables freely, including on behalf of the functions they call inside them.

What I have then is what is normally thought of as an object model. Each of the inside functions is a method. If I call the outside thing once, I get an instance, which I can treat as something to call later, and I can use its functions as methods within the class. If I call this a get-set class, I can use different trigger pins to call one of its functions, to access local variables and to assign variables to the output (each of those trigger pins is the way in of one function inside the instance, and so each function keeps the one way in of Chapter 6). Or, with a naming convention of class name and method name (each such name stands for an ordinal number, the number of a pin, as Chapter 7, Section II, gave it, and so I can treat the numbered pins as labeled entities), I can call the methods inside it and reach its internal variables. I can do that from anywhere I have access to the instance I created when I called the factory once, which is anywhere I have access to its wire. I can split that wire and make this one instance of the factory function available in many places, and the variables inside it are now specific to that instance: they are class-scoped variables.

When the instance is called, anything connected to it can reach what it exposes: any value on an output pin can simply be picked off. The methods reach the internal variables, but I have no view of those variables unless they are exposed through the output pins of the internal methods. Class-scoped variables, which a caller reaches only through the output pins of the methods, matter once we get into object-oriented or functional programming. For straightforward code, such as a program that runs the machines of a mill, we do not need anything as elaborate as class-scoped variables very often.

But I can have many instances of the class, each with its own variables, and so I can wrap the factory in another layer, a factory that makes factories. In place of routing the inner factory's output somewhere exposed, I

have a module that represents a vector, and perhaps another module that is an incrementer. I take the instances the factory makes and store them in the vector as a sequence: the incrementer gives an integer, the place where the next instance is to go, the vector stores the instance there when it is triggered, and the new instance can then be passed out (Figure 8.4). Read the drawing from the outside in: the outermost ring is the factory that makes factories, and the line into the vector comes from the square at the nadir of the ring labeled Factory. So what the vector stores is an instance of that factory, and each stored instance can go on making items of its own. So I am constructing a table of module instances, and each module has data and functions inside it. A table of module instances, each instance with data and functions inside it, is the abstraction to see. The general pattern is that I need a way of creating a new instance of a module, which has both data and the method functions that act on that data, and that instance is my record.

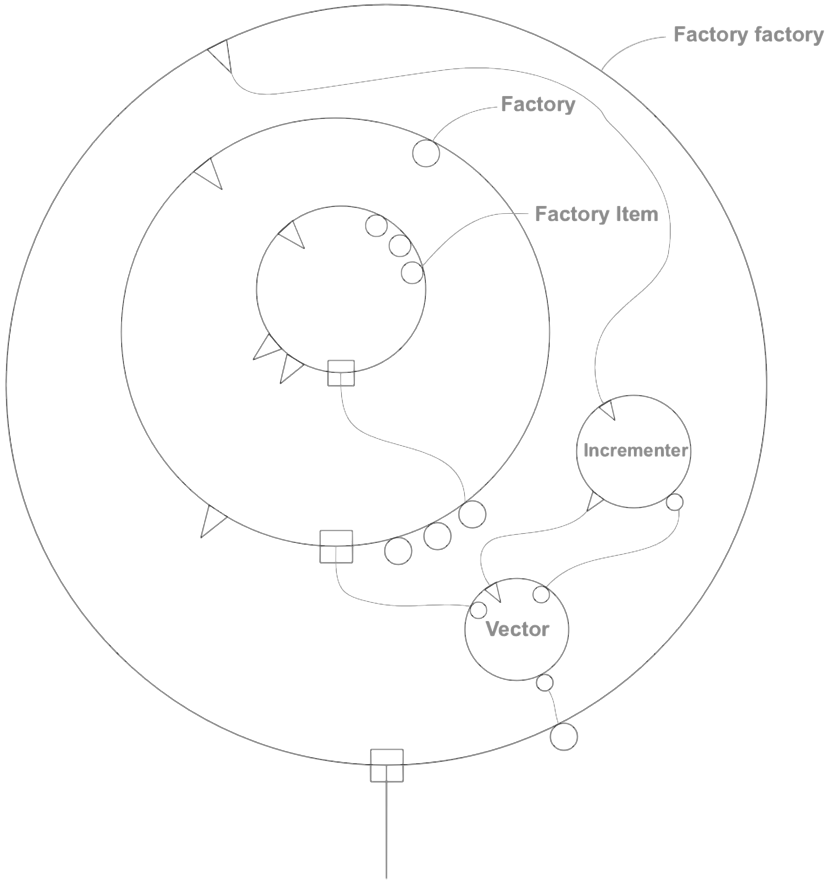


Figure 8.4. A factory factory: the factory and its item wrapped in a further ring, labeled Factory factory, with a module named Vector, which stores the instances in sequence, and a module named Incrementer, which gives the place where each is stored. Each call makes a new instance and stores it in the vector: a table of module instances.

IV. Tools That Make Tools

A factory floor makes the pattern concrete. Say I have a processing step: a machine that, when you pull its handle, cuts a hole in a lunch box. That step takes as its input a lunch box (call that the data), performs an operation, and generates output data, the lunch box with a hole in it.

Say I need to drill the hole differently depending on which kind of lunch box I am passing it. I might set up a different instance of the drill for each kind, and pass each instance a parameter that slightly adjusts the way it drills. Now I want a tool-making machine. When I call this machine, it generates a drilling machine, and each time I call it I pass it a parameter describing the variation I need. It is a tool that makes tools that perform transformations on data.

An engineering department is itself a tool that makes tools, which it deploys to the factory floor. If I have to scale up very fast, I want this tool to start putting out a great many extra drilling tools: I need a factory that makes tools. Say that halfway through the week tool number 53 fails, but it has a function in it that can calibrate itself. I need to address the fifty-third instance, call methods on it that change the state of the tool, and then, when it is done, put it back into production. The vector is important because it is a table of tools. I need to isolate the instance that matters to me, make changes, and put it back.

Sometimes there are relationships between modules of various kinds. The drilling tool might call a validation tool to make sure its input is correct: an inspection step. I have another factory, which makes inspectors, but inspection is a relatively fast operation, so one inspector can be shared among, say, five drilling tools. So I might have another table, an inspector table, and associate one inspector instance with each record of the drilling machine table, so that there are five drillers per inspector. When my fifty-first drilling machine goes down, I might call a function that detaches it from its inspector and attaches the inspector to another drilling machine that is waiting. Perhaps I do not have enough inspectors, and this way at least the inspectors are busy while that drilling machine is offline. Now I have a relationship between modules in different tables. That relationship is the foreign key of Section II: each record of the drilling machine table carries the key of one record of the inspector table, as each person in Figure 8.1 (c) carries the key of one address. If every drilling machine held its own copy of the inspection step, the same fact would be written five times

over, as the address in Figure 8.1 (a) is written twice; with the inspector table it is written once. Here, in code, is the third normal form that Section II set out to make recognizable.

V. Tables of Modules

In this section I add to the mapping the table of instances, in which the records that a factory makes are kept as the rows of a table of modules. The vector here is a table. A vector is a collection of numbered things in sequential order: this is the first, this is the second. I might as well keep them in sequential order, though there is no particular reason for any sequence: it is the order in which I created them, appending as I went.

So I have a global working environment, in the same way that I have a database, and tables within that environment. There may be relationships between modules of various kinds, and the relationships can span tables. I can split the data into subgroups, have a set of functions work on the same data, and then encapsulate them as a single record in a single table. If I need several kinds of data, I may create several kinds of table, each with its functionality in its own table.

It turns out that any module that performs a function, any module that takes inputs and generates outputs, can be treated as a table of modules of that kind holding one record: that one instance. What we think of as object-oriented programming is having many instances of any one of these functions. And we can begin to think about the relationships between functions in terms of what data they have in common.

We can also put limits on the number of data lines that can cross from the inside of a circle to the outside, which is useful too, because global variables are a nuisance. The limit is this: no data line runs from inside a circle across its membrane to the outside, and a value leaves a circle only through one of its data outputs, on the push. It stands beside the limits of Chapter 6, and Appendix A gives it among the rules of a drawing. It belongs with tables because a record keeps its private data to itself (Chapter

2, Section I), and this limit draws that privacy: the inside of a circle is reached only through what the circle exposes on its pins. This lets us think about the scope of a variable in a controlled way. I can say that any named variable defined inside a circle is local to that circle and cannot be reached from outside it. But if one circle encloses another (I have a meta-definition for a class outside, and put the variable there), the inside can read that variable. In Figure 8.5 the outer variables of module A, outer_1 to outer_3, are passed to the inner variables of module B, inner_1 to inner_3: the variables of A can be reached inside B when they are passed in, and the inner variables of B can never be reached from A. Passed in means wired to a data input of the inner circle, as Figure 8.5 draws it, so that the inner circle pulls the value when its trigger enters; a line that crossed the inner circle's membrane without meeting a pin would carry a global value (Chapter 6, Section II). So now I have class variables and function-scoped variables, and I can do, as a convenience, what is essentially a version of object-oriented programming in this model.

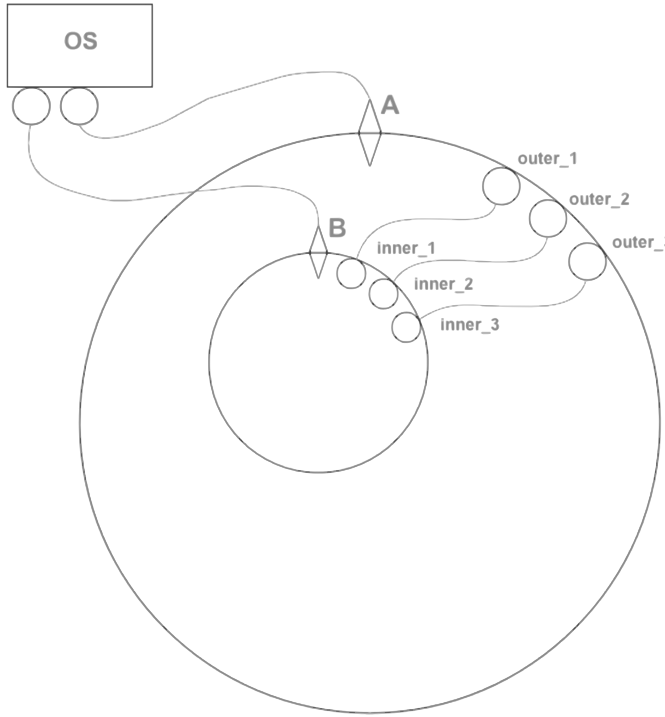


Figure 8.5. Scope: module A, whose instance comes from the operating system, encloses module B. A's variables, outer_1 to outer_3, are passed into B's, inner_1 to inner_3; B's inner variables are not reachable from A.

You can see that this is a superset of the flowchart. Instead of a separate symbol for each of the flowchart's kinds of step, everything is essentially what a flowchart's diamond would be. Its yes and no are generalized to any number of branches, and then, by the convention of success and fail, turned back into yes and no. And instead of boxes, diamonds and triangles, everything is a circle. So it is very close to a flowchart. The other part of it is that it represents the actual data being passed: which functions take which inputs and generate which outputs, where the data comes from and where it goes, on the pull and then on the push. With that, you have everything you need to draw any one module of a system: its flow of control and its flow of data.

There are disciplines beyond this that must be followed for the merit (the practical worth) of software engineering to be earned. What is shown so far is the merit that lets us get from any programming language to any other. The next question is how to organize the complexity of these algorithms. How do we split an entire wiring diagram, one that is global, into components, modules that can be reused and that a person can understand as components doing something that means something to a human being? It is just like ordinary database organization, except that in place of data in tables, I think of functions in tables, and the data is incidental. The factory floor of Section IV shows the split at a small size. Drawn as one global diagram, every drilling machine and every inspector would be wired together on one page. Split by the kinds of thing the floor holds, the page becomes a drilling module, an inspector module, a table of instances for each and a key between the two tables, and each component now does something that means something to a human being: it drills, or it inspects.

VI. Points of Articulation

Entire systems can be built this way; in fact, all systems can be translated into this form. When you do that, you have organized the complexity in a useful way. It gives a way of thinking about refactoring the whole code base, in terms of points of articulation.

In the space of all possible designs, a *point of articulation* is a place where we can refactor, such that the result of the refactoring makes it possible to change the code so that it gains functionality. A much simpler way to say it: my arm has a point of articulation at the elbow. Because there is a joint in my arm at that position, my arm has the functionality of bending there. If there were another point of articulation halfway along my forearm, I could bend my forearm there, but I cannot, because there is no joint there. So when we think about adding functionality to a system, we find

the position where we would need a point of articulation, and once it is found, we identify the refactoring that the overall geometry of the third normal form would need.

David Parnas gave the classic rule for where the boundary of a module belongs: put it where it keeps a likely change inside one module (Parnas 1972). Michael Feathers calls a place where the behavior of code can be changed without editing it in place a seam (Feathers 2004). A point of articulation is that joint, and code in third normal form shows where it is.

When code is in third normal form, finding the point of articulation is relatively easy. When it is not, it is anybody's guess whether you will be able to create a point of articulation that lets you add the functionality. Having code in third normal form is so important because the points of articulation are then easy to find. Third normal form gives you the flexibility to update the system from time to time and to add new functionality. You cannot think about adding new functionality in a disciplined way unless the code base, as a whole, already has this discipline built in. Kept in this form, code never takes on *technical debt* (the accumulated cost of changes made where the design did not allow for them) in the first place. What happens, then, when the requirements change? A business that once shipped every order to one address and now ships an order to several is one such change: the address leaves the order's table for a table of its own, keyed to the order, once. The tables show the programmer where to cut: the new requirement names the fact that must now vary on its own, the column that holds that fact is the joint, and the key of the order says what the new table hangs from. The cost of change does not accumulate: every change has its cost, and that cost is paid once, at the joint where the change is made. When the world the system represents changes where it divides, the code is refactored at the new joint, which the normal form makes findable; the cost is paid once, at the change, and does not accumulate, and so it is not debt.

The same walk holds for a system of modules. Go back to the factory floor of Section IV, and suppose the inspection step must now use a new gauge. The inspection step is written once, in the module of the inspector

table, and so the change is made once, in that module, and every drilling machine that carries the key of an inspector gets the new inspection. Now suppose a drilling machine must be inspected by two inspectors. The key of the inspector leaves the drilling machine's record for a table of its own that joins each machine to each of its inspectors, once, and the inspector module does not change at all. The key between the two tables was the joint, and it was in plain sight because the tables put it there.

The same holds when the world did not change and its joints were misread at the start. Say a schema assumed that each account belongs to one customer, and stored the customer's name and address in the account's table. When it is found that an account can be held by several customers, the customers leave the account's table for a table of their own, and a table of holdings joins each account to each of its customers, once. Third normal form is defined by the true dependencies among the facts, and so a design built on a misread dependency was not in third normal form of that world. Finding the misread is learning where the world divides, and the correction is the same once-paid refactoring, at a joint the normal form makes findable. The cost is paid at that correction, and it does not accumulate.

For anyone who has been in the software industry for a while, this is extraordinary: code in third normal form shows where its joints are, and each change is paid for once, at its joint. It is probably the single most important insight about large-scale system design that it is possible to know.

VII. Iteration and the Bounded Loop

You may ask whether this discipline of third normal form is a further layer, the next step after object orientation and functional programming. The two paradigms are two readings of one model, and this discipline organizes whichever reading is used. We are thinking about transformations. Look again at Figure 8.3. Read one way, the outer ring is a class, and each call makes an object: an instance holding its data and its methods. Read

the other way, the outer ring is a function that returns a function, and the values the first call took in are held for the later calls. The drawing is the same under both readings, and each reading can be turned into the other. There is object-oriented programming and there is functional programming, and what we have just shown is that there is a strong isomorphism (a translation exact in both directions) between them. I can think of one as generating a function that I then call in another function, and so I can create situations of currying.

This model can be used to think about recursion, but it works much better if we think in terms of iteration. From a mathematical point of view it is easier to define structures by recursion, because recursion is a fully abstract concept. But on actual hardware, instances matter. You cannot assume an unlimited expansion of stack positions for recursion. You have to have a limit, and so you think in terms of instances, which means that you are actually thinking in terms of iteration. This also settles what happens when a trigger arrives at a circle that is still running, as in recursion. Under the limit of one destination for each trigger leaving a call (Chapter 6), control follows a single path, and so a trigger can reach a running circle only through a call made from inside it. That call names its instance at the zenith, and the instance it names is a new one, taken from the table of that function's instances; the depth of the recursion is then the number of instances in use, and the table's ceiling bounds it as a counter bounds a loop (Chapter 3, Section VI). Iteration has a way of defining how many times you pass through a loop, and of ending the loop if you have gone through it more than that number of times.

So if I want to prove that a particular algorithm halts, I put a counter on its loop, and when the counter goes above a certain known number, I exit the loop. I can handle the halting problem in this way if I follow the discipline of capturing every possible loop, which is why single entry and single exit matter (here single exit means that a call, which still has its two exits, success and fail, leaves by one of them, and that each trigger leaving a call has one destination, the limit of Chapter 6). Under that discipline every loop path can be identified, and a limit can be put on every loop

path. The discipline makes every program written under it halt by construction, because every loop has a fixed limit. It does not decide halting for programs in general, which Turing showed cannot be done: the *halting problem*, whether a given program will ever stop on a given input, has no solution that works for every program. The general language of Chapter 5 remains Turing complete; the discipline is a limit laid over it, and the programs written under the discipline are a part of the language in which halting is guaranteed, which is the part a system built to run for years needs. Count to Ten, in hello world, is such a counter (Figure 8.6): when it stops the program, the program may not end in an organized way, but it ends in a known way.

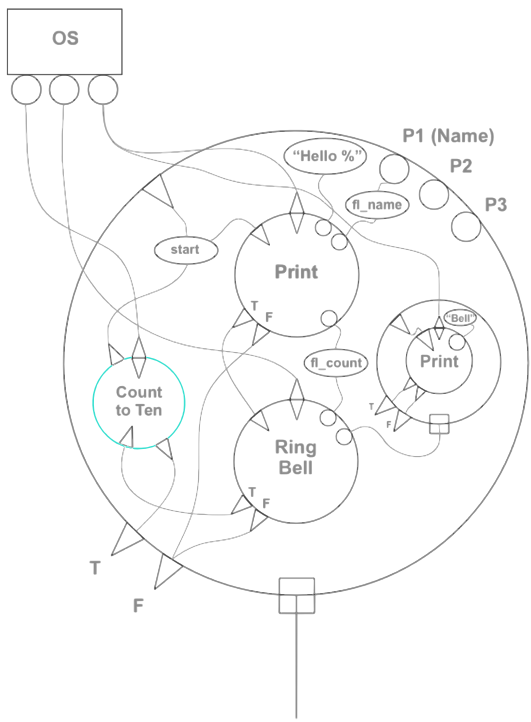


Figure 8.6. Hello world with its conventions in place: the named variables fl_name and fl_count, the named state start, and the counter Count to Ten, the circle at the left inside the program (outlined in a light color in the drawing), whose fixed limit makes the program end in a known way.

In a great many ways, what we arrive at is this: there are isomorphisms among many ways of thinking about things, but some isomorphisms are much more useful than others. The key thing is to understand the general class of isomorphisms that gives us the general class of all possible functionality. The one this book follows is the mapping of Chapter 1: every module a table, every instance a record and every reference a foreign key. Organize your code in third normal form, because then you can find the places to make changes in a relatively sane way. So this is not algorithm design. Organizing code in third normal form is system design.

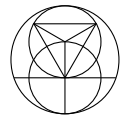
VIII. Summary

Code constructed in third normal form is fully general to all system design, as data in third normal form can represent any construction of data; that is the equivalence thesis of Chapter 1 seen from the side of code. The normal forms can be taught as a ladder (a network, a table, linked tables joined by keys), beside the stricter textbook definitions of Chapter 3. Data is moved up the ladder to the third rung wherever it can be, because the most structured form has the most use. In code, a class is a factory: each call makes an instance holding its own data and the methods that act on it. A vector of those instances is a table of the system's parts, through which any one part can be found, changed and put back while the rest keep working. A Halo program is a superset of the flowchart: every step is a circle, every branch a success or a fail, and the data the flowchart leaves out is drawn as well. The use of the discipline is that it shows the points of articulation, the joints at which a system can be refactored to take new functionality; code kept in third normal form never takes on technical debt. And because every loop can be found and bounded, every program written under the discipline halts by construction.

The structure of a system is now in hand. How is it carried into everyday code, so that a running system can be changed while it runs?

KEY CONCEPTS IN THIS CHAPTER

- **Third normal form in code**: code constructed so that every module is a record in a table of modules of its kind, linked by keys; fully general to all system design.
- **Teaching ladder of normal forms**: a network of data, a table, linked tables joined by keys; a picture of what normalization does, beside the textbook definitions.
- **Factory**: a class drawn as an outer ring whose every call makes a new instance of the item inside it.
- **Table of instances**: a vector of the instances a factory has made, through which each can be addressed, changed and returned.
- **Factory factory**: a factory that makes factories and stores each instance it makes in a vector, at a place an incrementer gives, so that the instances form a table of modules.
- **Point of articulation**: a place in a design where a refactoring makes it possible to add functionality, as a joint in the arm makes it possible to bend there.
- **Technical debt**: the accumulated cost of changes made where the design did not allow for them.
- **Bounded loop**: a loop with a counter and a fixed limit; with single entry and single exit, every loop can be found and bounded, so every program written this way halts.



Chapter 9

Modules as Tables; The Call Triple and Hot Patching

How is the Halo model carried into everyday code, so that a running system can be changed while it runs?

ABSTRACT

This chapter shows how the Halo model is carried into everyday code, so that a running system can be changed while it runs. Its account of hot patching in ANSI C, in Section IV, is written for readers who program in C, and its Summary states the chapter's results for every reader. System design, followed with this discipline, turns out to be a little easier than algorithm design. Each language receives Halo through its own conventions, the RISL conventions, which include idioms chosen for how a human brain parses them. A module is a table with a record part and a table part. A call names three things (which table, which record, which method in that record), and because every call finds its method through the table, a module can be reloaded while the system runs and every later call that finds its method through the table reaches the new code. Reloading a module while the system runs is hot patching, and code in third normal form makes hot patching orderly, even in ANSI C.

I. Conventions for Each Language

Why would anyone need to change a program that is already running? Suppose a web server is in use by a thousand people at this moment, and one function in it has a fault. The repair is three lines long. If the only way to make those three lines take effect is to stop the server, all thousand people pay for the repair of one function. Section IV gives a case in which a stop of that kind was out of the question. Before that, we need to see how the model of Chapter 8 reaches the code of an everyday language.

It turns out that system design is a little easier than algorithm design, if you follow the discipline. Think of the difference between inventing a new way to sort a list and laying out a warehouse in which every shelf holds one kind of thing. Each new algorithm has to find its own way through its own problem. In a system built with this discipline, the question "where does this go?" already has its answer: each thing goes in the one table where third normal form puts it. Everything in this chapter is that second kind of work.

A canonical version of the map from a Halo diagram to a given language exists, and it is a set of conventions, the RISL conventions. RISL stands for Reduced Interaction Structuring of Language. RISL is effectively Halo implemented in code across each language: the standard that carries Halo into practice, and the bootstrap to working in third normal form. For each language, the RISL conventions give the translation of the concepts of this book into that language, together with the idioms that a trained eye reads, and as long as the RISL conventions are used, code can be translated from any language to any other. An idiom is a shape on the page that the eye takes in whole, the way a reader takes in the word "because" without spelling it. Chapter 11 shows one: in JavaScript written to these conventions, every call that can fail has the same shape, if (! name(f)) followed by the return that reports the failure, so that one look takes in the call, its failure and where the failure goes. The standard is set out in full, with its reasons, in *The RISL Standard; Halo Written as Code, Across Languages*: its first chapter gives the map from a drawing to each

language, and its later chapters give the standards for C and for JavaScript. The file encoding of a drawing is a separate standard again, a colleague to Halo and to RISL: it stands beside both and is part of neither.

One language shows what keeping to the conventions does. In JavaScript, avoiding the language's sugar (Chapter 5, Section I) gives code that translates directly into ANSI C, and pre-allocating each table to a known ceiling (Chapter 3, Section VI) gives the effect of memory management without its cost. Both habits leave a JavaScript program in a shape that another language can carry, and translation from one language to another asks for that shape.

A drawing is saved as a file under that standard for the file encoding of drawings, which is separate from Halo. Two versions of a drawing, mine and someone else's, are compared and merged by their structure, by the syntax tree and not line by line, which is the semantic merging that *Keeping Software Soft; Explicit Code, Safe Change, and the Discipline of Subtraction* gives in its Chapter 4, on code collaboration and semantic merging (Chapter 12, Section V).

II. The Module as a Table

In Chapter 8 I gave the factory and the table of its instances; in this section I divide the module itself into a record part and a table part. The first thing we know is that the individual functions are organized into classes. The classes make collections of functions, bound with data, into a single record. Then we need one more thing: something to keep track of the table of those instances. These things themselves live in an overall system space, so that they can reach one another, and that, in effect, is the program as a totality.

A program, or an application system, then consists of a collection of however many modules (Figure 9.1). Here I use "system", "app" and "program" more or less interchangeably, for the global unit of executable information.

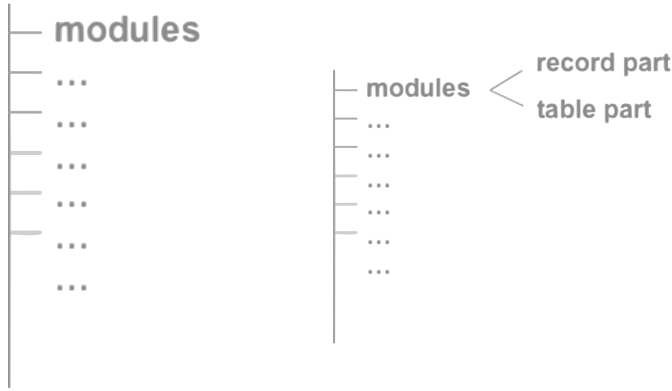
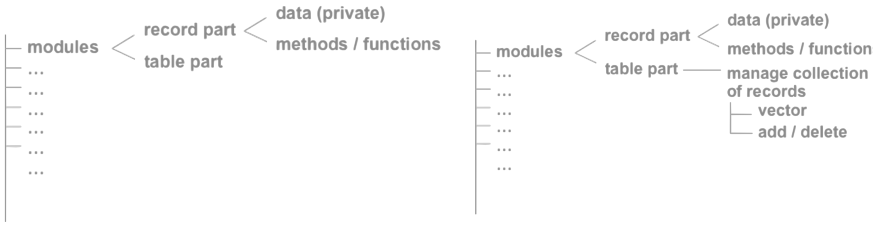


Figure 9.1. Modules. (a) A program as a collection of modules. (b) Each module has a record part and a table part.

I think of a module as being like a table in a database. A module has a record part and a table part (Figures 9.1 and 9.2). The record part consists of data, usually private (I cannot reach the data directly), and of methods: the input and output functions that let me access, manipulate or change the table. A record part of private data and methods is, at this point, fairly classical object-oriented programming. So a module has data and methods. The table part is just something that manages a collection of records. It is essentially a data structure like a vector, together with create and destroy (add and delete). If I need a particular entry, I call the table part and say "give me entry such and such", and it does. If I say "create a new entry", it adds the entry to the bottom of the table and gives me an index to it. You can think of the table part as roughly the equivalent of what pointers are in ANSI C.



(a) (b)

Figure 9.2. The two parts of a module. (a) The record part: its data, private, and its methods, the functions through which the data is reached. (b) The table part: it manages a collection of records, through a vector and the operations to add and to delete.

You can also think of the table part as what I called the factory in Chapter 8. The factory makes records, and it also stores them and keeps track of them. In a browser environment a module is a single included file, and in the Node environment it is a single file. For convenience we treat modules as files. And "convenience" here means a great deal, because in Node.js I have written a system-level function that, every time I change the source code of a file, automatically reloads that file into the running space.

III. The Call Triple

In a call in this framework, I have a record number, a function number, and the parameters of that function. Which table the record is in is implicit in the record number. So a call actually consists of three things: which table, which record, which method in that record. That triple is the *call*

triple (Figure 9.3). It gives me the absolute address. And if I update the module, then when I call that function (because I got the instance of the function through the table), I reach whatever is the updated one.

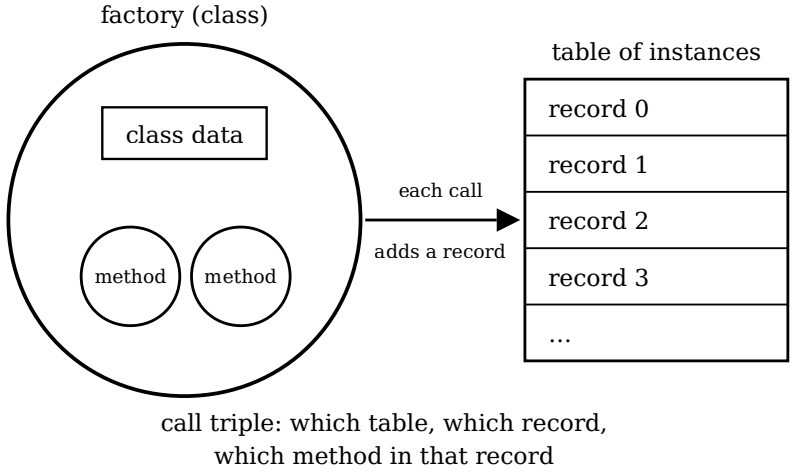


Figure 9.3. The factory and the table of its instances. Each call to the factory makes a new record holding its own data and methods; the table keeps the records in order. A call names three things, the call triple: which table, which record, which method in that record.

Of everything in the language, the single most important part is the call. How we structure the call, and how we represent it in the language, is the whole matter.

In JavaScript, for example, a call is a dereference of the table I mean to reach, a dereference of the record I need, a dereference of the function I need, and then the actual invocation. JavaScript invokes through a pointer to the function anyway, so I add the dereferences, and they guarantee that I have the updated code every single time. On the page it looks like this. Suppose a table of drills is held under the name com.company.project.drills, and I want the cut method of record 3. The call is written com.company.project.drills[3].cut(f). Read from the left: com.company.project.drills finds the table, [3] finds the record, cut finds the meth-

od, and (f) makes the call. Each of those steps is a fresh lookup when the line runs, so if the drill module was reloaded a moment ago, the lookup finds the new function. The array f is the one parameter of every function, and I show why in Chapter 11.

In some cases I can cache the reference, if I am looking for performance. The cost is that if I cache it and the table is updated in the meantime, the cached reference points to the old version of the function, which can sometimes cause problems. If I change the data structure, it will; if I do not change the data structures, it may not. But as a developer making changes, I can keep track of it. I can say: if this module caches references, I will send that module a signal to drop all its cached references and pull fresh ones. Suppose a module that writes reports looks up the cut method of the drill table once, keeps the address, and uses that address for every line of every report. Now I reload the drill module. The table holds the new cut, and the report module still holds the old one. If the new version changed only what cut does, the old function goes on working on the same records. If the new version changed the layout of the record, the old function is reading a record it no longer understands. The signal closes that gap: the report module drops the old address and looks up the new one.

For anything that caches, I create a method on the table, and I say: here is the dependency, the linkage tree (the tree of which modules cache references to which others); propagate the linkage tree so that the caches are refreshed. Each module that caches a reference states that dependency through that method, and those statements taken together are the tree. With an idiom such as the linkage tree, I can now ask and answer questions about which modules cache references to which others. Had I not used third normal form, I would not even have them as a possibility. In the case of the report module, the new address is in one place, the drill table, and the table can name each module that keeps a copy. Without that one place, the address would be written into every module that uses it, and nothing in the system could say where the copies are. Code in third

normal form makes hot patching actually make sense, and lets it be deployed in an organized, consistent and coherent way. I cannot tell you the number of times it has saved me in an emergency.

IV. Hot Patching

Code in third normal form makes something very powerful possible. Since everything is a file (one module to a file), and the system can reload a module when its source code changes, I can do *hot patching*: changing the code of a running system without stopping it.

Many languages give you a read-eval-print loop (a REPL: an interactive prompt that runs one statement at a time and prints its result). You can interact with the language, but you interact at the level of the statement. Here, we interact at the level of the module. When I save a module, I save a cluster of changes that is expected to be executable within whatever environment is running at that moment. The change propagates out through the whole system.

I also keep each module's version notes, design notes, log sections and abstract within the file itself, under the ITAG convention of the RISL standard (four upper case letters and a colon, set against the left edge of a file, mark its version notes, design notes, logs and abstract, and the last tag of every file is ENDF); *The ITAG and EGS Standards; Structured Plain Text for Files and Notes* sets out that layout in full, in its Chapter 1, and sets out EGS (Extended Grammatical Structuring) in full, in its Chapter 2. EGS is a standard for writing notes, and EGS is a substandard of RISL. In EGS, each statement begins with a dash and ends with a period, opens with 'where', 'that' or 'as', and takes the context of the statement from the indentation of the statement. EGS is named here because the design notes kept in a file are notes, and this is the standard for writing them. Think of a file headed by tags such as ABST:, above its abstract, and LOGS:, above its log, and ending with the tag ENDF: on its last line. Think also of a machine that carries its service log in a pouch on its own frame: whoever

opens the machine has the record in hand. Keeping the record in the file makes sense when hot patching anyway, and it matters for a further reason: a module may be written in two or three languages at once. One file can hold its JavaScript, its ANSI C and its assembly language, each marked as such, because a module is a file that the Halo metalanguage processor reads and writes in each of these forms. Think of a product manual printed in three languages between one pair of covers: one product, three texts, each headed by its language.

In some frameworks, some kinds of change to the data require the server to be restarted. This system does not require that restart, because the system is *idempotent* all the way down: an operation is idempotent when doing it many times gives the same result as doing it once. That means I can shut the server down at any moment and all the data is already flushed to disk, so that when I start the system up again it comes up in the same state, and starting the system, once or many times, from what is on disk always gives the same state. From the user's point of view, in a web server application, the client may not even know that the server has been restarted. Part of the reason for working this way is that it lets me make changes like these.

Data is managed separately from code. The table module usually does not change much: it creates instances, stores them and provides access to them. For the most part the table modules are the same everywhere, and the record modules are unique to whatever functionality is needed. Suppose the drill module holds three records, whose counts of holes cut are 4, 9 and 12, and I want to change how cut works. So if I need to change the design of a record module, I flush all of its data to the disk, change the code, and load the new module in. The table module then loads the data and binds it into the records that are already there. The three records still say 4, 9 and 12, and the next call to cut runs the new code. I end up with changed code and unchanged data. If the data itself has to be different, the table module calls a special function that adjusts the data, so that the

data is changed at the moment it is loaded. Suppose the new design also records the size of each drill: that function writes a size into each record as the record is loaded.

A module is unavailable for the brief moment in which its new code is loaded. Usually, while the system loads a new module, it freezes the rest of the application, so there is a pause. A request sent at that moment simply takes a little longer, roughly as long as a garbage collection event: very brief. The time to parse the module again is small; it depends on the module's size and complexity, on what it touches and on how many records there are. You pay for it somewhere. But that is why single threaded programming is so handy: I know it is a situation where the world stops. I stop the world, I make the change, and the world picks up again.

There were times, in work I did at the Pentagon, when I had to change a system that was already running, with no time to shut it down. We needed an answer in the next ten minutes, and the feature had to be fixed right then. Shutting the system down and restarting it took three hours, and it could not be done, because it was a distributed network of servers running all over the world, across all of those departments. Managing a global shutdown and restart to distribute code and recompile was out of the question. The risk of the shutdown is huge compared with the risk of the hot patch, because the hot patch is a local change and not a global one.

I implement this routinely now. In fact, if a language will not let me hot patch, I fix the language. You can also hot patch in ANSI C. It is delicate, and you have to really know what you are doing, but it is possible. You set the memory pages to be writable and not executable, make the changes, and make sure the code does not get any bigger, because it might overwrite the next function. Then you set the pages back to executable and not writable, and, on a processor with a separate cache for instructions, you flush that cache, so that the processor does not go on running the old code. If it is bigger, you move it to another piece of memory large enough to hold it, make sure all the pointer references are right, and make sure nobody is caching it. That last turns out to be the really hard part. So you change the way you write C code so that these things become sensible

again: the same scheme of dereferencing, and the same scheme of third normal form. If you implement third normal form in ANSI C, all of this becomes easy to do. In a C program kept in third normal form, every call reaches a function through a table entry, so the address of each function is held in one place (Listing 9.1). The really hard part was knowing that nobody is caching the old address, and here that is known: the table holds the address, and any module that keeps a copy has said so, in the linkage tree of Section III. A new version that is bigger goes into its own piece of memory, and the patch writes one address into the table.

```
// the type of a method: takes the call array, returns 1 or 0
typedef int (*ty_method)(void **f);

// module local: the one place that holds the method's address
static ty_method ml_cut;

// the call: through that place, every time
fl_status = ml_cut(f);

// the patch: the new code is in its own memory; one address
changes
ml_cut = fl_new_code;
```

Listing 9.1. Hot patching in ANSI C, four statements taken from a module kept in third normal form: the type of a method, the table entry that alone holds the method's address, a call that reaches the method through that entry, and the patch, which writes one new address into the entry.

V. Summary

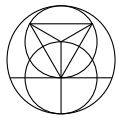
Each language receives Halo through its RISL conventions (Reduced Interaction Structuring of Language), which translate Halo's concepts into the language and choose idioms that a human brain parses easily, so that code kept to them can be translated from any language to any other. In JavaScript, avoiding the language's sugar gives code that translates directly into ANSI C, and pre-allocating the tables gives the effect of memory management without its cost. A module is a table with two parts: a record

part, holding private data and the methods that act on it, and a table part, which creates, stores, finds and deletes records and is the factory of Chapter 8. A call names three things, which table, which record, which method in that record, and a call made through the table always reaches the current code; a module that caches references is signaled to refresh them. That makes hot patching orderly: a module saved while the system runs is reloaded, its data is flushed and bound again to the new code, and the world stops only for a moment. A hot patch is a local change, where a restart is a global one, and code in third normal form can be hot patched even in ANSI C; a language that does not allow it can be fixed so that it does. Each file keeps its own record, in ITAG sections ending with ENDF, and one file can hold a module in several languages at once.

A system that can be changed while it runs still has to be changed by someone, with tools. What keeps a code base changeable for decades, and what destroys that?

KEY CONCEPTS IN THIS CHAPTER

- **RISL conventions**: the Reduced Interaction Structuring of Language, my coding standard, which sets out for each language how my designs are written in that language and which idioms to use; for Halo, the translation of its concepts into each language, with idioms chosen so that the human brain can parse them. Code kept to the conventions can be translated from any language to any other.
- **Idiom**: a visual pattern that a developer comes to recognize as a pattern.
- **Syntactic sugar**: notation added to a language for convenience that adds no power.
- **Record part and table part**: the two parts of a module; the record holds data and methods, the table creates, stores and finds records.
- **Call triple**: the three things a call names: which table, which record, which method in that record.
- **Hot patching**: changing the code of a running system without stopping it.
- **ITAG convention**: four upper case letters and a colon set against the left edge of a file, as a marker for version notes, design notes, logs and abstracts kept within the file; the last tag of every file is ENDF. It is the RISL layout of a file.
- **EGS (Extended Grammatical Structuring)**: my standard for writing notes, in which each statement begins with a dash and ends with a period, opens with where, that or as, and takes its context from its indentation; a substandard of RISL.
- **Idempotent all the way down**: a system whose data is always flushed to disk, so that a restart at any moment brings it back in the same state, and starting it once or many times from what is on disk always gives the same state.



Chapter 10

Code That Lasts; Refactoring Tools and the Cost of Sugar

What keeps a code base changeable for decades, and what destroys that?

ABSTRACT

A language is useful only to the degree that a person can read it after it has been written and a refactoring tool can rewrite it after it has been written. This chapter shows what follows from that rule. A change to Java that needed a second file to know the type of a symbol broke the refactoring tools written for the language, and so the code worth keeping stayed on the older standard, where the tools still worked. If you cannot refactor, you cannot have something that lasts, because the needs of the world evolve and software that cannot evolve with them fails. Syntactic sugar is waste: every wish for it comes from an expectation that is not valid (in my experience, about nine times in ten), or from a language that made hard what should be easy. Libraries in any language are circles with pins, and translating each into Halo and out again removes the need for any master language. The reason for all of it is the horizon: software meant to evolve for a thousand years, through catastrophe, on hardware far weaker than ours or not yet invented, must depend on almost nothing.

I. Hot Patching in Java

Does the pattern of Chapter 9 depend on one language, and does it hold up over years of use? Java answers both. The same design pattern works in Java. I have run Java that way for fifteen years. I have a way of reloading a class that has already been compiled, linking the class in, and having the change take effect with no interruption at all. That took a little while. Java does not like reflection methods being used for hot patching, and it is just barely possible. Hot patching of this kind works as far back as Java 1.4. What could end a run of that length? A change to the language underneath it could, and the next section gives that change.

II. The Change That Broke the Tools

When Java went from version 1.4 to version 1.5, its makers did what I consider one of the worst decisions ever made in the world of software. In version 1.4 and every earlier version of Java, I can take a single file by itself, analyze it, and know the type of every symbol in the file just by looking at that file. In version 1.5 they introduced a feature called boxing, by which a symbol can have a type that I can find out only by looking at another file. I have to parse another file to find out what the type is. The smallest case is one line. A file Foo.java holds the call bar.put(5);, where bar is of a class Bar declared in another file. Under Java 1.4 that line compiles only if Bar has a method put that takes an int or a wider number type, and in every case the 5 reaches put as a number and never as an object (Gosling and others 2000, Section 5.3), and so a tool reading Foo.java alone knows that the call makes no object. Under Java 1.5 it compiles as well when Bar's only put takes an Integer, and then the 5 is converted at the call into an Integer object; which of the two the call does, and so which method it reaches when Bar declares more than one, is written only in Bar.java. Version 1.5 of Java, released in 2004 under the name J2SE 5.0

(J2SE: Java 2 Platform, Standard Edition), introduced autoboxing together with generics, the enhanced for loop, annotations and static imports (Sun Microsystems 2004). Of those five features, this chapter follows boxing.

That meant that every idempotent refactoring tool (a tool that gives the same result run many times as run once) written for the Java platform was now broken. They made a very minor improvement to the convenience of the language, one tiny bit of sugar, and they destroyed every refactoring tool written up to that point. All of them had to be rewritten, and some of them could not be, because boxing required a parse of more than one module. Each of those tools worked from one source file at a time, and boxing took away what one source file could tell them. A refactoring tool that can work on one module at a time gives me capabilities. But if the module I need to get a type from exists only as a class file, which means it can only be linked, then I cannot refactor, because the linker is not the same thing as the compiler. A refactoring tool parses source text and rewrites source text. A class file is what the compiler hands on to the linker, and it holds no source text for the tool to parse.

When the makers of Java made that change, the boxing of version 1.5, I left the later versions of the Java language behind. The change made my refactoring tools obsolete overnight. So what did I do? I pinned my code to Java 1.4, and I never went to any later version, because the refactoring tools are where the gold is. A language is useful only to the degree that a person can read it after it has been written and a refactoring tool can rewrite it after it has been written. If those two conditions are not met, the language is useless.

III. Longevity

Code in a language that a person cannot read or a refactoring tool cannot rewrite stays in use only as long as some hardware somewhere still runs it, and you end up with a legacy condition: you cannot throw the computer away. Take COBOL code, for example, translated into Halo and re-

compiled on a new platform: all at once you have a thousand-fold speedup, and the capacity to move to new hardware whenever you want. The speedup is measured against the same COBOL program running on the legacy hardware it was kept on, the computer that could not be thrown away. Who would not want that? On the other hand, if you break the refactoring tools by changing the language again and again, I will say "forget it" and move on.

I kept writing to the Java 1.4 standard. Of course, the compiled code runs in every later version. For my own code base there is no reason to use any of the later features. I did not want the syntactic sugar in the first place, so I simply avoid it. If I want to pull in code written later, I write refactoring tools to translate version 1.7 back to version 1.4, and then use my existing tools on the result.

Keeping my refactoring tools working is about longevity. If you cannot refactor, you cannot have something that lasts. The needs of the world are going to evolve, and if the software cannot evolve with them, it will fail. Software without the capacity to evolve built into it is dead from the start. How does that stand to the one structure of Part I? Chapter 2 said that clarity shows up in knowing where the refactoring points are, and that third normal form is one such principle. A code base in third normal form is cleaved where the world it represents is cleaved, and those cleavages are the places where a person or a tool can take it apart and change it.

IV. A Thousand-Year Horizon

Longevity is also the reason Halo and the RISL conventions are as small as they are. They are designed for the long-term, conscious and sustainable evolution of software, and of systems built in software, in the face of existential risk and the collapse of civilizations. The evolution of humanity is measured in thousands of years. Software, in only the last few decades, has seen whole new languages, whole new paradigms of programming and standards, practices, idioms and frameworks of every kind arise and

change, and each of them becomes obsolete or is replaced, rapidly. On a scale of a thousand years we look at software completely differently, and we approach the maintenance of code in the deepest way we can imagine. We want software that will run on hardware that has not yet been invented, and software that will run on hardware that is a pale fraction of what we have today.

Take a Carrington event: a geomagnetic storm as strong as the one of 1859, which followed the solar flare that Richard Carrington observed (Carrington 1859) and disrupted telegraph systems across the world. A storm that disrupted telegraph wires across the world would strike today's long power lines in the same way, and the plants that make chips depend on that power. After such an event the computing power available to the average person would probably drop by a factor of a thousand. What the average person would have left is whatever simple hardware still runs. Infrastructure for the design of a civilization would fail at once wherever it depends on large amounts of computing power, as most cryptographic systems do. So Halo and the RISL conventions are designed to need as little computational strength as possible to support that infrastructure.

Halo keeps the complexity with which a system is represented as close as possible to what Turing completeness strictly requires. A language may have a great many features, elaborate handling of threads and of objects, and a great deal of syntactic sugar for special cases, and none of that can be assumed to be available indefinitely. There is a way of writing code that depends on little in the way of infrastructure: on no particular optimizer, compiler or design of operating system. Writing to that minimum of infrastructure means using the smallest number of constructs needed to represent the logic, and a language of computation needs very few constructs (Chapter 5). Suppose a loop is written in a language's newest form for walking through a collection. It needs a compiler that knows that form. Written as a counter, a test and a jump back, the same loop needs only what every machine has. Writing that way can feel spartan, and the code it gives can be carried to hardware that does not yet exist.

The same horizon bears on hardware. Today's chips are made of silicon, with billions of dollars of industry behind them. Integrated circuits built on carbon have been demonstrated (Hills and others 2019), and carbon circuits are not made at industrial scale, because retooling for carbon circuits, which means working out their physics from the beginning, is not competitive in today's market. A closed-loop ecology (one that reuses all of its own materials) and a sustainable economy of materials need computation at ordinary temperatures and pressures, including the making of the computing substrate itself. An ecology that reuses all of its own materials has no far-off industry to send to for a new chip, and so it has to be able to make one under the conditions it lives in. On Mars, repairing a system may mean rebuilding its microchips, and silicon technology needs too much embedded infrastructure for that: the equivalent of a whole fabrication plant would have to go to Mars, the plant would need a town to maintain it, and the town a city, until, to make anything survivable there, we would have to move the whole planet. So we need computing substrates that are far easier to make. The technical base needed to set up computing infrastructure today is beyond the capacity of most nation states, let alone of single people, and for reasons of security we will eventually need ways to make computing hardware at a far more ordinary level. Chapter 2 gave the reasons: whoever cannot make the hardware has to trust whoever does. The picture is the Mars plant turned around: hardware whose making does not need the town and the city behind it. That means accepting trades between power and performance on one side and capability and creativity on the other, trades that are no part of the equation today: we need to be able to run on less complex and more secure platforms, and not only on more complex ones. Software written today should be able to run on that kind of hardware, and that calls for great care about the idioms in which code is written.

The same smallness pays now, in practice. Projects written to the RISL conventions have stood untouched for more than ten years, and a person versed in the conventions can take the code of an archived project, come up to speed, and have it compiled and running within a few hours, at

most a day, and sometimes within the hour. One compiler can be substituted for another. Halo is an *intermediate language*, one that sits between two others, as the abstract syntax tree (the tree of a program's structure that a compiler builds from the program's source text) sits inside a compiler. The portability that Halo gives as an intermediate language is extreme. New systems can be built quickly, because writing the fundamental logic of an application is the whole of the work: there is no framework to import or to learn, no compatibility between frameworks and libraries to check, and no bug in the interaction of two libraries to diagnose for an unknown length of time. A library kept to these conventions is a circle with pins (Section VI): everything it takes in and everything it gives back crosses at a pin. Two such libraries can meet only where a wire joins them. Working at that fundamental level, with a computing substrate and well-tested libraries whose interfaces cannot entangle one another, there are no unknowns, the interaction of the libraries cannot fail, and estimates of the time a project will take become clearer and more accurate.

V. Sugar Is Waste

If one tiny bit of sugar can cost that much, why do people keep asking for it, and what should a programmer do when the wish comes? What I propose is that all of the sugar is waste, and the remedy is not for each person to make sugar of their own. A language can let each programmer add notation of their own, and that only multiplies the sugar. The whole industry at present is caught in a commercial feedback loop that holds change to be worthwhile all of the time, because change is a way to trap people into buying more. But as a software developer, what I care about is my capacity to do the work that needs to be done, which is a completely different structure of incentives from the one driving the whole industry now.

Every time you find yourself wanting sugar, it is for one of two reasons. Either you have an expectation that is actually invalid, which, in my experience, drives about nine tenths of the wishes, or, the other tenth of the time, the language itself has made something hard that at a fundamental level of computer science should be easy. Take the wish behind boxing: to write 5 where a method asks for an Integer. Behind that wish is the expectation that a number and an object holding a number are one thing, and they are two things. Written out, as bar.put(new Integer(5));, the line says that an object is made, and a tool reading that one file can see it. Now take a wish of the other kind: to give back several results from one function. Chapter 5 allows a function more than one output, and the languages in common use allow one (Chapter 11), and therefore the difficulty comes from the language. I suggest that you make the language such that you do not need the sugar in the first place.

This design stands apart from the present culture of software development. As far as I am concerned, the whole development culture made the wrong choice several times. It made those choices out of abstract mathematical interests, out of commercial interests, or because some person had an ego. That accounts, in my experience, for about 90 percent of the drivers of software change in the world today. Each of those choices is a change that every code base built on it has to absorb, as code written in Java had to absorb boxing, and so each one bears on what keeps a code base changeable.

Think of a phone replaced every year by a new model, sold because it is new. This design pattern breaks a wheel of power: the cycle in which change is sold for its own sake. A developer who has lived inside that cycle has learned to treat change as improvement. And so every developer in the world will want to find some way to change this design pattern, while it is the purity of the design that makes it work. I treat it as a whole system. As a developer, what I care about is the capacity to refactor my code.

VI. Libraries Without a Master Language

Most developers today work through package managers, pulling a package off the shelf with no idea what it does or how it works, and it just plugs in. This design is entirely compatible with that. For me, libraries are just circles with a number of pins on them. I wire them together, translate the wiring into whatever language I need in order to bundle the libraries together, and now I have an application, and I am done.

I want libraries, and I want a design that builds libraries into everything; what I am against is the idea that I need a master language (a single language in which every library must be written). Say I have a library written in C, another written in JavaScript, and another written in Python. As things stand, my options are to find some way to network them, serializing the data they exchange, with a great many things that can go wrong. Or I can take each library's language, translate it into a master form that belongs to no language (a Halo drawing), and then turn it into whatever language makes most sense for my execution environment. A master form asks less than a master language does: no library has to be written in it, each library stays in its own language, and the drawing is what the translation passes through. If I want to run something embedded, I take the Python code, turn it into Halo, turn that into assembly code, load it onto a firmware chip, and I am done. I no longer have to care what language any of these things are written in.

VII. Summary

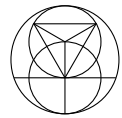
A language is useful only as far as a person can read it after it has been written and a refactoring tool can rewrite it after it has been written. Java's change in version 1.5, which made the type of a symbol depend on another file, broke the refactoring tools written for single files. So the code worth keeping was pinned to version 1.4, where the tools still work, and later code is translated back to it. Code that cannot be refactored cannot

last, because the world's needs change. Third normal form cleaves a code base at the places where a person or a tool can change it. Code that can be translated into Halo can move to new hardware whenever it is wanted. The horizon of the design is a thousand years: after a catastrophe such as a Carrington event, or on substrates far easier to make than silicon chips, software must run with a fraction of today's computing power, and so Halo keeps its complexity as close as possible to what Turing completeness strictly requires. Halo's smallness pays now as well: archived projects written to the RISL conventions compile and run again within hours, and Halo is an intermediate language between any two languages kept to the RISL conventions. Syntactic sugar is waste: wishes for it come from invalid expectations or from languages that made hard what should be easy, and much of the change in software is driven by commerce, by abstract mathematical interest or by ego. Libraries in any language are circles with pins. Translated into Halo, they can be wired together and written out in whatever language the execution environment needs, with no master language.

A refactoring tool can rewrite a call only when it can see, in the file before it, what the call takes in and what it gives back, and that is what boxing took away. Translation from one language to another asks the same of every call, and Chapter 9 named the call as the single most important part of the language. All of this rests on one way of writing a call. What, exactly, does Halo look like when it is written in JavaScript?

KEY CONCEPTS IN THIS CHAPTER

- **Refactoring tool**: a program that rewrites code without changing what it does, and where, on this account, the value of a code base lies.
- **Reflection**: a program's inspection and change of its own classes while it runs, by which hot patching is just barely possible in Java.
- **Boxing**: the Java 1.5 feature by which a symbol's type can be known only by parsing another file.
- **Longevity**: the capacity of software to evolve with the needs of the world. Without refactoring, software cannot last.
- **Master language**: a single language in which every library must be written, replaced here by translation into Halo and out again.
- **Thousand-year horizon**: the scale on which Halo and the RISL conventions are designed: software that runs on hardware not yet invented, or a pale fraction of today's, and so depends on almost nothing.
- **Carrington event**: a geomagnetic storm as strong as that of 1859, after which the computing power available to an ordinary person would probably drop by a factor of a thousand.
- **Intermediate language**: a language that sits between two others, as Halo sits between any two languages kept to the RISL conventions, and as the abstract syntax tree sits inside a compiler.



Chapter 11

Halo in JavaScript; The Module Pattern and the Call Geometry

What, exactly, does Halo look like when it is written in JavaScript, and why is that form worth its cumbersome look?

ABSTRACT

The specifics of how Halo is written in JavaScript matter, because the call is the whole matter. This chapter gives the module pattern: a function that receives one record, attaches its public methods to it, keeps its private data in local variables, and returns the record; in its canonical form the function, like every other, takes one array, creates the record itself and returns it in that array. It gives the call geometry. Every function takes one array, f. The caller loads its inputs into f by position, makes the call, and reads its outputs back from f, and the return value of 0 or 1 selects the branch the caller takes. The pattern looks cumbersome, and it is what makes the translation between a Halo diagram and its JavaScript exact in both directions, lets refactoring tools work automatically, and runs fast. I close this chapter with the written RISL standard for JavaScript, and with a module written in the canonical form that the standard requires. Every rule of the RISL standard follows from the statement that the function of code is to document its function. I set out the rules in full in *The RISL Standard; Halo Written as Code, Across Languages*. This chapter is written for readers who program in JavaScript; its Summary states its results for every reader.

I. The Module Pattern

How do I write a module in JavaScript so that any other module can reach it, and so that it has a public part and a private part? Now to how the actual JavaScript representation works, because the specifics matter. In JavaScript, I make sure that the objects that hold the names of my functions exist: the global object com and its members, and the next paragraph tells how I set them up. JavaScript allows a function to be defined in several ways, and the one I care about is essentially the definition by variable. If I write the keyword var, a name, the keyword function and the parameters, with the var at module scope, then I have to assign the function to an export, and the means of export differs by environment: the mod-

ule.exports object in Node.js, the export statement in ECMAScript modules, and none at all for a classic browser script. If in place of var I assign the function to a variable under com (com, and then the name of my function), I have visibility of it wherever I want.

For com to be visible to every module, it has to be made a scope above all of them. In a classic browser script, the variables defined in each module are visible to every other module; in Node.js the variables of a module are private to it, and another module can see them only through what the module exports, by the module.exports object in Node.js or by the export statement in ECMAScript modules. Think of the first person to arrive in a new office, who puts up one shelf with a label on it, so that everyone who comes later knows where to put things and where to look. So one of the very first modules Node.js loads, in everything I run, is a small one that sets up com. It is a short series of statements that, by trying every way there is to define a global object, guarantees one global object in every environment I have tested, which is a great many: Node.js, every variety of browser and tools such as PhantomJS. It looks redundant, and it works everywhere. I do not treat that module as my own code: I treat it as part of the runtime, a repair of something its implementers left out, and so if something goes wrong I can adjust the setup in one place. At the top of each module the same check is made one level down: if com has no member for the name of my company, I add one, an empty object, and if that has no member for my project, I add that, and then I add my own things to it. A reader who also writes Java has already met this shape. In Java the naming works by the same dot: one element is reached within another, along a tree of object relationships that are, underneath, hash tables within hash tables.

Assigning the function to a variable under com breaks out of the frame of the module, and now I have visibility of the function wherever I want it. Assigning a function to a variable under com is the way I export my classes. Listing 11.1, a few lines below, shows the whole of what follows. In a class, in place of a set of parameters, all I need is a place to which the methods are exported, so I put a single variable, r, in the parameter list.

Then each function definition is written r dot the function's name, repeated over and over. Now I can call from the outside and get a collection of methods. If I want to, I can put my common data in the body of the class, declared with var, and that data is now private. The methods can reach the data, but I cannot reach the data from outside by that name. That gives me my public and private scope.

Every time I call the function, it generates a new instance of these methods. A member is private if I put the var keyword in front of it, and public if I put "r." in front of it, because r is exposed. The last thing in the function is "return r", so that the caller gets the record.

```
com.company.project.drill = function (r)   // one parameter:
r
   {
   var fl_count = 0;          // private: var in front
   r.cut = function (f)       // public: r dot function name
      {
      fl_count = fl_count + 1;
      return 1;
      };
   r.count = function (f)
      {
      f[0] = fl_count;
      return 1;
      };
   return r;                  // the caller gets the record
   };
```

Listing 11.1. The module pattern described above, written out: methods attached to r are public, variables declared with var are private, and return r gives the caller the record.

Listing 11.1 shows the shape of the pattern. Its canonical form, which the written standard of Section V requires, is Listing 11.3: there the constructor, like every function in the system, takes the one array f, creates its record itself, gives its names their prefixes and version suffixes, and returns the record in f[0] with a return of 1.

The module pattern is essentially a way of constructing modules. I have one of these for the record and another for the table, and beneath them a singleton function defined at module scope, followed by an invocation of that singleton. When the module loads, the singleton is called, the singleton calls the table module, and the table module executes and sets up the table. Now I have a table loaded into the global namespace, which I can use to create records and reach the methods of records. Take the drill module of Listing 11.1. The file loads, its singleton runs, and the table of drills is set up under the project's name. Any module can now ask that table for a new drill, and can reach that drill's cut method through the table. Everything in this construction fits together.

A reader may ask whether loading each module's table into the global namespace crowds the top level of that namespace. I have one variable at the top scope, and it has children, and those children are my tables. The namespace is not crowded at all. It is not crowded for the same reason that I use the convention of com, the name of the company, and the name of the project. For the most part the namespace is relatively coherent at the global level.

The whole point of design in third normal form is that it is very well suited to composition. I almost never use inheritance. In my entire career I have thought of perhaps two situations in which inheritance makes sense. Problems such as diamond inheritance (a class inheriting from two classes that share an ancestor) have never even come up. Embedding relationships are almost always what I use: containment, inclusion, function composition.

II. A Translation Exact in Both Directions

Why accept a way of calling that looks so awkward? The way of calling is a little more cumbersome than we are used to. Section III shows it line by line. What it gives us is an isomorphic transformation, a translation exact in both directions, between the underlying structure of the Halo language

and its JavaScript representation. I need it to be bidirectional. If I change a diagram, I want the JavaScript code to update automatically. If I change the JavaScript code, I want the diagram to update automatically. Suppose I draw a second Ring Bell circle after the first in hello world. The code gains one more call, written like the others. Suppose I then delete that call from the code. The circle leaves the drawing. So I need to be disciplined about how I make the call structures, so that the code parses directly back into the graphics. I could take raw JavaScript and parse it into the graphics, but when I came back out again the code would be in the canonical form, the one way of calling that Section III shows. Exact in both directions (Chapter 7, Section II) means that a drawing written out and read back is the same drawing, and that a text read in and written out is the same algorithm, in the canonical form of that text's language.

Post-processors carry raw JavaScript to the canonical form. There are dozens of them, which return raw JavaScript to the canonical form, and I have built them.

I have done translation from JavaScript to ANSI C. Getting from compiled code, such as assembly language, to something readable and manageable is difficult, and there are many problems in it. But once code is in structured form, you can go from almost anything to anything; that is now achieved, and to me it is a settled matter. Take one call in the form of Section III. Its inputs go into numbered positions of an array, the call follows, and the outputs come back from the same positions. ANSI C can write each of those lines in the same order, because it has arrays, calls and a returned 0 or 1.

And we are willing to give up some performance, because I am working at the level of architecture. If I care about performance in one section, I optimize that section. What matters, usually, is readability, and performance on a global scale is not what concerns us. The pattern is often the more efficient one, apart from certain edge cases. When we reach the actual geometry of the call, it looks wickedly strange, but it is actually faster than the way of calling we are used to. A register is one of the few small storage places inside a processor, and the fastest it has. Because the same

array is used for call after call (Section III), the same few places are used over and over. What happens is that the execution system ends up caching those register addresses, and then the code runs very fast indeed.

Chapter 10 said to make it so that you did not need the sugar in the first place. What I am aiming at in the design of the Halo language is something so close to the truth of how we actually think that we never feel the need for sugar.

III. The Call Geometry

What does a call actually look like on the page, and why? In this section I describe the geometry of a function call, so that the geometry does not come as a shock. We normally think of a function as having input parameters, but we need to represent multivariate output, because multivariate output was never ruled out: Chapter 5 allows a function to produce more than one output. Multivariate output turned out to be very important, particularly in Java, where I have multithreaded code. Sometimes I need to create a synchronized data structure: I need to see the state of a thing. I do not want to create a new data structure every time, because that clutters the namespace, but I do need to see all of those variables at one moment in time. Think of a group photograph: everyone in it at one moment, where asking people one at a time would give each answer at a different moment. A function with several outputs hands them back as that photograph. You would be surprised how much less synchronization you need if you can create states whose data is synchronized automatically. So multivariate output is very important. But all of the actual languages (ANSI C, Java, JavaScript) provide many input parameters and one output parameter. We need a way to represent output parameters at the same structural level as input parameters.

The way I do it is that every function definition takes a single parameter, f, and f is an array. The first time I call, at the very top of the call tree, I call the function with an open bracket and a close bracket, which creates a

standard array, and that becomes my f. Every function defined anywhere in the system has this form, and its parameter is always the letter f, in lower case, as every name is. Why? Simply because it is easy.

When I want to call another function, the first thing I do is load my input parameters, using ordinal notation: f[0] equals some string, f[1] equals some number. Then comes the actual call. If the function returns an error condition, I handle it. Otherwise I pull my output variables: some local variable equals f[0], another equals f[1]. At the top of the called function, I pull the input variables into local ones. Then, just before the return statement, I assign the outputs: f[0] equals a local variable, f[1] equals a local variable. Then comes the return (Listing 11.2).

```
// caller: inputs into ordinal slots, call, outputs back to names
f[0] = "Hello %";
f[1] = fl_name;
if (com.company.project.print(f) == 0)
   {
   // handle the error condition
   }
fl_written = f[0];

// callee: ordinal to local on entry, local to ordinal on exit
com.company.project.print = function (f)
   {
   var fl_format = f[0];
   var fl_text = f[1];
   var fl_written = 0;
   // ... the function's own work ...
   f[0] = fl_written;
   return 1;      // here 1 for success, 0 for fail
   };

// the first call, at the top of the call tree
com.company.project.main([]);
```

Listing 11.2. The single-array calling convention, written out: every function takes one array f; the caller loads the inputs into f by position and reads the outputs back from it; the return value of 0 or 1 selects the branch the caller takes.

Where, in com.company.project.print(f), are the three things of the call triple of Chapter 9? Print stands directly under the project's name and has a single instance, so there is one record to choose, and com.company.project names its table and that record at once. The last name, print, is the method. Where a module holds many records, one more step names the record, as in com.company.project.drills[3].cut(f).

What does this accomplish? My input variables are pulled into local variables, so I move from an ordinal notation to a named notation. When I reach a function call, I map from my local variables to ordinal slots within the array f, reusing the same array. I call the function, and it takes its input variables from the ordinal notation and does whatever logic it does. Just before it returns, it assigns its outputs to the positions in f. When I come back out, assuming there is no error, I recover the outputs into local variables again. So the wiring pattern is always the same: from ordinal to local, then from local to ordinal, then from ordinal to local and from local to ordinal again, over and over, every time I pass through a state transition.

I use the return keyword to signal a 0 or a 1, and that value controls the branch of the enclosing call. So the notion of a conditional and the notion of a function call are strictly isomorphic to one another, and I get my routing as a result. Across the two levels, the caller and the callee, the pattern works through the array f, which carries the values, and through the return value of 0 or 1, which selects the branch.

Again, it looks cumbersome, but the good thing about it is that it allows block editing. I can move code around just by capturing a whole section, and so I can restructure the flow, which turns out to be what you need to do most often when you refactor. Suppose a line reads print(format(trim(name))), and I want to put a logging step between the trim and the format. I have to take that line apart first. Written in this pattern, each of the three calls is its own block of lines: load f, call, recover from f. No block is nested inside another, so I capture a block whole and move it, or slip another block between two. Because everything is con-

tained in this way, I can track everything a particular call requires, and everything coming back from it, in the same way. That means my analytic tools can now do the refactoring automatically.

I also put structure checks everywhere in a system. In place of a bare return of 0, I usually print a USI number, the name I give the identifier of Chapter 3, Section II (a uniform source code unique number), so that the log file shows exactly where each error condition is thrown and I know precisely where in the code to go. Where knowing the exact place of an error matters, the USI number is a great help, because the USI number lets me do diagnostics in minutes rather than days. In effect I get the output of a stack trace, whatever the language.

JavaScript does not always produce a stack trace. If you are working on an iPhone, for example, getting it to produce a stack trace is very hard: it is possible, but only with great effort. Being able to produce signals that stand for a stack trace is very useful.

It also unifies the whole code base, because everything can be translated into this form. I could take existing code and translate it into this form in one pass with a refactoring tool. It always works, guaranteed.

Going from ordinal to local and back to ordinal simplifies the mapping, and it preserves the integrity of the data. A value in f lasts only until the next call writes over that position, and the step into a local name takes the value out of reach of that overwriting. The step into local names is the pull, and the step back into the array is the push. What you see here is a direct one-to-one translation between the underlying Halo concepts and their representation in JavaScript, and in Java you do the same thing.

The good thing is that, because I reuse the array, the execution environment treats f as a local register set. Because it caches the local registers and uses them over and over for both the inputs and the outputs, the stack trace is very clean. Each call leaves on the stack only its own local variables, because its inputs and outputs travel in the one array, and so a trace has little in it to sort through. The pattern creates almost no garbage. I am not using the stack very much: it holds the local variables of a function,

and when the function exits it simply drops them all. The registers keep the actual transfers from one call to the next. So this turned out to be very fast. It also lets me go directly to things like ASM.js (a strict subset of JavaScript designed to be compiled to run at close to the speed of machine code) and its successor, *WebAssembly* (a compact binary format for programs, run by web browsers at close to the speed of machine code; SpiderMonkey team 2026). The call pattern, with its one array f, is a natural form for ASM.js. If I want to think in terms of assembly language code, I am already there.

IV. The Pure Geometry and the Working Code

The underlying geometry is a way of showing that algorithms can be represented in a purely abstract way. When we talk about actual working JavaScript code, it is usually easier simply to write it straight in JavaScript. Keeping to conventions like these means that the code has a standard form that is very easy to parse. I do not have to spend much time reading code in this form, because it is very simple to read. It has the kind of structure that corrects mistakes before they are made.

At this point we have looked at the overall geometry of how to build systems, and at some of the specifics of how Halo translates into JavaScript. The details of how to write JavaScript code that complies with the RISL conventions, and of how JavaScript represents each of these things, are given in *The RISL Standard; Halo Written as Code, Across Languages*, Chapters 1, 4 and 5. Among those conventions are the ITAG convention and EGS (Extended Grammatical Structuring), both of which are part of RISL; their full standards are set out in *The ITAG and EGS Standards; Structured Plain Text for Files and Notes*, ITAG in its Chapter 1 and EGS in its Chapter 2. Both are named here because they set the layout of a module's file and the form of its notes, as Chapter 9, Section IV described. Section V of this chapter gives the canonical form of a module.

A Halo program can be drawn at more than one level. The circle diagram is the most abstract version. There is another version, closer to what a flowchart looks like, with the actual data values still there, which is also isomorphic with the circle diagram and much simpler to read. Figure 11.1 draws hello world in that version, beside the circle diagram of Figure 6.9. The flowchart version is a reading aid derived from the circle form, and it has no rules of its own: each of its circles is a call of the circle form, each line running down the page is a trigger line from a success exit, each dashed line is a trigger line from a fail exit, and each named value written beside a call stands for a named variable at a pull or a push. A tool reads and writes the circle form, and draws the flowchart version from it. A reader who has met Lisp, the older language whose programs are written as one plain nested structure, will ask how the circle form stands to it. And the Halo geometry, the representation by pure geometry (the circle diagram), is pristine in a way that even Lisp does not reach. Lisp has all of its special forms to handle, and Halo is more general. A special form is a construct that a language evaluates by its own rule, and not as an ordinary call, and so each one is a separate case. Halo has one element, and one rule for it. Lisp is not designed, or intended, to handle multithreaded code. Clojure later brought threads into a Lisp, but threads are not natural to Lisp itself, whereas with Halo they are. The standards of the classic Lisps, Common Lisp and Scheme, define no threads at all (ANSI 1994; Shinn, Cowan and Gleckler 2013). Clojure, a later Lisp built for concurrent programs, manages the state that threads share through its own reference types (atoms, refs and agents), written as text (Hickey 2020). In Halo the threads are the drawing itself: a trigger routed to several circles at once is a multithreaded program (Chapter 6), and every crossing of a boundary by a trigger is a synchronization of the data (Chapter 5).

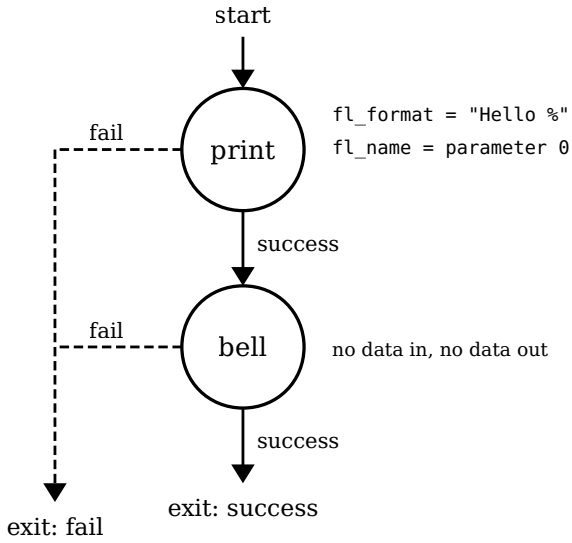


Figure 11.1. Hello world in the version of a Halo drawing that is closer to a flowchart. Control runs down the page from start, through print and bell, to the program's success exit, and each fail (dashed) runs to the fail exit. In place of data wires, each call carries its data values as named intermediate variables, as Chapter 7 describes; the drawing is isomorphic with the circle diagram of Figure 6.9, with the named state start of Figure 7.3 (a) added.

V. The RISL Standard for JavaScript

The RISL conventions for JavaScript are also written down as a standard, and its rules follow from one statement: the function of code is to document its function. The standard is set out in full, with every rule and its reason, in *The RISL Standard; Halo Written as Code, Across Languages*, Chapter 5. What a reader of a Halo program needs from it here is little. The standard calls the shared namespace object cc, where the examples of this chapter call it com; a module's constructor carries cl_ in its name, its state variables ss_ (static structure, Table 7.1), and its functions fn_; and every name ends in a version suffix, _1, so that no name is ever a proper

part of another. The names cc and com stand for the same object, and a reader can read one for the other. The suffix makes each name a unique key: a search for it finds that name and no other.

The written standard also keeps a reduced JavaScript, fixed at Node.js version 8 with nothing added since, so that every construct has one canonical form; gives every module that holds state across a reload the lifecycle functions fn_init_1, fn_term_1, fn_link_1 and fn_kill_1, which carry its state across a reload while the system runs (fn_term_1 saves what must survive the reload, and fn_init_1 restores the state afterward); loads each input into f from a named variable before every call and recovers each output into a named variable after it; puts one action and one kind of value on each line; bounds every loop in plain sight; sets the form of the page (indentation of three spaces, braces on lines of their own, related assignments aligned on their equals signs), so that each construct has one appearance a tool can find; and returns every failure as a trap, an error report that always returns 0 and whose identifier, in the form I_XXX_XXX_X, is unique across the code base. *The RISL Standard; Halo Written as Code, Across Languages*, Chapter 5, gives each rule with its reason.

Listing 11.3 writes the drill module of Listing 11.1 in this canonical form. It is a module with a single instance, so its table holds the one record, and it leaves out the four lifecycle functions to stay short. Its count is a field attached to r, which puts it within reach by the rule of Section I. The prefix ss_ says that only the record's own functions touch that field, and in the listing every other caller asks fn_count_1 for the count.

```
// the constructor: defines the record of the drill module
cc.shop_cl_drill_1 = function (f)
   {
   var r = {};

   r.ss_count_1 = 0;                   // state: a field of the record

   // intent:  add one to the count of holes cut
   // return:  1, or a trap
   // example: if (!fn_add_one_1(f)) return cc.fn_trap("I_XXX_XXX_X");
   var fn_add_one_1 = function (f)     // private: a variable
      {
      try
         {
         var fl_count_1 = r.ss_count_1;
         fl_count_1     = fl_count_1 + 1;
         r.ss_count_1   = fl_count_1;
         return 1;
         }
      catch (fl_err_1) { return cc.fn_trap("I_KQW_RMB_T"); }
      };

   // intent:  cut a hole in the box given in f[0]
   // return:  1, with the box in f[0], or a trap
   // example: f[0] = fl_box_1;
   //          if (!r.fn_cut_1(f)) return cc.fn_trap("I_XXX_XXX_X");
   //          fl_box_1 = f[0];
   r.fn_cut_1 = function (f)           // public: attached to r
      {
      var fi_box_1 = f[0];
      try
         {
         if (!fi_box_1) return cc.fn_trap("I_PZD_HNL_F");
         var fl_box_1 = fi_box_1;
         if (!fn_add_one_1(f)) return cc.fn_trap("I_VXC_JSG_Y");
         f[0] = fl_box_1;
         return 1;
         }
      catch (fl_err_1) { return cc.fn_trap("I_BTN_QWE_K"); }
      };

   // intent:  give the count of holes cut, in f[0]
   // return:  1, or a trap
   // example: if (!r.fn_count_1(f)) return cc.fn_trap("I_XXX_XXX_X");
   //          fl_count_1 = f[0];
   r.fn_count_1 = function (f)
      {
      try
         {
         var fl_count_1 = r.ss_count_1;
         f[0] = fl_count_1;
```

```
         return 1;
         }
      catch (fl_err_1) { return cc.fn_trap("I_MLA_ZOC_U"); }
      };

   f[0] = r;                            // the record, in the call array
   return 1;
   };

// the init function: makes the table, the module's single instance
cc.shop_fn_drill_1 = function (f)
   {
   try
      {
      if (!cc.shop_cl_drill_1(f)) return cc.fn_trap("I_GHE_TYR_D");
      var fl_rec_1 = f[0];
      cc.shop_ss_drill_1 = fl_rec_1;
      return 1;
      }
   catch (fl_err_1) { return cc.fn_trap("I_WUO_PLS_N"); }
   };
```

Listing 11.3. The drill module of Listing 11.1 in the canonical form of the written standard: the shared namespace cc; a constructor whose name carries cl_, which takes the array f, creates the empty record r, gives it a state field carrying ss_, and defines a private function as a variable and public functions attached to r, each carrying fn_; a version suffix on every name; and, at its end, f[0] = r; return 1;. The init function stores the record as the module's single instance, its table, under a name carrying ss_. Each function opens with its header and wraps its body in a try and a catch, and each failure returns a trap with an identifier of its own; the lifecycle functions, the call to the module's entry point, the file's ITAG header, the line at its foot that constructs the module, and its terminator are left out.

Each rule is the Halo element written in text. The array f is the bundle of data pins, the return of 1 or of a trap is the pair of exits, success and fail, the loop with its bound at the top is the counted ring (a loop drawn with a counter on the path back to its start), and the module with its record and its single instance is the table of Chapter 9. The rules stand close to those written for software on which lives depend, which also bound every loop, check every value returned and keep to a part of the language that can be analyzed: Gerard Holzmann's ten rules for safety-critical code at

NASA's Jet Propulsion Laboratory (Holzmann 2006), and the MISRA guidelines (the Motor Industry Software Reliability Association) for C in vehicles (MISRA 1998).

VI. Summary

In JavaScript, a Halo module is a function, filed under the name of a company and a project, that receives one empty record, attaches to it the methods other code may call, keeps its private data in variables that only it can see, and hands the record back. In the canonical form of the written standard, the function that builds the record takes the one array that every function takes, creates the record itself, places it in the first position of that array and reports success (Listing 11.3). One such function makes records, another makes the table, and a singleton sets the table up when the module loads. Design in third normal form uses composition, containment and inclusion, and almost never inheritance. Every function takes one array: the caller loads inputs into the array by position, calls, and reads outputs back from it. The called function pulls its inputs into variables with names of their own and pushes its outputs into the array just before it returns 0 or 1, and that return selects the caller's branch, so that a call and a conditional are the same structure, exact in both directions (strictly isomorphic). The pattern looks cumbersome, and it buys a great deal. It gives an exact translation between the diagram and the code in both directions, block editing, and refactoring done automatically by tools. It gives a stack trace in any language through USI numbers, and fast code, since the array works as a register set. Halo's pure geometry is more general than Lisp, and multithreading is natural to it, since its threads are drawn. A small module loaded first makes one shared name, com, under which every module is found, visible in every environment. The written RISL standard for JavaScript follows from one statement, that the function of code is to document its function. It keeps a reduced JavaScript, names that carry their role and a version suffix, lifecycle functions for reloading, one

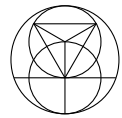
array loaded and recovered through named variables at every call, one action and one kind of value on each line, loops bounded in plain sight, and failures returned as traps with identifiers unique across the code base. *The RISL Standard; Halo Written as Code, Across Languages* sets out each of these rules, with its reason, in its Chapter 5.

The code now does exactly what its diagram says. How can a person be sure that the code a computer runs does what its author intended?

KEY CONCEPTS IN THIS CHAPTER

- **Module pattern**: a function that receives one record r, attaches public methods as r dot name, keeps private data in var declarations, and returns r; in canonical form (Listing 11.3) the constructor takes the array f, creates r, and returns it in f[0] with a return of 1.
- **Composition over inheritance**: building by containment, inclusion and function composition, which is the use that third normal form suits.
- **Isomorphic transformation**: a translation exact in both directions, here between a Halo diagram and its JavaScript.
- **Call geometry**: every function takes one array f, and inputs and outputs pass through it by position.
- **Ordinal to local, local to ordinal**: the wiring pattern of every state transition, which is the pull and the push of the Halo element.
- **USI number**: the name given to the identifier of Chapter 3, a uniform source code unique number, printed at each point where an error can be signaled, so that a log shows where in the code the error arose.
- **EGS (Extended Grammatical Structuring)**: my standard for writing notes, in which each statement begins with a dash and ends with a period, opens with where, that or as, and takes its context from its indentation; a substandard of RISL.
- **Reduced subset**: the JavaScript of the written RISL standard, fixed at Node.js version 8 and without the conveniences added since, so that every construct has one canonical form.
- **Version suffix**: the _1, _2 that ends every name, so that no name is part of another and every name is a unique, searchable key.
- **Lifecycle functions**: fn_init_1, fn_term_1, fn_link_1 and fn_kill_1, which carry a module's state across a reload while the system runs.
- **Load and recover**: before every call each input is loaded into f from a named variable, and after it each output is recovered from f into a named variable.

- **Trap identifier**: the USI number in the form I_XXX_XXX_X, unique across a code base, returned and logged at the point of every failure.



Chapter 12

Code as Intention; The Hard Problems and the Integrity of Execution

How can a person be sure that the code a computer runs does what its author intended?

ABSTRACT

Halo represents code as intention in diagrams, and a one-to-one relationship between hardware and intention is the path toward trustable computing. This chapter sets that claim beside the three things that are extraordinarily hard in computer science (multithreaded programming, cryptography and compiler design), and beside the three threats to the integrity of a program (exfiltration, infiltration and modification). It then compares two ways of making execution trustworthy that are in use, computation spread across many machines and hardware sealed by its maker, and shows what each one costs, and that trusted hardware serves its maker's will. Halo, if it is implemented all the way to hardware, allows a one-to-one relationship between the hardware and the intention drawn in a diagram the author can read, and that is the path toward trustable computing.

I. Security From the Point of View of Intention

Chapter 2 traced the regress of trust: we trust the hardware, the language, the compiler and the tools of inspection, each resting on the layer beneath it, before we say anything about the people in our community. Chapter 2 also showed why a community needs to know that the code actually running is the code it elected to run. In this chapter I ask the same question from the side of the one who writes the code.

Think about security from the point of view of intentionality. A person writes software expressing their intention. We want to be sure that what they intended is what actually happens.

II. Three Threats

There are three threats to that, and each has its name.

The first threat is *exfiltration*: someone steals data, such as a password typed into a bank's website, or the key to an encrypted message.

The second threat is *infiltration*: someone inserts a Trojan horse, hidden code that makes the system do that person's bidding rather than the bidding of the system's author.

The third threat is *modification*: the author wrote code that expressed the author's intention, and someone changed that code, so that what actually happens is not what the author wanted.

Each threat is a difference between the diagram the author drew and what the hardware does, and Halo, implemented all the way to hardware, leaves each of them a place where it can be seen. Think of a music box: each pin on the box's drum strikes one note, in order, and between the pin and the pin's note there is nothing that could play anything else. Implemented all the way to hardware means what Chapter 5 said a drawing can be: compiled to circuits that run straight up in hardware and represent the computation directly. Data leaves a system only along a path, and in a Halo drawing every path of data is drawn: a data line that crosses a

boundary without meeting a pin is a global value (Chapter 6), visible as such, so a path along which data could be exfiltrated is a line in the drawing. A Trojan horse is code that has no place in the author's diagram, and whatever cannot be drawn is not code (Chapter 7); where the hardware carries out the diagram one to one, code that is not in the diagram has nowhere to run. A modification of the code is a modification of the diagram, because the translation between the two is exact in both directions (Chapter 11), and so the author who reads the diagram reads the change.

Hello world, as Figure 8.6 draws it, shows each of the three. It is a program of two calls, Print and then Ring Bell, and Figure B.1 repeats the drawing. Suppose the name typed after hello were a password, and someone wanted it to leave. To leave, it would have to follow a data line from fl_name across the membrane of the program to something outside, and in the drawing that is a line the author did not draw, there to be seen. Suppose someone else added a circle, one that sends the name to a server of theirs. In the drawing it is a circle, with its instance at its zenith and its own wires, and the author sees a circle he did not draw. Suppose someone changed one wire, so that Print's fail exit ran to Ring Bell in place of the program's exit F. In the diagram read back from the code, that wire now runs to Ring Bell, and the change is visible as the change.

Suppose a circle compares a typed password with the stored one, letter by letter, and leaves at the first letter that differs. A guess with more right letters takes longer to be turned down, and someone with an accurate clock can learn the letters one at a time. A value can also be learned from outside by a route that is no path of data at all, through timing or power. A *side channel* is such a route: a measurement, from outside, of how the hardware carries out a program, such as the time it takes, the power it draws or what it radiates, from which a value can be inferred. It is not a line of data, and a drawing does not close it by itself. The drawing gives the place to look for a side channel. Every circle that pulls the secret value is drawn, so the places where the handling of the secret value could vary with that value are the ones to examine; in the password case, the compar-

ing circle is one of those places. A linear compiler (Section III) keeps each element of machine code in the order that the diagram gives, so that what the hardware does with the secret can be read beside the drawing.

III. The Three Hard Problems

Three things in computer science are extraordinarily hard: multithreaded programming, cryptography and compiler design.

Cryptography is very easy to get wrong, and you do not notice the failure until somebody has robbed the bank. Compiler design is not quite as treacherous, but errors are still easy to introduce. Multithreaded programming is simply hard. All three make heavy demands on the mind.

Each fails in a way of its own, and Halo's limits bear on each. Multithreaded programming fails by a race: two parts of a program write the same value, and the order in which they happen to run decides what is read. Halo's limits remove the race from the default program and show it wherever it can arise: each input reads from one source at a time, each trigger leaving a call goes to one place unless the program is meant to split, and every crossing of a boundary by a trigger is a synchronization of the data (Chapters 5 and 6). Cryptography fails silently, as in the generator with a planted weakness of Chapter 2, and nothing in a drawing can repair a weak cipher; Halo's answer is to need cryptography less, as the next paragraph says. Compiler design fails when the compiler puts into the machine code something the source does not say, as the compromised compiler of Ken Thompson's lecture does (Chapter 2). Halo's limits make each circle a unit with one way in, two ways out and no trigger line crossing its boundary, so that a compiler can translate the drawing one circle at a time, and its output can be checked against the drawing one circle at a time.

I focus on compiler design, because I want not to have to solve the cryptography problem or the parallel problem (multithreaded programming). The way around the parallel problem is in the paragraph above:

Halo's limits take the race out of the default program, and a compiler that translates one circle at a time carries those limits into the machine code. The way around cryptography is to make the compiler design as linear as possible. A *linear* compiler design is one in which each element of the diagram becomes one element of machine code (one short run of instructions), in order, so that a reader can check the one against the other. The guarantee wanted is that the machine code does what the diagram says. With a linear compiler, that guarantee is checked by reading the diagram beside the machine code, element by element, and a check made by reading needs no secret key and rests on no hidden logic that could be made one increment harder to inspect. Chapter 2 counted the tools of inspection among the things we have to trust. Here the tool is a listing of the machine code in order, each run of it short enough for a person to check by eye, as Table 12.1 shows for one call.

One call shows what a linear compile looks like: the call to Print in hello world, as Figure 8.6 draws it. Table 12.1 sets each element of the drawing beside the line of Listing B.1 that writes it in JavaScript and the step the machine code takes for it, in the order the drawing gives.

Table 12.1. The call to Print, compiled linearly: each element of the drawing, the line of Listing B.1 that writes it, and the machine code's step for it, in order.

Element of the drawing	Line of Listing B.1	Step of the machine code
The pull of the constant "Hello %" on Print's first data input	`f[0] = fl_format_1;`	Store the constant's address in slot 0 of the call array.
The pull of fl_name on Print's second data input	`f[1] = fl_name_1;`	Store the value held for fl_name in slot 1 of the call array.
The trigger entering Print, through the instance at its zenith	`cc.osys_fn_print_1(f)`	Load the address of Print's instance, and jump to it with the address of the call array.

Element of the drawing	Line of Listing B.1	Step of the machine code
The two exits of Print, T and F	`if (!...)`	Compare the one bit Print returns with zero.
The exit F, wired to the program's exit F	`return cc.fn_trap("I_PRN_TWO_L");`	If the bit is zero, branch to the program's exit F, which logs the trap identifier and returns zero.
The exit T, and the push of Print's output into fl_count	`fl_count_1 = f[0];`	Otherwise, load slot 0 into the place held for fl_count, and go on to the next element, the call of Ring Bell.

Each row is one element, and each element gives one short run of machine code, in the order of the drawing, so a reader can check the one against the other row by row. The first three rows are one moment of the drawing: the trigger enters Print and pulls its two inputs. The machine code writes that one moment as three steps, and it stores the two inputs first, because they must be in the call array when the jump is made. A reader may ask whether checking machine code against the drawing this way ends the need to encrypt what is sent over a network. The linear compiler takes the place of cryptography in one use: the check that the code a machine runs is the code its author intended, which reading the drawing beside the machine code makes with no secret key. Keeping a message private while it crosses a network that others can read is a different use, and the linear compiler leaves it in place: such a message still needs a cipher, and the cipher is then one more circle in the drawing, its pulls and pushes drawn like those of every other.

IV. Trust in Code and Trusted Hardware

Two ways of making execution trustworthy are in use, and each has its cost.

Think of a ledger copied out by thousands of clerks, each of whom checks every line of every other. Ethereum's approach is code executed in a distributed computer, such that the execution cannot be tampered with, because the substrate is factored across many machines. The cost is the complete infrastructure of cryptography, enormous redundancy, and a cryptocurrency to pay for the gas (the fee, paid in Ethereum's currency, for each step of computation). It is slow and expensive, and it is socially corrosive for the reason given in Chapter 2: security that rests on logic can always be made one increment harder to inspect than the tools of inspection allow. So whoever holds more computational or mathematical power can hide intent, and no outside party can establish whether the holders of that power have done so. A community in that position has to take the holders of that power on faith.

The trusted compute hardware approach is the Secure Boot (a check that lets a computer start only code signed with an approved key) of the Unified Extensible Firmware Interface (UEFI), together with a Trusted Platform Module chip (a sealed chip that holds keys and records what the computer has started). UEFI is the firmware that starts a computer before its operating system, and on most personal computers the default keys of its Secure Boot are Microsoft's. The approach works like a dongle you cannot reverse-engineer. But you are depending on a manufacturer: a potential backdoor, a Trojan implementing the corporation's will rather than yours. Trusted compute hardware is trusted only to the corporation.

Halo gives us a way to think about the relationship between intention and result. If it is implemented all the way to hardware, it allows a one-to-one relationship between the physical hardware and the intention of the computation as represented in a diagram. A linear compiler carries a drawing to the machine code of an ordinary processor, where a reader can check the one against the other. Compiled to circuits, the drawing is carried further, and the circuit itself represents the computation. Halo implemented all the way to hardware is the path toward trustable computing.

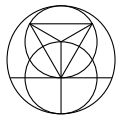
V. Summary

Trust in code rests on a one-to-one relationship between what the author intends and what the hardware does. Three threats break that relationship: exfiltration, which steals data; infiltration, which makes a system serve someone other than its author; and modification, which changes what the author wrote. A drawing carried one to one into hardware leaves each threat a place where it can be seen: every path of data is drawn, code that is not in the diagram has nowhere to run, and a change to the code is a change to the diagram. Of the three hard problems of computer science, multithreaded programming, cryptography and compiler design, I focus on compiler design so as not to have to solve the other two, and the way around cryptography is to make the compiler as linear as possible, each element of the diagram becoming one element of machine code, in order, so that the one can be checked against the other by reading. The call to Print in hello world, compiled this way, gives six elements and six short runs of machine code, row by row. The linear compiler takes the place of cryptography in the check that the code a machine runs is the code intended; a message kept private across a network still needs a cipher, drawn as one more circle. Computation spread across many machines protects execution at great cost in money, speed and social trust, and hardware sealed by its maker is trusted only to its maker. Halo, carried down to the hardware, gives a diagram that the author can read and that the hardware carries out one to one, and that is the path toward trustable computing. So far the relationship has been between one author and the code, and code is also shared among authors. How people can share and merge code without losing that relationship is the subject of *Keeping Software Soft; Explicit Code, Safe Change, and the Discipline of Subtraction*, in its account of code collaboration and semantic merging (its Chapter 4). It shows that code is shared without losing that relationship when it is merged by its structure (by its syntax tree, and not line by line), when what is shared is insight, and when the machine it is written on is kept under its owner's control.

Trust in code, as this chapter has found it, depends on the author being able to read the drawing and know what the hardware does. That holds only where the drawing is the program itself. I have long wanted a diagram that is the program and not a picture of it. What form must such a diagram have, and what makes Halo that form?

KEY CONCEPTS IN THIS CHAPTER

- **Exfiltration**: the theft of data from a system.
- **Infiltration**: the insertion of a Trojan horse that makes a system do someone's bidding rather than its author's.
- **Modification**: a change to an author's code, so that what happens is not what the author wanted.
- **The three hard problems**: multithreaded programming, cryptography and compiler design.
- **Linear compiler design**: a compiler in which each element of the diagram becomes one element of machine code, in order, so that a reader can check the one against the other.
- **Trusted compute hardware**: hardware sealed by its maker, such as UEFI Secure Boot with a Trusted Platform Module chip; trusted only to the maker.
- **Trustable computing**: computing in which the hardware carries out, one to one, the intention represented in a diagram its author can read.
- **Side channel**: a measurement, from outside, of how the hardware carries out a program (its timing, its power, what it radiates), from which a value can be inferred; not a line of data, so a drawing shows where to look and does not close it by itself.



Chapter 13

Drawing the Program; The Axiomatic Form

I have long wanted a diagram that is the program and not a picture of it. What form must such a diagram have, and what makes Halo that form?

ABSTRACT

Halo has the axiomatic form. A language for drawing programs has this form when every element it has is required for completeness of representation, when nothing else is in it, and when the same elements cover every layer of compute. This chapter derives the elements from the one question a program answers, what happens when, and shows what is lost when any one of them is taken away. The derivation gives one circle, two directions, three kinds of crossing, six parts and one rule, and it shows that any language with the axiomatic form is Halo drawn another way. The six parts map one to one onto the three modalities of the Immanent Metaphysics and their pairs. In this chapter I then say what other drawn languages add to the six parts or leave out of the six parts, and what the axiomatic form gives a programmer. It closes by showing how the simplest teaching picture of a drawn program, the one given in my book *The Study of Pure Causality; On the Philosophy of Computer Science*, maps onto Halo's own notation. Its circles become Halo's circles, and its decisions become the two exits of every call. Its line of control becomes Halo's trigger line, and its line of data becomes Halo's data line. Every feature of Halo that the derivation uses is one that Chapters 5 to 12 show.

I. The Wish

For most of my working life I have wanted the same thing: to draw a diagram of a system, and to have the diagram be the program. The wish is for the program itself, as distinct from a picture of the program, or documentation that is out of date within the week. My book *The Study of Pure Causality; On the Philosophy of Computer Science* gives the wish its plainest form: "When I draw a diagram today I can export it as an image. I want a second entry on that menu that exports it as running code, and I want the two to be the same act". I met the same picture in Chapter 7: a drawing program can export a drawing as an image. The wish is that the same menu should offer a programming language, so that saving the drawing would produce the program.

II. The Axiomatic Form

A language for drawing programs is *axiomatic* when three things hold of it. Every element it has is required: take any one away, and some computation can no longer be represented. Nothing else is in it: every feature a programmer meets is one of the required elements, or that element used in a particular way. And the same elements cover every layer of compute, from a logic gate to a whole system, with no element reserved for any one layer. I call this the *axiomatic form*, because each element stands to the language as an axiom stands to a system: it cannot be derived from the others, it cannot be dropped, and everything else in the language is derived from the set. Chapter 7 called a function not axiomatic when its definition rests on something left unstated. A language is axiomatic when its drawings leave unstated nothing that compute requires, and state nothing that compute does not require. Chapter 5 calls the whole circle, with its pins and poles, the one element of Halo. In this definition an element is each thing in the language that can be counted and taken away, and for Halo the tables below count them: the six parts of that circle and its one rule.

By compute I mean what Chapter 5 means: the imperative model, which is the physical substrate of all computation done today. Halo has the axiomatic form, and the rest of this section shows it by deriving Halo from the notion of compute itself. The derivation starts from one question and asks what an answer to that question must state.

Chapter 5 gave the language one question: what happens when? More than one thing happens in a program, so a program must be divided into units, and a division is a boundary, with the definition of a unit inside it and the use of the unit outside it. The one symbol that marks a boundary is the circle.

For each unit, three things must be stated. *When* it happens must be stated, and the trigger states it. *What* it takes and gives must be stated, and the data states it. *Which* unit it is must be stated, and the instance states it. Chapter 5 drew the line between the last of these and the first: the call says which, and the trigger says when.

Each of the three crosses the boundary in two directions, in and out, and that gives six parts, as follows.

What must be stated	Crossing in	Crossing out
When	Trigger in: the unit begins.	Trigger out: what comes next.
What	Data in: the values the unit takes.	Data out: the values the unit gives.
Which	The instance called, at the zenith: a unit that comes from elsewhere is named.	The instance defined, at the nadir: a unit defined here goes out to be used.

The structure can be counted: one circle, two directions, three kinds of crossing, six parts. One rule completes it, and the rule joins when to what: a trigger entering pulls the values on the data inputs at one moment, and a trigger leaving pushes the values on the data outputs (Chapters 5 and 6). Chapter 5 stopped at this point and said that the language was finished, and that everything after that point is a specialization of this element.

Each of the six parts and the rule is required. Take any one away, and the loss is the one stated here.

Taken away	What can no longer be represented
Data in	A unit can use only what is fixed inside it, so no unit can compute on what another unit produced.
Data out	No result of a unit reaches anything outside it, because under the limit of Chapter 8, Section V, a value leaves a circle only through one of its data outputs.
Trigger in	Nothing can say when a unit begins, so the order of any two units cannot be stated.
Trigger out	Nothing can say what comes next. Sequence is lost. The conditional is lost, because a conditional is two exits from a call (Chapters 6 and 8). The ring is lost, because a ring is an exit wired back to an earlier entry (Chapter 6, Section IV).
The instance called, at the zenith	The drawing cannot say which instance is called, so two instances of one function cannot be told apart, and a function handed to another function cannot be called (Chapters 5 and 6).
The instance defined, at the nadir	A definition cannot produce an instance, so no function, callback or class can be defined in the drawing (Chapters 5, 6 and 8).
The rule of pull and push	The moment at which a value crosses the boundary is unstated, and two units that read and write one value race (Chapter 6, Section IV).

Two kinds of line is also the required number. Merge control into data, so that the order of operations follows the arrival of values, and the order of two effects that share no value cannot be stated: print these words, and then ring the bell (Chapter 6). A wire added to repair this, carrying order and no value, is a trigger line under another name. Keep control and take the data out of the drawing, as a flowchart does, and the values go somewhere the drawing cannot show. So the two lines are distinct, both are required, and no third kind of line has anything left to carry.

III. One Language, Up to Relabeling

Nothing else is in the language. Each thing that other languages supply as a feature is, in Halo, one of the six parts used in a particular way. The conditional is the two exits of a call (Chapters 6 and 8). The loop is an exit wired back to an earlier entry (Chapter 6, Section IV). The variable is a label at a pull or a push, or the simplest module, one that remembers the value it had the last time it was triggered (Chapter 7). The class is a factory, a circle whose output is an instance (Chapter 8). The callback is a definition drawn inside a circle, with its instance wired to a data input (Chapter 6, Section V). The join is a module with two triggers in, at which two threads meet (Chapter 6, Section V). The lozenge and the box are circles drawn short (Chapters 6 and 7). Halo has no keywords, no built-in constructs and nothing supplied in the box, and even the notion of logic is imported (Chapter 5). A diagram carries no linguistic tokens other than its constants (Chapter 7).

Halo's set of elements is minimal in the exact sense, which is minimal for full coverage. A language can be made smaller than a useful one by giving up what a person needs in order to read it, and the Turing machine and Brainfuck are the standing examples: each computes, and neither gives a legible description of its own structure, which Chapter 3 showed to be a condition of generality. Halo gives up nothing of that. It is the smallest set of elements with which every program of the imperative model can be drawn, read and checked.

The derivation in Section II used nothing about Halo. It used the question, the boundary, and the three things a unit must state. So the result holds of every language with the axiomatic form. Such a language has units with boundaries, because more than one thing happens. It has a mark for each of the six parts and a rule for when values cross, because the table of Section II shows what is lost without each. It has no further element, because a further element would be one of the six under another name, or one of the six used in a particular way, and the form admits neither as an element of its own. Such a language can differ from Halo

only in how its marks look and where they sit: a square in place of a circle, or the triggers on the other side. Every drawing in it corresponds to one Halo drawing, part for part and rule for rule. Such a language is Halo, drawn another way.

The axiomatic form therefore has one language in it, up to relabeling, and the claim rests on this derivation and on no survey of what has been built. In this book I name the axiomatic form, derive that form, and give Halo, the language that has that form. Halo has been drawn in this form since 1995.

IV. Every Layer, One Element

The same circle, with the same six parts and the same rule, is the unit at every layer of compute, as follows.

Layer	The one element at that layer
Logic gate	A NAND gate is a circle: a clock pulse enters, A and B are pulled, the result is pushed, and the pulse goes on. Each gate stamped out is an instance (Chapter 5, Section IV).
Primitive and operator	A function supplied from outside the language, by a library or by hardware, is a circle whose interior the drawing does not specify (Chapters 5 and 7).
Function	A function is a circle, defined at its nadir and called at its zenith (Chapter 5).
Module, class and table	A class is a factory circle whose instances are the records of a table, and a function of the module is a column (Chapters 8 and 9).
Program	A program is the outer circle, and the operating system that calls it is a leaf (Chapter 6, Section II).
Threads and callbacks	Several intake threads and several active output threads are the same symbols used as they are (Chapter 6, Section V).
Hardware	A drawing compiles to ANSI C, JavaScript or assembly language, to an FPGA or a GPU, or to circuits that represent the computation directly (Chapter 5; Chapter 12).
Whole system	When a whole system of circles is organized in third normal form, the drawing and the normal form of Part I are one structure (Chapter 8).

V. The Form and the Modalities

This section is for readers who want the ground of the axiomatic form in the Immanent Metaphysics; the engineering argument is complete without it.

The three kinds of crossing are the three modalities, and the six parts are their pairs. The frame of the assignment must be named, as for every modal assignment (Introduction, Section III). The frame here is one call, seen from the drawing. Each modality has its pair, two words for the two sides of what that modality does: sameness and difference for the immanent, content and context for the omniscient, subject and object for the transcendent.

Kind of crossing	Modality	Why, in this frame	The pair its two directions make
Trigger: when	Immanent	A trigger is an event, here and now, and what it does depends on the context in which it fires.	Sameness and difference: a trigger is a change of state at a leading edge, one state before it and another after. The trigger in starts from the state that has held the same, and the trigger out leaves the state the unit has made different.
Data: what	Omniscient	A value pulled is held the same at its pin, on every wire that reads it, at one moment.	Content and context: the value is the content, and the boundary at which it is held is its context. Data in is a value taken into the boundary as content, and data out is a value set on the boundary as context for whatever reads it next.
Instance: which	Transcendent	A definition is a possibility that is not bound to any one instance until a call names one.	Subject and object: the two sides of the boundary, the side that calls, at the zenith, and the side that is called, at the nadir.

Axiom III holds of the three kinds of crossing as a matter of the drawing. They are distinct: a trigger, a value and an instance are three different marks. They are inseparable: every call has all three, and a call that carries

no values, as Ring Bell carries none in hello world (Chapter 6), still has its crossing of data, with no pins on it: its trigger pulls nothing and pushes nothing. They are non-interchangeable: Section II showed what is lost when a line of values is made to do the work of a line of control.

Axiom I holds of them in the order of a call. The trigger is the most fundamental of the three: nothing is pulled and no instance is called until a trigger fires. Data and instance are in conjugate relationship: every instance reaches a zenith as a value on a wire, and every value is taken up by an instance. And each can be perfected only at the expense of the other: a drawing with every value fixed at its pin has no instance left open to choose, and a drawing with every instance left open has no value fixed.

Axiom II holds of them as a cycle. A class of the transcendent precedes an instance of the immanent: a definition at a nadir precedes any firing of that definition. A class of the immanent precedes an instance of the omniscient: a pattern of firing, set by the trigger wiring, precedes the particular values pulled and pushed. A class of the omniscient precedes an instance of the transcendent: the values a unit gives out precede a handle wired to the next zenith, as a factory's output is the instance that a later call names (Chapter 8).

The Method note of Chapter 1, Section II, assigns the modalities in another frame, that of a whole path of control, where sequentiality is omniscient, conditionality immanent and the ring transcendent. Modal assignments rotate under axiom two with the orientation of the observer, and the two assignments are one structure seen from two orientations.

VI. Other Drawn Languages

Several systems for drawing programs do real work in the domains they were built for. LabVIEW, released by National Instruments in 1986, is programmed in a graphical language called G, in which the block diagram is the program that runs; a node of the diagram runs when data is available on all its inputs, and the diagram is compiled to machine code (Kodosky

2020). Prograph, begun at Acadia University in 1982 and sold from 1989, is a visual, object-oriented dataflow language in which a method is a diagram of operation icons; data flows from the top of the diagram through its operations, and loops and branches are made by marking operations with annotations (Cox, Giles and Pietrzykowski 1989). Simulink, from MathWorks, draws a dynamic system as a diagram of blocks, chosen from a library of many kinds, for modeling and simulation, and with a companion product it writes C source code from the diagram (MathWorks). Node-RED, first released by IBM in 2013, wires nodes into flows through which messages pass, each node handing its result to the next; the flows are stored as text in the JSON format and run on Node.js (OpenJS Foundation). Scratch, from the MIT Media Lab (version 1.0 in 2007), is a language of blocks that its users, most of them young, arrange into scripts (Resnick and others 2009).

Each is a product for its domain, and each stands apart from the axiomatic form on two counts, the two of Section II: each states something that compute does not require, and each leaves unstated something that compute requires, as follows.

Language	What it adds to the six parts	What the drawing leaves unstated
LabVIEW	Many kinds of node, structure and annotation.	A line of control of its own: order follows the arrival of data.
Prograph	Many kinds of operation and annotation; loops and branches are annotations.	A line of control of its own: order follows the arrival of data.
Simulink	Many kinds of block, from a library.	A line of control separate from the signals.
Node-RED	Kinds of node, and a message that passes through them.	The separation of control from data: one line carries both at once.
Scratch	Many kinds of block, which draw the structure of a text.	Lines of either kind: the data is held in variables that are not drawn.

Models of computation written as text and as formal pictures each hold a part of what the six parts draw. Gilles Kahn's networks of processes joined by channels compute deterministically (Kahn 1974), as a Halo drawing of circles joined by wires does when it keeps the limits of Chapter 6. The synchronous languages Lustre and Esterel read their inputs and write their outputs in step with a clock (Halbwachs and others 1991; Berry and Gonthier 1992), as a Halo trigger pulls and pushes the values on its pins at one moment; Lustre underlies tools used to build and certify the control software of aircraft. Carl Adam Petri's nets draw control as tokens moving through a net (Petri 1962), and David Harel's statecharts draw a system as states nested within states (Harel 1987), as Halo's trigger lines can pass through named states (Chapter 7). And the hardware description languages, VHDL and Verilog, describe a circuit as modules wired together and run on a clock (IEEE 1988; IEEE 1996), which is what a Halo drawing is when it is carried into hardware (Chapter 5). Halo draws all of these parts with one symbol and two kinds of line.

Each of these systems runs on an imperative machine, so every program made in any of them can be drawn in Halo (Chapter 7). What each of them adds is a specialization for its domain, and what the axiomatic form states is the ground on which such specializations stand.

VII. What the Form Gives a Programmer

A programmer can recognize the axiomatic form by four properties, and I call a system for drawing a program *clean* when it has them. One symbol and two kinds of line, one for control and one for data, are enough to draw every imperative program. Every crossing of a boundary by control, where a trigger line ends at a triangle on the circle and its pulse enters or leaves, is a synchronization of the data. The drawing and the text of any imperative language translate into each other exactly, in both directions.

And a whole system of such drawings can be organized in third normal form. Chapters 5 and 6 give Halo the first two of these properties, Chapter 11 the third, and Chapter 8 the fourth.

Each property is there for what it gives a programmer. One symbol and two kinds of line let a drawing be read as two separate graphs, the flow of control with no values mixed into it and the flow of values with no control mixed into it (Chapter 6). Synchronization at every crossing of a boundary by control removes the race from the default program (Chapter 12, Section III). Exact translation in both directions (a drawing written out and read back is the same drawing, and a text read in and written out is the same algorithm, in the canonical form of its language; Chapter 7, Section II) makes the drawn language a Rosetta Stone among imperative languages, and it lets refactoring tools work automatically (Chapters 7 and 11). Organization in third normal form shows the points of articulation, the joints at which a system can be changed to take new functionality (Chapter 8). When a tool counts the symbols of the language, the pads, the triangles and the two poles are parts of the one element, the circle (Chapter 5), and the lozenge is a convenience that changes nothing in the language: holding a name on the data lines, it is the simplest module drawn short; holding a constant, it is a node that produces that constant, drawn short; and on the trigger lines it is a named state (Chapters 6 and 7). The box labeled OS is a leaf node, a circle whose interior the program does not specify, drawn as a box to show that it stands outside the program (Chapter 6, Section II). So the first property answers the question of how many kinds of element a tool must know, and the answer is one: the circle, with its parts, and with its leaf nodes drawn short.

Part II has shown each of these in Halo. Every program in the imperative model can be drawn in Halo, and whatever cannot be drawn in it is not code (Chapter 7). A Halo drawing can be written out into any imperative language and read back in from any, as a Rosetta Stone stands among scripts (Chapter 7). The graphic can become directly executable code (Chapter 7). And Halo's form in JavaScript is built so that the translation between diagram and code is exact in both directions: a change

made to the diagram is meant to update the code, and a change made to the code the diagram (Chapter 11). A drawing with those properties answers the wish of *The Study of Pure Causality*, that drawing a system and building it should be one act.

VIII. From the Teaching Picture to Halo's Notation

The Study of Pure Causality teaches the drawn program with a simpler picture, one that any reader can hold at once. It keeps the flowchart, which shows only control, and adds the half the flowchart is missing, which is data. In its words: "Every operation is a circle ... Every decision is a diamond ... Then draw a second kind of line. The first kind carries control: after this, that. The second kind carries data".

Halo is the full system of which that picture is the beginning, and each element of the picture has its place in Halo's own notation, as follows.

The teaching picture (The Study of Pure Causality)	**Halo's notation (this book)**
"Every operation is a circle".	Every function is a circle, the one symbol of the language, which marks the boundary between a function and whatever uses it. The program itself is the outer circle, and the functions it calls are circles drawn inside it (Chapters 5 and 6).
"Every decision is a diamond, with as many outgoing paths as it has outcomes, which for most decisions is two".	Every circle does the work of the flowchart's diamond (Chapter 8). Every circle has its own trigger outputs, and by convention exactly two, success and fail, so that every call is also a decision: the notion of a conditional and the notion of a function call are strictly isomorphic (Chapters 6, 8 and 11). Chapter 5 also gives a diamond that crosses the circle at its zenith: it is the instance being called, and it marks no decision.
"The first kind carries control: after this, that".	Trigger lines, drawn to small triangles (triggers in at the upper left of a circle, triggers out at the lower left), carry the flow of control from circle to circle. A trigger line never crosses the boundary of a circle; a trigger entering pulls every input at one moment, and a trigger leaving pushes every output (Chapters 5 and 6).

The teaching picture (The Study of Pure Causality)	Halo's notation (this book)
"The second kind carries data: this operation's output is that operation's input".	Data lines, drawn to small circles (inputs at the upper right, outputs at the lower right, numbered in order), carry the values. A data line may cross the boundary of a circle; each input takes one source, and an output may feed many (Chapters 5 and 6).
"Draw a variable as a small bubble. A bubble drawn inside a circle belongs to that circle".	A named variable stands at every pull and push. A variable defined inside a circle is local to it, and one defined in an enclosing circle can be read inside, which gives class variables (Chapters 7 and 8).
"Restrict how many control paths may enter or leave a region".	One way into each function, exactly two ways out, one destination for each trigger leaving a call, and a limit on the data lines that cross from inside a circle to outside (Chapters 6 and 8).
"If every region has one entrance and one exit, then every loop in the system can be identified".	With one way into each function, and one destination for each of its two exits (what single exit becomes in Halo), every loop path can be identified and given a counter with a fixed limit, so that every program written under the discipline halts by construction (Chapter 8).

Halo also carries what the teaching picture leaves out, because a drawing that is the program needs it. Each circle has two poles, the instance being called at its zenith and the instance being defined at its nadir, so that the drawing says which function is meant and where it came from (Chapter 5). A function may have any number of outputs, as it has any number of inputs (Chapter 5). And the pins keep their numbered order, because the numbers are the contract between the caller and the function it calls (Chapter 7).

IX. Two Pictures, One Discipline

The two pictures agree on the point that matters most. A flowchart shows control and leaves out data, and both the teaching picture and Halo draw the data as well. For this reason, Chapter 8 called Halo a superset of the flowchart. Both remove the written syntax of a programming language (its keywords, punctuation and grammar), which is the wall most people meet first, and leave the discipline, which is the real work. Chapter 7 named that work: assembling components so that given inputs become the required outputs reliably, a kind of meta-thinking, beside which the rest is plumbing.

The teaching picture is where a reader begins, with circles, diamonds and two kinds of line. Halo is where the picture arrives once it is made exact enough to run: one symbol, two kinds of line and a few rules at the boundary of every circle. And because every Halo program can be organized in third normal form (Chapter 8), the drawing of a program and the normal form of Part I are one structure.

X. Summary

Halo has the axiomatic form: every element it has is required for completeness of representation, nothing else is in it, and the same elements cover every layer of compute. The one question a program answers, what happens when, requires a boundary, and three things stated at each boundary: when, what and which. Each of the three crosses in two directions, and that gives one circle, two directions, three kinds of crossing, six parts and one rule, the rule of pull and push. Take any part away and some computation can no longer be represented, and every other feature a programmer meets is one of the six parts used in a particular way. Any language with the axiomatic form is therefore Halo drawn another way. The three kinds of crossing are the three modalities of the Immanent Metaphysics, in the frame of one call seen from the drawing. Other drawn

languages, LabVIEW, Prograph, Simulink, Node-RED and Scratch among them, add to the six parts or leave some of them unstated. A programmer recognizes the axiomatic form by four properties: every imperative program drawn with one symbol and two kinds of line, every crossing of a boundary by control a synchronization of the data, exact translation in both directions between the drawing and the text of any imperative language, and a whole system of drawings organized in third normal form. The teaching picture of *The Study of Pure Causality* maps onto Halo element for element. The operation's circle is Halo's circle, and the decision's diamond is the work every circle does with its two exits. The line of control is the trigger line, and the line of data is the data line. The variable's bubble is the named variable and its scope, and the limits drawn on regions are Halo's limits on wiring. Halo adds what a drawing must have to be the program: the two poles of every circle, outputs as many as inputs, and pins numbered as a contract. The drawing of a program, organized in third normal form, is the structure that Part I established for code, data and the architecture of a system.

KEY CONCEPTS IN THIS CHAPTER

- **Diagram as program**: a drawing that is the program itself, written out into code and read back exactly, as distinct from a picture of a program.
- **Axiomatic form**: a language for drawing programs has it when every element is required for completeness of representation, nothing else is in the language, and the same elements cover every layer of compute.
- **Six parts and one rule**: trigger in and out, data in and out, the instance called at the zenith and the instance defined at the nadir, with the rule that a trigger entering pulls and a trigger leaving pushes.
- **Clean system for drawing a program**: one in which one symbol and two kinds of line draw every imperative program, every crossing of a boundary by control is a synchronization of the data, drawing and text translate exactly in both directions, and a whole system of drawings can be organized in third normal form; the four properties by which a programmer recognizes the axiomatic form.
- **Teaching picture**: the drawn program of The Study of Pure Causality: circles for operations, diamonds for decisions, one line for control and one for data.
- **Trigger line**: Halo's line of control, drawn to small triangles, which never crosses a circle's boundary.
- **Data line**: Halo's line of values, drawn to small circles, which may cross a circle's boundary.

Summary

In this book I have answered one question: what single structure do code, data and the architecture of a system share, and how can a program be written in that structure directly? Its answer has two parts, the normal form and Halo, and a closing chapter that names the axiomatic form, which Halo has, and sets Halo beside the other ways people have drawn programs.

Part I established the structure. All software architecture can be represented in third normal form, and there is a strict one-to-one correspondence between a software module and a database table: a class is a table, an instance is a record, a function is a column and a pointer is a foreign key (Chapter 1). The mapping holds without exception, and a departure from it for performance is a departure from a known ideal. Normalization is compression, and compression at the level of architecture is the factoring of intentionality: the operation performed in factoring a system and the operation performed in factoring an understanding of the world are the same operation (Chapter 2). From that follows the first consequence: where a system represents intention and its factoring can be inspected, a community can verify that what runs is what it chose, and where the factoring is hidden, it cannot. Clarity, a correspondence between the divisions of a system and the divisions of what it represents, is what trumps complexity, where simplicity loses every arms race. A model of computation that means to be general must be able to describe its own structure in a form a person can read, and code organized in third normal form can (Chapter 3). And to call anything an instance of a class obliges one to say what is essential to the class and what is free to vary, because every instance is unique (Chapter 4).

Part II gave Halo, a language for writing a program in that structure directly, and Part II with Appendix A is the definition of the language. Halo has essentially one symbol, a circle, the boundary between a func-

tion and whatever uses it. Small circles carry data in and out, small triangles carry triggers, and a trigger entering pulls every input at one moment while a trigger leaving pushes every output; a diamond at the zenith is the instance being called, and a square at the nadir the instance being defined (Chapter 5). A few limits make a drawing read the way people think: one way into each function, exactly two ways out, data and trigger wires kept apart, one source for each input and one destination for each trigger leaving a call (Chapter 6). Whatever cannot be drawn in Halo is not code, and any classical program can be read into a Halo diagram and written out into any imperative language; names are a layer for people over a structure whose identity is the position of each pin and the unicity of each name (Chapter 7). Code drawn in Halo and constructed in third normal form is fully general to all system design. It shows its points of articulation, and code kept in that form never takes on technical debt (Chapter 8). In everyday code, a module is a table, a call names which table, which record and which method, and a running system can therefore be patched while it runs (Chapter 9). A code base lasts as long as people can read it and refactoring tools can rewrite it, and syntactic sugar is waste; Halo keeps its complexity as close as possible to what Turing completeness strictly requires because it is designed for a horizon of a thousand years, on hardware far weaker than ours or not yet invented (Chapter 10). In JavaScript, Halo is a module pattern and a single-array call geometry that translate into the diagram and back exactly, and the written RISL standard (Reduced Interaction Structuring of Language) gives every rule of that form, each following from the statement that the function of code is to document its function (Chapter 11). And Halo, carried down to hardware, gives a one-to-one relationship between what the author intends and what the hardware does, which is the path toward trustable computing (Chapter 12).

Chapter 13 named the axiomatic form: every element required for completeness of representation, nothing else in the language, and the same elements at every layer of compute. It derived one circle, two directions, three kinds of crossing, six parts and one rule from the question of

what happens when, and it showed that any language with that form is Halo drawn another way. Other drawn languages, LabVIEW among them, add to the six parts or leave some of them unstated. The simplest teaching picture of a drawn program maps onto Halo element for element, and the drawing of a program, organized in third normal form, is the structure Part I established.

The parts fit as ground and building. The normal form is the ground: one structure for code, data and architecture. Halo is the building raised on it: a way of writing in that structure from the first circle drawn, so that the structure is never lost and never has to be recovered.

Appendix A. The Rules of a Halo Drawing

This appendix gathers the rules of a Halo drawing in one place, for a reader who will build an editor, a translator or a checker for the language. Each rule is copied from the chapter that teaches it, and the chapter keeps its text; where a rule has its reason, the chapter gives it.

I. The Element

1. The language has one element, a circle, which marks the boundary between a function and whatever uses it; its parts are its pads, its triangles and its two poles, and the lozenge and the box are leaf nodes drawn short (Chapter 5, Section II; Chapter 6, Sections II and III; Chapter 13, Section VII).
2. Data inputs are small round pads just inside the upper right of the circle, and data outputs are pads just inside the lower right; a function may have any number of each (Chapter 5, Section II).
3. Triggers in are small triangles just inside the upper left, and triggers out are triangles just inside the lower left; the arrival of a pulse on a trigger in is the event of the function being called, and each trigger out is an exit (Chapter 5, Section II).
4. A diamond crossing the circle at its zenith is the instance being called; a square crossing it at its nadir is the instance being defined (Chapter 5, Section II).
5. Wires drawn outside the circle treat it as a function being called, defined somewhere else; wires drawn inside a circle large enough to fill the page treat the drawing as the definition of the function. One drawing never shows both a definition and a call of the same circle (Chapter 5, Section II).

6. A pad carries no type. The type of a value is documented where what the value is for is documented (Chapters 5 and 7).
7. Pads, triangles and the two poles are parts of the one element (Chapters 5 and 13).

II. Numbering

1. Every pin is numbered in order, starting from zero (Chapter 5, Section III).
2. Data pins are numbered clockwise: data inputs from the zenith, data outputs from the three o'clock point (Chapter 5, Section III; Figure 5.5).
3. Trigger pins are numbered counterclockwise: triggers in from the zenith, triggers out from the nine o'clock point (Chapter 5, Section III; Figure 5.5).
4. Pins keep their numbered order, because the numbers are the contract between the caller and the function it calls; names given to pins are for people (Chapter 7, Section II).

III. Timing

1. A trigger entering a circle pulls the values on all its data inputs at one moment, on the trigger's leading edge (Chapter 5, Section III).
2. A trigger leaving a circle pushes the values on all its data outputs; when an exit trigger fires, all the data outputs are set (Chapter 5, Section III).
3. Every crossing of a boundary by a trigger is a synchronization of the data (Chapter 5, Section III).

4. A value on an output is available to an input only once the trigger has left the circle that wrote it, and the next circle pulls it when its own trigger enters (Chapter 6, Section III).
5. Every exit pushes the values on the data outputs, the fail exit as well as the success exit; on a fail exit the caller trusts only the outputs registered to describe the failure (Chapter 6, Section I).

IV. Lines

1. There are two kinds of line: trigger lines, which carry control, and data lines, which carry values (Chapter 6).
2. Trigger lines and data lines are never joined: no input or output is wired to a trigger line, and no trigger line to an input or output (Chapter 6, Section I).
3. A trigger line never crosses the boundary of a circle, its membrane; a data line may (Chapter 6, Section II).
4. A data line that crosses a membrane without meeting a pin carries a global value, a value from outside the program, which has whatever state it holds at the moment a trigger pulls it; a data line connected to a numbered pin is synchronized with the triggering of that circle (Chapter 6, Section II).
5. Lines that cross on the page do not connect. A line connects only at a pin, at a pole or at a lozenge, and fan-out is drawn as several lines leaving one pin (Chapter 6, Section IV; Figure 6.12 (a)).
6. Every data input of a circle that is called is wired. A drawing in which such an input has no wire is incomplete, and a checker reports it (Chapter 6, Section IV; Chapter 7, Section I).

V. Fan-In and Fan-Out

1. A data input reads from one source; a value that must come from one of several sources passes through a named variable, as rule 5 of this section gives (Chapter 6, Section IV).
2. A data output may write to any number of places (Chapter 6, Section IV).
3. A trigger input may take any number of trigger lines (Chapter 6, Section IV).
4. A trigger output goes to one place. A trigger output wired to several places is a multithreaded program, which the general language can draw and the default program does not use (Chapter 6, Sections IV and V).
5. With a named variable at every pull and push, every pin has a fan-in and fan-out of one, and the variable takes fan-in from any number of places and fans out to any number (Chapter 7, Section II).

VI. The Limits Accepted for Readability

1. One way into each function (Chapter 6, Section I).
2. Exactly two ways out, success and fail, also called normal and signal or true and false, and in code the return values 1 and 0; fail tells the caller not to trust the outputs (Chapter 6, Section I).
3. Every limit is a choice laid over a general language that can still draw parallel code, loops, callbacks and functions defined inside functions (Chapter 6, Section V).
4. The fail exit says that a failure occurred, and in code it is one bit: 0 is the only value that means it. Which failure it was is a data output of the call, registered with its other outputs, so that no caller ever makes

a second call to learn the result of the first (Chapter 6, Section I; *The RISL Standard; Halo Written as Code, Across Languages*, Chapter 1, Section V).

VII. Instances and Primitives

1. Every circle that is called needs an instance from somewhere, a library or the operating system, and the line at its zenith says where (Chapter 5, Section IV; Chapter 6, Section II).
2. Primitives, such as a NAND gate, are defined outside the language and supplied to it, and everything else is built from them (Chapter 5, Section IV).
3. A function defined inside another has a square at its nadir, and its instance can be passed as data to a third function, which may call it: a callback (Chapter 6, Section V).
4. Code that makes code is drawn the same way: the code it makes is an instance that arrives at a zenith as a value on a data line (Chapter 7, Section I).
5. The box labeled OS is a leaf node: a circle whose interior the program does not specify, drawn as a box to show that it stands outside the program and supplies instances and values to it (Chapter 6, Section II).

VIII. Constants, Named Variables and Named States

1. A constant is drawn as a lozenge, a long box with rounded ends, with its value inside (Chapter 6, Section III).
2. A named variable is drawn as a lozenge with its name inside, and stands at every pull and push; the simplest module, which remembers the last value it carried, is what it stands for (Chapter 7, Section II).
3. A named state is a lozenge on the trigger lines naming where the program is, such as start (Chapter 7, Section II).

4. A name carries the prefix of its kind of symbol (Table 7.1), is written with underscores and no upper case in the name of any symbol, and ends in a version suffix (Chapter 7, Section II; Chapter 11, Section V).
5. A drawing with every label taken away is still the same program; identity stays in the structure, as the position of each pin and the unicity of each name (Chapter 7, Section II).
6. Where a lozenge is drawn tells its use: a lozenge on the trigger lines is a named state; a lozenge on the data lines holding a value written in quotation marks or as a numeral is a constant, a node that produces a constant value; and any other lozenge on the data lines is a named variable (Chapter 7, Section II).

IX. Scope, Factories and Tables

1. A named variable defined inside a circle is local to that circle and cannot be reached from outside it; a variable of an enclosing circle can be read inside when it is passed in, which means wired to a data input of the inner circle, so that the inner circle pulls it when its trigger enters. No data line runs from inside a circle across its membrane to the outside; a value leaves a circle only through one of its data outputs (Chapter 8, Section V).
2. A class is drawn as a factory, an outer ring whose every call makes a new instance of the item inside it (Chapter 8, Section III).
3. The instances a factory makes are kept in a table, and a call names three things: which table, which record, which method in that record (Chapter 8, Section III; Chapter 9, Section III).
4. A table is sized to a known, generous ceiling; when a module needs one more instance than its table reserves, the call that asks for it returns through its fail exit (Chapter 3, Section VI).

X. Loops and Recursion

1. Every loop has a counter with a fixed limit, and with one way in and one way out every loop path can be found and bounded, so that every program written under the discipline halts by construction (Chapter 8, Section VII).
2. Under one destination for each trigger leaving a call, a trigger reaches a circle that is still running only through a call made from inside it, and that call names a new instance from the function's table, so that the table's ceiling bounds the depth of recursion (Chapter 8, Section VII).

XI. What Counts as Code

1. Whatever cannot be represented in this diagram is not code (Chapter 7, Section I).
2. The translation between a drawing and the text of any imperative language is exact in both directions: a drawing written out and read back is the same drawing, and a text read in and written out is the same algorithm, in the canonical form of its language (Chapter 7, Section II; Chapter 11, Section II).

XII. What the Module Defines, and What Other Standards Define

1. A module may have several trigger outputs, and the module designates what each one means: one may mark an error while three others mark different kinds of success. Two exits, success and fail, is the default that the limits of Chapter 6 adopt for readability (Chapter 6, Section I).

2. Several intake threads into one module, and several active output threads from one module, are special uses of the same symbolism. Nothing in the drawing prevents them, and the module itself defines how it works; the symbols are used as they are (Chapter 6, Section V).
3. A starting value on a wire is provided by the operating system, as a leaf node, or by a node whose job is to produce a constant value and for which no interior is specified (Chapter 6, Sections II and III).
4. Halo defines how a program is drawn. How a drawing is stored as a file is a separate standard, a colleague to Halo; and how a drawing is written as code in C, in assembly, in JavaScript or on an FPGA is the RISL standard, which is Halo implemented in code across each language (Chapter 9, Section I; *The RISL Standard; Halo Written as Code, Across Languages*).
5. In the general language, what a circle does with a second trigger that arrives while it is still running is defined by the module, as every use of several intake threads is; in the default program a trigger reaches a running circle only through a call made from inside it, which names a new instance (Chapter 6, Section V; Chapter 8, Section VII). A join is a module with two triggers in that fires its exit once a trigger has arrived on both, and two threads that must wait for each other each wire their exit to one trigger in of a join (Chapter 6, Section V; Figure 6.19).

XIII. The Flowchart Version

1. The drawing closer to a flowchart is a reading aid derived from the circle form, with no rules of its own: each of its circles is a call of the circle form, each line running down the page a trigger line from a success exit, each dashed line a trigger line from a fail exit, and each named value written beside a call a named variable at a pull or a push. A tool reads and writes the circle form, and draws the flowchart version from it (Chapter 11, Section IV; Figure 11.1).

Appendix B. Hello World, Drawn and Written

This appendix carries one program, hello world, from its drawing to its complete text, so that a reader can check, element by element, that the translation between the two is exact in both directions. The drawing is Figure 8.6, repeated as Figure B.1, and the text is Listing B.1, written to the RISL standard for JavaScript (Chapter 11, Section V; *The RISL Standard; Halo Written as Code, Across Languages*, Chapter 5), with the shared namespace cc.

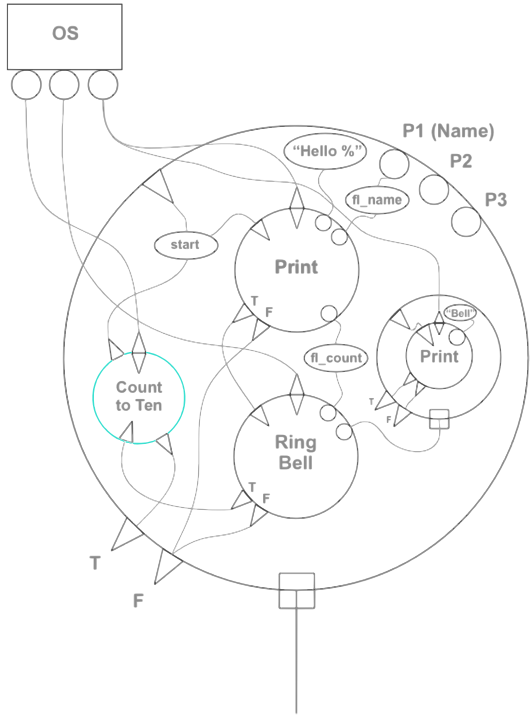


Figure B.1. Hello world, the drawing of Figure 8.6: the operating system's box supplies Print, Ring Bell and Count to Ten; the program's trigger in reaches the named state start, which triggers Print; Print's success triggers Ring Bell, and Ring Bell's success triggers Count to Ten, which sends the loop around to start again until it reaches ten, and then to the program's exit T; every fail runs to the program's exit F. The constant "Hello %" and the named variable fl_name, from the first parameter P1, feed Print; Print's output, through fl_count, feeds Ring Bell, which also takes the callback, the function defined at the right, as data.

```
// hello world, as drawn in Figure B.1
cc.helo_fn_world_1 = function (f)
   {
   // P1 (Name), which arrives with the trigger in
   var fi_name_1 = f[0];
   try
      {
      if (!fi_name_1) return cc.fn_trap("I_HWN_AME_Q");
      var fl_name_1   = fi_name_1;     // the named variable fl_name
      var fl_format_1 = "Hello %";     // the constant, in its lozenge
      var fl_count_1  = 0;             // the named variable fl_count
      var fl_limit_1  = 10;            // Count to Ten
      var fl_i_1      = 0;

      // the callback: the function defined inside the program
      var fn_bell_1 = function (f)
         {
         try
            {
            // its constant, in its lozenge
            var fl_text_1 = "Bell";
            f[0] = fl_text_1;
            if (!cc.osys_fn_print_1(f))
               return cc.fn_trap("I_CBP_RNT_Z");
            var fl_written_1 = f[0];
            return 1;
            }
         catch (fl_err_1) { return cc.fn_trap("I_CBK_ERR_J"); }
         };

      for (; ; fl_i_1++)
         {
         // Count to Ten: the bound, tested at the top
         if (fl_i_1 >= fl_limit_1) break;

         // start: the named state, which triggers Print
         f[0] = fl_format_1;
         f[1] = fl_name_1;
         if (!cc.osys_fn_print_1(f)) return cc.fn_trap("I_PRN_TWO_L");
         fl_count_1 = f[0];            // Print's output, into fl_count

         // Print's success triggers Ring Bell
         f[0] = fl_count_1;
         f[1] = fn_bell_1;             // the callback, passed as data
         if (!cc.osys_fn_ring_bell_1(f))
            return cc.fn_trap("I_RBL_ELL_V");
         }
      return 1;                        // the program's exit T
      }
   catch (fl_err_1) { return cc.fn_trap("I_HWO_RLD_C"); }
   };
```

```
// the operating system calls the program:
// its trigger in, with P1 in f[0]
var ml_f_1 = [];
ml_f_1[0] = "World";
var ml_ok_1 = cc.helo_fn_world_1(ml_f_1);
```

Listing B.1. Hello world, the program of Figure B.1, in JavaScript written to the RISL standard. Every return of a trap is the drawing's exit F, and the return of 1 its exit T.

Each element of the drawing has one place in the text, and each line of the text one element of the drawing.

1. The program's circle, defined on the page, is the function cc.helo_fn_world_1, and the square at its nadir is the name under which that function is stored.
2. The program's trigger in is the call of the function, and its two exits are its returns: 1 for the exit T, and the return of a trap for the exit F.
3. The first parameter, P1, arrives in f[0] and is pulled into fi_name_1 at the top of the function, and from there into the named variable fl_name_1.
4. The constant "Hello %" in its lozenge is fl_format_1, and the constant "Bell" inside the callback is fl_text_1.
5. Print and Ring Bell, whose instances the operating system's box supplies at their zeniths, are the functions cc.osys_fn_print_1 and cc.osys_fn_ring_bell_1, supplied from outside the program.
6. Each call loads its inputs into f by position, and each output is recovered from f into a named variable: Print's output into fl_count_1, which is the named variable fl_count.
7. Each fail exit wired to the program's exit F is a test of the call followed by the return of a trap.
8. The named state start is the top of the body of the loop, from which Print is triggered.

9. Count to Ten, a counter whose instance the operating system supplies in the drawing, is written as the counted ring of the standard, the for loop whose bound is tested at the very top of its body (Chapter 11, Section V); while the count is below ten the loop goes around to start, and at ten it leaves to the exit T.

10. The callback, the function defined inside the program with its own square at its nadir, is fn_bell_1, and its instance is passed as data to Ring Bell's second input.

Read in the other direction, the listing gives the drawing back: each function defined is a circle with a square at its nadir, each call a circle with its instance at its zenith, each named variable and constant a lozenge, each test of a call a pair of exits, and the loop with its bound a counter on the path back to start.

Terms

The terms of art of the book, each with the chapter in which it is defined.

Amdahl's law (Chapter 6): the part of a computation that must run in sequence limits the speedup that more processors can give.

ANSI C (Chapter 5): the C programming language as standardized by the American National Standards Institute.

Architectural identity (Chapter 2): class to table, instance to record, function to column, pointer to foreign key; the mapping is exact and admits no exceptions.

Architecture astronauts (Chapter 2): Joel Spolsky's name for engineers who add elaboration for its own sake, which hides the underlying primitives instead of making them plain.

ASM.js (Chapter 11): a strict subset of JavaScript designed to be compiled to run at close to the speed of machine code.

Axiomatic form (Chapter 13): a language for drawing programs has it when every element is required for completeness of representation, nothing else is in the language, and the same elements cover every layer of compute.

Axioms (Introduction): the three axioms of the Immanent Metaphysics, which state how the modalities relate: process is more basic than structure or possibility, and structure and possibility are a conjugate pair; the three follow one another in a fixed cycle (axiom two); and no one of the three can be reduced to the others, occur without the others, or be substituted for another.

Bernstein's theorem (Chapter 1, Chapter 3): every collection of tables, with the dependencies among its fields, can be decomposed into third normal form without losing any fact and without losing any dependency (Bernstein 1976); with the construction of Chapter 1 it carries every architecture to third normal form.

Bounded loop (Chapter 8): a loop with a counter and a fixed limit; with single entry and single exit, every loop can be found and bounded, so every program written this way halts.

Boxing (Chapter 10): the Java 1.5 feature by which a symbol's type can be known only by parsing another file.

Brainfuck (Chapter 3): the minimalist esoteric programming language invoked as a comparison; argued to presuppose more built-in structure, an addressable cell array and loop constructs, than a bare Turing machine provides.

Callback (Chapter 6): a function defined inside another and passed to a third, which may call it.

Call geometry (Chapter 11): every function takes one array f; inputs and outputs pass through it by position, and the return value of 0 or 1 selects the caller's branch.

Call triple (Chapter 9): the three things a call names: which table, which record, which method in that record.

Canonical form (Chapter 11): the one form of each construct that the written RISL standard allows, so that tools can edit, analyze and refactor it without guessing; for a module, Listing 11.3.

Carrington event (Chapter 10): a geomagnetic storm as strong as that of 1859, after which the computing power available to an ordinary person would probably drop by a factor of a thousand.

Church-Turing thesis (Chapter 1): the foundational result of Alonzo Church and Alan Turing (1936), asserting that every function that can be calculated by an effective procedure can be computed by a Turing machine; with the structured program theorem, it means that any language implementing sequentiality, conditionality and rings (or equivalent constructs) can represent any computable algorithm.

Clarity (Chapter 2): a correspondence, as distinct from a quantity: the divisions of a system match the divisions of what it represents, so that the system is understood the way the world is understood; clarity trumps complexity where simplicity loses.

Class (Chapter 4): a type specification, such as a blueprint, that exists nowhere in particular but defines the pattern any instance of it must follow.

Clean system for drawing a program (Chapter 13): one in which one symbol and two kinds of line draw every imperative program, every crossing of a boundary by control is a synchronization of the data, drawing and text translate exactly in both directions, and a whole system of drawings can be organized in third normal form; the four properties by which a programmer recognizes the axiomatic form.

Closed source as concealed intent (Chapter 2): where source is private, the system's intentionality is unavailable for inspection by those it affects.

Code colonialism (Chapter 2): the imposition of an uninspectable system upon a community, structurally identical to cultural colonialism.

Composition over inheritance (Chapter 11): building by containment, inclusion and function composition, the use that third normal form suits.

Conjugate pair (Introduction): two things of which each can be perfected only at the expense of the other, as structure and possibility are.

Contact space (Chapter 2): the middle bank of a switching arrangement; normalization reduces it from about the number of entries multiplied by itself to about the number of entries multiplied by its logarithm, as a multistage switching network does (Clos 1953; Beneš 1964).

Content, context, choice (Introduction): the three aspects of any interaction: what is given, compared or acted on; the setting that gives it its relations; and the act of the one who takes part.

Currying (Chapter 8): turning a function of several arguments into a chain of functions, each taking one argument.

Data line (Chapter 5, Chapter 13): a line of values, drawn to the small circles of a Halo element; it may cross the boundary of a circle.

Diagram as program (Chapter 13): a drawing that is the program itself, written out into code and read back exactly, as distinct from a picture of a program; exactly in the sense of the entry Exact in both directions.

EGS (Extended Grammatical Structuring) (Chapter 9, Chapter 11): my standard for writing notes, in which each statement begins with a dash and ends with a period, opens with where, that or as, and takes its context from its indentation; a substandard of RISL. It is set out in full in *The ITAG and EGS Standards; Structured Plain Text for Files and Notes*, Chapter 2.

Equivalence thesis (Chapter 1): the claim that all software architecture can be represented in third normal form, via a one-to-one mapping between modules and tables, and between inter-module references and foreign keys.

Essential versus accidental properties (Chapter 4): the distinction, forced into practical precision by software design, between what a class specifies as binding and what it leaves free to vary among its instances.

Exact in both directions (Chapter 7): said of the translation between a Halo drawing and the text of an imperative language: a drawing written out and read back is the same drawing, and a text read in and written out is the same algorithm, in the canonical form of its language.

Exfiltration, infiltration, modification (Chapter 12): the three threats to the integrity of a program: the theft of data; the insertion of a Trojan horse that makes a system do someone's bidding rather than its author's; and a change to an author's code, so that what happens is not what the author wanted.

Factoring intentionality (Chapter 2): the identification of architectural compression with the cleaving of understanding; the central identification of Chapter 2.

Factory (Chapter 8): a class drawn as an outer ring whose every call makes a new instance of the item inside it; in code, the table part of a module.

Factory factory (Chapter 8): a factory that makes factories and stores each instance it makes in a vector, at a place an incrementer gives, so that the instances form a table of modules.

Fan-in and fan-out (Chapter 6): how many wires arrive at a pin and how many leave it; a data input takes one source, a data output may feed many, a trigger input may take many, and a trigger output goes to one place.

Fidelity to a class (Chapter 4): the degree to which an instance must conform to its class's specification in order to still count as an instance of it, rather than of some other pattern.

Foreign key (Introduction, Chapter 1): the field of one record that names a record of another table.

FPGA (field-programmable gate array) (Chapter 5): a chip whose logic circuits can be rewired after it is made.

Fungibility (Chapter 7): the property that one part can be exchanged for another that does the same work.

Global value (Chapter 6): a value on a data line that crosses a membrane without meeting a pin; it comes from outside the program and is not necessarily synchronized with the call.

GPU (graphics processing unit) (Chapter 5): a processor built to carry out many calculations in parallel.

Grouping (Chapter 5): the third of the three concepts of computation when they are named sequence, conditionalization and grouping: the ability to form rings of sequence, named functions that can be called, and instances of data objects.

Halo (Chapter 3, Chapter 5): the model of computation and visual programming language of this book: essentially one symbol, a circle with data pins, trigger pins and two instance poles, which can express any imperative program; an expression of an underlying metaphysics, and the language with the axiomatic form (Chapter 13).

Halo element (Chapter 5): the one symbol of the language, a circle marking the boundary between a function and whatever uses it.

Halting problem (Chapter 8): the question whether a given program will ever stop on a given input; Alan Turing proved that no program can answer it for every program.

Hot patching (Chapter 9): changing the code of a running system without stopping it.

Idempotent all the way down (Chapter 9): an operation is idempotent when doing it many times gives the same result as doing it once; said of a system whose data is always flushed to disk, so that a restart at any moment brings it back in the same state, and starting it once or many times from what is on disk always gives the same state.

Idiom (Chapter 9): a visual pattern that a developer comes to recognize as a pattern.

Imperative model (Chapter 5): the model of programming in which the program tells the computer what to do; the physical substrate on which all computation in the world is done today.

Incommensuration Theorem (Introduction): symmetry and continuity cannot both be applied, simultaneously and fundamentally, to any event or comparison; and neither can asymmetry and discontinuity. The Incommensuration Theorem is a structural fact about the nature of knowing.

Instance (Chapter 4): a particular, concrete realization of a class, such as an actual house built from a blueprint, occupying a specific place and made of specific materials.

Instantiation (Chapter 4): the act or process of producing an instance from a class; always dependent on the context in which it takes place.

Intentionality (Introduction, Chapter 2): what the people who make and use a system mean it to do.

Intermediate language (Chapter 10): a language that sits between two others, as Halo sits between any two languages kept to the RISL conventions, and as the abstract syntax tree sits inside a compiler.

Intermediate variable (Chapter 7): a named variable placed at every pull and push, so that the mapping from pin to name is always explicit.

ITAG convention (Chapter 9): the name of the convention of four upper case letters and a colon set against the left edge of a file, as a marker for version notes, design notes, logs and abstracts kept within the file; the last tag of every file is ENDF. It is the RISL layout of a file. It is set out in full in *The ITAG and EGS Standards; Structured Plain Text for Files and Notes*, Chapter 1.

J2SE (Java 2 Platform, Standard Edition) (Chapter 10): the name under which version 1.5 of Java was released in 2004, as J2SE 5.0.

Join (Chapter 6): a module with two triggers in that fires its exit once a trigger has arrived on both; the synchronization wall at which two threads meet.

K matrix (Chapter 2): the worst-case connectivity in which every element of one set may connect to every element of another (what graph theory calls a complete bipartite graph); the thing normalization avoids.

Leading edge (Chapter 5): a term from electronics: the moment the voltage on a wire to a chip's pin changes, and that change of voltage signals the chip that something is ready to be done.

Leaf node (Chapter 6): a circle whose interior the program does not specify, which supplies instances or values to the program; the box labeled OS is one, drawn as a box to show that it stands outside the program, and a constant's lozenge is another, drawn short.

Legible self-description condition (Chapter 3): the requirement that a model of computation claiming full generality must be able to represent its own architecture, within itself and in a form a person can read and reason with.

Lifecycle functions (Chapter 11): fn_init_1, fn_term_1, fn_link_1 and fn_kill_1, which carry a module's state across a reload while the system runs.

Linear compiler design (Chapter 12): a compiler in which each element of the diagram becomes one element of machine code, in order, so that a reader can check the one against the other; Table 12.1 works the call to Print in hello world through it.

Load and recover (Chapter 11): before every call each input is loaded into the call array f from a named variable, and after it each output is recovered from f into a named variable.

Lozenge (Chapter 6, Chapter 7): the drawn symbol for a constant, a named variable or a named state: a long box with rounded ends, with the value or the name written inside. On the trigger lines it is a named state; on the data lines, holding a value written in quotation marks or as a numeral, it is a constant; any other lozenge on the data lines is a named variable.

Man, machine, nature triad (Chapter 2): the claim that humanity and technology cannot be stabilized as a two-pole system, and that an independent third pole, the natural world, is required to prevent either of the other two from dominating.

Membrane (Chapter 6): the boundary of a circle; data lines may cross it, trigger lines may not.

Microscopic, mesoscopic, macroscopic (Chapter 1): the three levels of software organization: the level of individual operations, the level of named modules and the level of system architecture.

Modalities (Introduction): three roles that anything under consideration can play: the immanent (process, what happens here and now and depends on its context), the omniscient (structure, what holds the same regardless of position) and the transcendent (possibility, what might become). A modal assignment says which role a thing plays within a given frame.

Module pattern (Chapter 11): a function that receives one record r, attaches public methods as r dot name, keeps private data in var declarations, and returns r; in the canonical form of the written standard (Listing 11.3) the constructor takes the array f like every function, creates the record r, places it in f[0] and returns 1.

Multipolar trap (Chapter 2): a situation in which each party, acting in its own interest, drives all of them toward an outcome none of them wants.

Multivariate convention, multivariate output (Chapter 2, Chapter 5): treating functions as having multiple inputs and multiple outputs, which allows any call signature to be regularized into fields of a record.

Named state (Chapter 7): a lozenge on the trigger lines naming where the program is, such as start, so that a program can be drawn as a state machine.

NAND gate (Chapter 5): a logic gate whose output is false only when both inputs are true; every other logic circuit can be built from it.

Necessity and sufficiency (Chapter 3): legible self-description is necessary for generality, and is argued directly; the claim of sufficiency, that the normal form covers every architectural form, rests on the construction of Chapter 1.

Normalization as compression (Chapter 2): third normal form factors out redundancy exactly as file compression does, with the same limit.

NoSQL database (Chapter 8): a database that stores data without the relational model's tables and joins, essentially one big table of associative relationships; on the teaching ladder, the second rung.

Not code (Chapter 7): whatever cannot be represented in a Halo diagram; its definition is incomplete or undefined.

Occult intentionality (Chapter 2): intentionality that is hidden from inspection, as in closed-source code.

Ordinal position (Chapter 7): the number of a pin, which is the contract between caller and callee.

Point of articulation (Chapter 2, Chapter 8): a natural joint at which a representation should be divided; in a design, a place where a refactoring makes it possible to add functionality, as a joint in the arm makes it possible to bend there.

Primitive (Chapter 5): a function defined outside the language and supplied to it, such as a NAND gate supplied by a library or by hardware.

Principal-agent problem (Chapter 2): how one party can make another, who acts on its behalf, act in its interest.

Process (Chapter 1): used both in computer science (a running program) and in the Immanent Metaphysics (the dynamic of change across content, context and choice); the two uses are structurally identical.

Pull and push (Chapter 5): a trigger entering a circle pulls the values on all its data inputs at one moment; a trigger leaving pushes the values on all its data outputs.

Reduced subset (Chapter 11): the JavaScript of the written RISL standard, fixed at Node.js version 8 and without the conveniences added since, so that every construct has one canonical form.

Refactoring tool (Chapter 10): a program that rewrites code without changing what it does.

Reflection (Chapter 10): a program's inspection and change of its own classes while it runs; in Java, hot patching through reflection is just barely possible.

Regress of trust (Chapter 2): hardware specification, library interface, compiler and inspection tool, each requiring trust in the layer beneath it.

Reification (Chapter 2): the making of a relationship or a process into a thing that can be stored and passed.

REPL (read-eval-print loop) (Chapter 9): an interactive prompt that runs one statement at a time and prints its result.

RISL conventions (Chapter 9, Chapter 11): the Reduced Interaction Structuring of Language, my coding standard, which sets out for each language how my designs are written in that language and which idioms to use; for Halo, the translation of its concepts into each language, with idioms chosen so that the human brain can parse them. Code kept to the conventions can be translated from any language to any other. RISL is effectively Halo implemented in code across each language, and the bootstrap to working in third normal form; it is set out in full in *The RISL Standard; Halo Written as Code, Across Languages*.

Rosetta Stone language (Chapter 7): a language into which any imperative language can be translated and out of which any can be written.

Rules of a Halo drawing (Appendix A): the rules of the language, gathered in one place, each with the chapter that teaches it.

Sequentiality, conditionality, rings (Chapter 1): the three necessary and sufficient elements of computation: ordered steps, branching based on state, and returns to a prior point in the sequence, which, joined with a condition, repeat a block of operations until the condition is met.

Side channel (Chapter 12): a measurement, from outside, of how the hardware carries out a program (its timing, its power, what it radiates), from which a value can be inferred.

Simplest module (Chapter 7): a module with one data input and one data output whose trigger and data pass straight through, and which remembers the last value it carried; it draws a local variable or a place on a stack.

Six parts (Chapter 13): trigger in and out, data in and out, the instance called at the zenith and the instance defined at the nadir, with the rule that a trigger entering pulls and a trigger leaving pushes.

SQL (Structured Query Language) (Chapter 8): the standard language of relational databases.

State machine (Chapter 1): a system that occupies one of a set of states at any given moment, and moves between states by input and rules; on any machine with finite memory the set is finite, and in any case a table may hold as many rows as the function has states.

Structured programming (Chapter 6): the discipline that a function has one way in; the first limit Halo accepts by choice.

Structured program theorem (Chapter 1): the result of Böhm and Jacopini (1966) that sequence, selection on a condition and repetition until a condition holds suffice to compute every computable function.

Success and fail (Chapter 6): the two trigger outputs of every function, by which a caller knows whether to trust its outputs (also called normal and signal, or true and false; in code, the return values 1 and 0). The fail signal is one bit, and which failure it was is a registered data output of the call; the general language lets a module have more exits, each with the meaning the module designates.

Symmetry, continuity (Introduction): symmetry is a sameness of content across a difference of context; continuity is a sameness of content across a sameness of context; asymmetry and discontinuity are their opposites.

Syntactic sugar (Chapter 5): notation added to a language for convenience that adds no power.

System catalog (Chapter 3): a table, stored and normalized like any other, that describes the structure of a database's own tables; the ordinary precedent for architectural self-reference.

Table of instances (Chapter 8): a vector of the instances a factory has made, through which each can be addressed, changed and returned.

Teaching ladder of normal forms (Chapter 8): a network of data, a table, linked tables joined by keys, called the first, second and third rungs; a picture of what normalization does, beside the textbook definitions of Chapter 3.

Teaching picture (Chapter 13): the drawn program of The Study of Pure Causality: circles for operations, diamonds for decisions, one kind of line for control and one for data.

Technical debt (Chapter 8): the accumulated cost of changes made where the design did not allow for them.

Third normal form (Chapter 1, Chapter 3, Chapter 8): the standard of well-organized relational data, in which every non-key field depends on the key, the whole key, and nothing but the key, so that each fact is stored in exactly one place; applied to code, the arrangement in which every module is a record in a table of modules of its kind, linked by keys.

Thousand-year horizon (Chapter 10): the scale on which Halo and the RISL conventions are designed: software that runs on hardware not yet invented, or a pale fraction of today's, and so depends on almost nothing.

Three hard problems (Chapter 12): multithreaded programming, cryptography and compiler design.

Transpile (Chapter 3): to translate source code from one programming language into another.

Trap identifier (Chapter 11): the USI number in the form I_XXX_XXX_X, unique across a code base, returned and logged at the point of every failure.

Trigger line (Chapter 5, Chapter 13): a line of control, drawn to the small triangles of a Halo element; it never crosses the boundary of a circle.

Trojan horse (Chapter 12): hidden code that makes a system do someone's bidding rather than its author's.

Trustable computing (Chapter 12): computing in which the hardware carries out, one to one, the intention represented in a diagram its author can read.

Trusted compute hardware (Chapter 12): hardware sealed by its maker, such as the Secure Boot of UEFI with a Trusted Platform Module chip; trusted only to the maker.

Trust equation reified (Chapter 2): the mapping of competence, intimacy and reliability onto connection to self, other and world respectively, with self-interest read as the degree of orientation toward a surface rather than a deep sense of self.

Trusting-trust problem (Chapter 2): the argument that even full access to source code cannot guarantee trustworthy behavior, since the compiler translating that source could itself be compromised.

Turing complete (Chapter 1, Chapter 5): able to simulate a Turing machine; an algorithm represented in one Turing-complete language can be faithfully translated into any other.

Turing machine (as impoverished model) (Chapter 3): the tape-and-state-table model of computation, technically universal but unworkable as a structure for human architectural reasoning, because it lacks any capacity for legible self-description.

UEFI (Unified Extensible Firmware Interface) (Chapter 12): the firmware that starts a computer before its operating system.

Unicity (Chapter 7): the property of a name that it stands for exactly one thing; once a name is chosen, it is the only property of the name that matters.

Uniquity (Chapter 4): the principle that every instance is unique, since no two contexts of instantiation are the same; a structural necessity of the framework.

USI number (Chapter 11): the name given to the identifier of Chapter 3, a uniform source code unique number, printed at each point where an error can be signaled, so that a log shows where in the code the error arose.

Variable prefixes (Chapter 7): the naming convention by kind of symbol, written in code in lower case with an underscore: function input (fi_), function local (fl_), function output (fo_), function (fn_), class (cl_), operator (op_), static structure (ss_), module local (ml_), module input and output (mi_, mo_), preprocessor definition (pr_), label (lb_), type (ty_) and transmit identifier (tx_); Table 7.1.

Version suffix (Chapter 11): the _1, _2 that ends every name, so that no name is part of another and every name is a unique, searchable key.

Von Neumann self-reproduction and Gödelian self-reference (Chapter 3): two traditions of formal self-reference, one constructive and used as the closer precedent, the other limitative and set aside.

WebAssembly (Chapter 11): a compact binary format for programs, run by web browsers at close to the speed of machine code; the successor to ASM.js.

Zenith and nadir (Chapter 5): the two poles of a Halo circle; the diamond at the zenith is the instance being called, and the square at the nadir the instance being defined.

Bibliography

Amdahl, Gene M. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". *AFIPS Conference Proceedings* 30 (Spring Joint Computer Conference, 1967): 483-485. (Chapter 6.)

American National Standards Institute. *Programming Language Common Lisp*. ANSI INCITS 226-1994 (first published as ANSI X3.226-1994). New York: American National Standards Institute, 1994. (Chapter 11.)

Backus, John. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". *Communications of the ACM* 21, no. 8 (1978): 613-641. (Chapter 7.)

Beneš, V. E. "Optimal Rearrangeable Multistage Connecting Networks". *The Bell System Technical Journal* 43, no. 4 (July 1964): 1641-1656. (Chapter 2.)

Bernstein, Philip A. "Synthesizing Third Normal Form Relations from Functional Dependencies". *ACM Transactions on Database Systems* 1, no. 4 (1976): 277-298. (Chapters 1 and 3.)

Berry, Gérard, and Georges Gonthier. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation". *Science of Computer Programming* 19, no. 2 (1992): 87-152. (Chapter 13.)

Böhm, Corrado, and Giuseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". *Communications of the ACM* 9, no. 5 (1966): 366-371. (Chapter 1.)

Carrington, R. C. "Description of a Singular Appearance seen in the Sun on September 1, 1859". *Monthly Notices of the Royal Astronomical Society* 20 (1859): 13-15. (Chapter 10.)

Church, Alonzo. "An Unsolvable Problem of Elementary Number Theory". *American Journal of Mathematics* 58, no. 2 (1936): 345-363. (Chapters 1 and 5.)

Clos, Charles. "A Study of Non-Blocking Switching Networks". *The Bell System Technical Journal* 32, no. 2 (March 1953): 406-424. (Chapter 2.)

Codd, E. F. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM* 13, no. 6 (1970): 377-387. (Chapter 8.)

Codd, E. F. "Further Normalization of the Data Base Relational Model". In *Data Base Systems*, edited by Randall Rustin, 33-64. Englewood Cliffs, New Jersey: Prentice-Hall, 1972. (Chapters 3 and 8.)

Cox, P. T., F. R. Giles, and T. Pietrzykowski. "Prograph: A Step Towards Liberating Programming from Textual Conditioning". In *Proceedings of the 1989 IEEE Workshop on Visual Languages*. IEEE Computer Society, 1989. (Chapter 13.)

de Bruijn, N. G. "Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem". *Indagationes Mathematicae* 34 (1972): 381-392. (Chapter 7.)

Dijkstra, Edsger W. "Go To Statement Considered Harmful". *Communications of the ACM* 11, no. 3 (1968): 147-148. (Chapter 6, structured programming.)

Feathers, Michael C. *Working Effectively with Legacy Code*. Upper Saddle River, New Jersey: Prentice Hall, 2004. (Chapter 8.)

Gödel, Kurt. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I". *Monatshefte für Mathematik und Physik* 38 (1931): 173-198. (Chapter 3.)

Goldwasser, Shafi, Michael P. Kim, Vinod Vaikuntanathan, and Or Zamir. "Planting Undetectable Backdoors in Machine Learning Models". In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2022. https://arxiv.org/abs/2204.06974. (Chapter 2.)

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Second edition. Boston: Addison-Wesley, 2000. (Chapter 10: Section 5.3, method invocation conversion.)

Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud. "The Synchronous Data Flow Programming Language LUSTRE". *Proceedings of the IEEE* 79, no. 9 (1991): 1305-1320. (Chapter 13.)

Harel, David. "Statecharts: A Visual Formalism for Complex Systems". *Science of Computer Programming* 8, no. 3 (1987): 231-274. (Chapter 13.)

Hickey, Rich. "A History of Clojure". *Proceedings of the ACM on Programming Languages* 4, HOPL (June 2020). (Chapter 11.)

Hills, Gage, and others. "Modern Microprocessor Built from Complementary Carbon Nanotube Transistors". *Nature* 572 (2019): 595-602. (Chapter 10.)

Holzmann, Gerard J. "The Power of 10: Rules for Developing Safety-Critical Code". *Computer* 39, no. 6 (2006): 95-99. (Chapter 11.)

IEEE. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*. IEEE Std 1364-1995. New York: Institute of Electrical and Electronics Engineers, 1996. (Chapter 13.)

IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE Std 1076-1987. New York: Institute of Electrical and Electronics Engineers, 1988. (Chapter 13.)

Kahn, Gilles. "The Semantics of a Simple Language for Parallel Programming". In *Information Processing 74: Proceedings of IFIP Congress 74*, 471-475. Amsterdam: North-Holland, 1974. (Chapter 13.)

Kodosky, Jeffrey. "LabVIEW". *Proceedings of the ACM on Programming Languages* 4, HOPL (June 2020). (Chapter 13.)

Landry, Forrest. *An Immanent Metaphysics*. First published 2001. Wytheville, Virginia: UVSM Publications. ISBN 979-8-9896628-4-5. Cited by aphorism number. (Introduction, Chapters 1, 3, 4 and 13.)

Landry, Forrest. *Keeping Software Soft; Explicit Code, Safe Change, and the Discipline of Subtraction*. 2026. Wytheville, Virginia: UVSM Publications. ISBN 979-8-90710-245-3. DOI: https://doi.org/10.67291/NDZY5454. (Chapters 9 and 12: code collaboration and semantic merging, in its Chapter 4.)

Landry, Forrest. *Software, Wildness, and the Normal Form; From Software Architecture to Life, Justice, and Wisdom*. 2026. Wytheville, Virginia: UVSM Publications. ISBN 979-8-90710-237-8. DOI: https://doi.org/10.67291/AWME8110. (Chapters 1 to 4: its Part I, "The Normal Form", which carries the same four chapters, in its own wording, as the ground of an argument that runs on to life, justice and wisdom. Chapters 2 and 3: its chapters "The Bandwidth Bound on Value" and "Open-Sourcing the World; Universal Transcoding and the Politics of a General Computation".)

Landry, Forrest. *The ITAG and EGS Standards; Structured Plain Text for Files and Notes*. 2026. Wytheville, Virginia: UVSM Publications. ISBN 979-8-90710-424-2. DOI: https://doi.org/10.67291/SPXV7124. (Chapter 9 and Chapter 11: the ITAG convention, in its Chapter 1, and EGS, in its Chapter 2, each set out in full.)

Landry, Forrest. *The RISL Standard; Halo Written as Code, Across Languages*. 2026. Wytheville, Virginia: UVSM Publications. ISBN 979-8-90710-423-5. DOI: https://doi.org/10.67291/NSOV9747. (Introduction; Chapter 6 and Appendix A: one bit for failure, in its Chapter 1, Section V; Chapter 9 and Chapter 11: the RISL conventions for each language, and the written standard for JavaScript, in its Chapter 5. The full ITAG and EGS standards are cited to *The ITAG and EGS Standards; Structured Plain Text for Files and Notes*.)

Landry, Forrest. *The Study of Pure Causality; On the Philosophy of Computer Science*. 2026. Wytheville, Virginia: UVSM Publications. ISBN 979-8-90710-232-3. DOI: https://doi.org/10.67291/ZVTE4072. (Chapter 13: its chapter "Drawing the Program".)

Landry, Forrest. "Which Programs Can Be Drawn; Halo and the Quantum Exception". Essay. 2026. Wytheville, Virginia: UVSM Publications. (Chapter 7.)

Maister, David H., Charles H. Green, and Robert M. Galford. *The Trusted Advisor*. New York: Free Press, 2000. (Chapter 2.)

MathWorks. "Simulink". Product documentation. https://www.mathworks.com/products/simulink.html. (Chapter 13.)

MISRA. *Guidelines for the Use of the C Language in Vehicle Based Software*. Nuneaton: Motor Industry Software Reliability Association, 1998. (Chapter 11.)

Moseley, Ben, and Peter Marks. "Out of the Tar Pit". Paper presented at the Software Practice Advancement conference (SPA 2006), London, 2006. (Chapter 8.)

National Institute of Standards and Technology. "NIST Removes Cryptography Algorithm from Random Number Generator Recommendations". News release, 21 April 2014. (Chapter 2: Dual_EC_DRBG.)

OpenJS Foundation. "Node-RED". https://nodered.org. (Chapter 13.)

Parnas, D. L. "On the Criteria to Be Used in Decomposing Systems into Modules". *Communications of the ACM* 15, no. 12 (1972): 1053-1058. (Chapter 8.)

Petri, Carl Adam. *Kommunikation mit Automaten*. Doctoral dissertation, University of Bonn, 1962. (Chapter 13.)

Resnick, Mitchel, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. "Scratch: Programming for All". *Communications of the ACM* 52, no. 11 (2009): 60-67. (Chapter 13.)

Schönfinkel, Moses. "Über die Bausteine der mathematischen Logik". *Mathematische Annalen* 92 (1924): 305-316. (Chapter 7.)

Shinn, Alex, John Cowan, and Arthur A. Gleckler, editors. *Revised^7 Report on the Algorithmic Language Scheme*. 2013. (Chapter 11.)

Shirriff, Ken. "Intel's $475 Million Error: The Silicon Behind the Pentium Division Bug". Blog post, December 2024. https://www.righto.com/2024/12/this-die-photo-of-pentium-shows.html. (Chapter 2: the Pentium floating-point division error of 1994.)

SpiderMonkey team. "Saying Goodbye to asm.js". SpiderMonkey blog, 20 May 2026. https://spidermonkey.dev/blog/2026/05/20/saying-goodbye-to-asmjs.html. (Chapter 11: ASM.js and its successor, WebAssembly.)

Spolsky, Joel. "Don't Let Architecture Astronauts Scare You". *Joel on Software*, 21 April 2001. https://www.joelonsoftware.com/2001/04/21/dont-let-architecture-astronauts-scare-you/. (Chapter 2.)

Sun Microsystems. "J2SE 5.0 New Features and Enhancements". Release notes, 2004. (Chapter 10.)

Thompson, Ken. "Reflections on Trusting Trust". *Communications of the ACM* 27, no. 8 (1984): 761-763. (Chapters 2 and 12.)

Turing, Alan M. "On Computable Numbers, with an Application to the Entscheidungsproblem". *Proceedings of the London Mathematical Society*, series 2, 42, no. 1 (1937): 230-265. https://doi.org/10.1112/plms/s2-42.1.230. (Chapters 1, 3 and 8.)

United States Code. Title 17, Section 101, Definitions ("derivative work"). (Chapter 7.)

von Neumann, John. *Theory of Self-Reproducing Automata*. Edited and completed by Arthur W. Burks. Urbana: University of Illinois Press, 1966. (Chapter 3.)

Index

Amdahl, Gene, Chapter 6

Amdahl's law, Chapter 6

ANSI C, Chapter 5; hot patching in, Chapter 9

appendices: rules of a Halo drawing, Appendix A; hello world drawn and written, Appendix B

architectural identity, Chapter 2

architecture astronauts, Chapter 2

articulation, point of, Chapter 8

ASM.js, Chapter 11

assembly language, Chapter 5, Chapter 9

axiomatic form, Chapter 13; the six parts, Chapter 13; each part required, Chapter 13; every layer, Chapter 13

axioms of the Immanent Metaphysics, Introduction

Backus, John, Chapter 7

Beneš, V. E., Chapter 2

Bernstein, Philip A., Chapter 1, Chapter 3

Böhm, Corrado, and Giuseppe Jacopini, Chapter 1

bounded loop, Chapter 8; drawn, Chapter 6, Chapter 8; in the RISL standard, Chapter 11

box labeled OS, a leaf node, Chapter 6

boxing, Chapter 10

Brainfuck, Chapter 3

callback, Chapter 6; drawn, Chapter 6

call geometry, Chapter 11

call triple, Chapter 9

canonical form, Chapter 11; of a module, Chapter 11

carbon, integrated circuits built on, Chapter 10

Carrington event, Chapter 10
cc and com, namespace objects, Chapter 11, Chapter 11
Church, Alonzo, Chapter 1
Church-Turing thesis, Chapter 1
circle, drawn small or large, Chapter 5
clarity, Chapter 2
class and instance, Chapter 4, Chapter 4; class as factory, Chapter 8
clean system for drawing a program, Chapter 13
clean system, the four properties and what each gives, Chapter 13
Clojure, Chapter 11
Clos, Charles, Chapter 2
closed source as concealed intent, Chapter 2
COBOL, Chapter 10
Codd, E. F., Chapter 8
code colonialism, Chapter 2
code that makes code, Chapter 7
composition and inheritance, Chapter 11
compression, Chapter 2
conjugate pair, Introduction
contact space, Chapter 2
content, context and choice, Introduction
Count to Ten, Chapter 6, Chapter 8, Appendix B
cryptography, the use a linear compiler replaces and the use it leaves, Chapter 12
currying, Chapter 8
data line, Chapter 5, Chapter 6
de Bruijn, N. G., Chapter 7
definition of Halo, Part II and Appendix A, Introduction
denormalization, Chapter 1

derivative work, Chapter 7
diagram as program, Chapter 13
diamond and square (instance call and instance definition), Chapter 5
Dijkstra, Edsger, Chapter 6
drawing the program, Chapter 7, Chapter 13
drawn languages, other, what each adds, Chapter 13
Dual_EC_DRBG, Chapter 2
EGS (Extended Grammatical Structuring), Chapter 9, Chapter 11
ENDF, Chapter 9
equivalence thesis, Chapter 1, Chapter 8
essential and accidental properties, Chapter 4
Esterel, Chapter 13
Ethereum, and its gas, Chapter 12
exact in both directions, Chapter 7
exfiltration, infiltration and modification, Chapter 12, Chapter 12
factoring of intentionality, Chapter 2
factory, Chapter 8; table of instances, Chapter 8; factory factory, Chapter 8
fail exit, one bit, Chapter 6
fan-in and fan-out, Chapter 6
Feathers, Michael, Chapter 8
fidelity to a class, Chapter 4
file, how a drawing is stored as (a separate standard), Chapter 9, Chapter 9
flowchart, Chapter 8; the version of a Halo drawing closer to a flowchart, Chapter 11
foreign key, Introduction
form of the page, Chapter 11
FPGA and GPU, Chapter 5
fungibility, Chapter 7

globally unique identifier, on every function, Chapter 3
global value, Chapter 6
Gödel, Kurt, Chapter 3
Goldwasser, Shafi, and others, Chapter 2
Gosling, James, and others, Chapter 10
grouping, Chapter 5
Halo, Chapter 3; the element, Chapter 5; hello world, Chapter 6; what counts as code, Chapter 7; superset of the flowchart, Chapter 8; modules as tables, Chapter 9; libraries, Chapter 10; in JavaScript, Chapter 11; trustable computing, Chapter 12; the axiomatic form, Chapter 13
halting problem, Chapter 8
hardware description languages (VHDL, Verilog), Chapter 13
Harel, David, Chapter 13
hello world, Chapter 6, Chapter 8, Chapter 12, Appendix B
Holzmann, Gerard, Chapter 11
homomorphic encryption, Chapter 7
hot patching, Chapter 9
idempotent all the way down, Chapter 9
idiom, Chapter 9
immanent, omniscient, transcendent, Introduction
imperative model, Chapter 5
Incommensuration Theorem, Introduction
inheritance, as tables, Chapter 1
Intel Pentium, division error, Chapter 2
intentionality, Introduction
intermediate language, Chapter 10
intermediate variables, Chapter 7
ITAG convention, Chapter 9
JavaScript, Chapter 9, Chapter 11

Java, version 1.4, Chapter 10; version 1.5, Chapter 10
join, Chapter 6; drawn, Chapter 6
Kahn, Gilles, Chapter 13
K matrix, Chapter 2
LabVIEW, Chapter 13
leading edge, Chapter 5
leaf node, Chapter 6
learning Halo, Chapter 5
legibility, why generality requires it, Chapter 3
legible self-description, Chapter 3
libraries, Chapter 10
lifecycle functions, Chapter 11
linear compiler design, Chapter 12; one call compiled, Chapter 12
lines, crossing, and inputs left unwired, Chapter 6
Lisp, Chapter 11
load and recover, Chapter 11
logarithm, where it comes from, Chapter 2
longevity, Chapter 10; the thousand-year horizon, Chapter 10
lozenge, Chapter 7; its three uses told apart, Chapter 7
Lustre, Chapter 13
Maister, David, Charles Green, and Robert Galford, Chapter 2
man, machine and nature, Chapter 2
Mars, computing on, Chapter 10
membrane, Chapter 6
merging drawings, Chapter 9
metalanguage processor, Halo, Chapter 9, Chapter 7
microscopic, mesoscopic, macroscopic, Chapter 1
MISRA guidelines, Chapter 11
misread joints, and technical debt, Chapter 8

modalities, Introduction
module as a table, Chapter 9; module pattern, Chapter 11; in the RISL standard, Chapter 11; one module in several languages, Chapter 9
Moseley, Ben, and Peter Marks, Chapter 8
multipolar trap, Chapter 2
multivariate convention, Chapter 2; multivariate output, Chapter 5
names, as a layer over identity, Chapter 7
NAND gate, Chapter 5
Node.js, Chapter 9, Chapter 11, Chapter 11
Node-RED, Chapter 13
non-imperative languages (SQL, Prolog, Haskell), Chapter 5
normal forms, teaching ladder of, Chapter 8
normalization, Chapter 1, Chapter 2
NoSQL database, Chapter 8
not code, Chapter 7
occult intentionality, Chapter 2
ordinal position, Chapter 7
Parnas, David, Chapter 8
Petri nets, Chapter 13
poles of the circle (zenith and nadir), Chapter 5
primitives, Chapter 5, Chapter 5
principal-agent problem, Chapter 2
process, Chapter 1
Prograph, Chapter 13
pull and push, Chapter 5, Chapter 6
recursion, and instances, Chapter 8
reduced subset of JavaScript, Chapter 11
refactoring tools, Chapter 10, Chapter 11
reflection, Chapter 10

regress of trust, Chapter 2
REPL (read-eval-print loop), Chapter 9
RISL conventions, Chapter 9, Chapter 11; any language to any other, Chapter 9; projects written to, Chapter 10; the written standard for JavaScript, Chapter 11
Rosetta Stone language, Chapter 7
rules of a Halo drawing, Appendix A
Schönfinkel, Moses, Chapter 7
Scratch, Chapter 13
scope of variables, Chapter 8
seam (Feathers), Chapter 8
sequentiality and the limits of parallelism, Chapter 6
sequentiality, conditionality and rings, Chapter 1
side channel, Chapter 12
simplest module, Chapter 7
Simulink, Chapter 13
single array f, Chapter 11
Spolsky, Joel, Chapter 2
SQL (Structured Query Language), Chapter 8
stack trace, Chapter 11
state machine, Chapter 1; named states, Chapter 7
structured programming, Chapter 6
structured program theorem, Chapter 1
success and fail, Chapter 6
switching networks, multistage, Chapter 2
symmetry and continuity, Introduction
syntactic sugar, Chapter 5
syntax and semantics, separation of, Chapter 7
system catalog, Chapter 3

table at its ceiling, Chapter 3
tables of modules, Chapter 8
teaching picture, Chapter 13
technical debt, Chapter 8
third normal form, Chapter 1; as compression, Chapter 2; from first principles, Chapter 3; at the system level, Chapter 8
Thompson, Ken, Chapter 2
three hard problems, Chapter 12, Chapter 12
transpile, Chapter 3
trap identifier, Chapter 11
trigger line, Chapter 6
trigger triangles, Chapter 5
Trojan horse, Chapter 12
trustable computing, Chapter 12
trusted compute hardware (UEFI Secure Boot), Chapter 12
trust equation, Chapter 2
trusting trust, Chapter 2
Turing, Alan, Chapter 1
Turing complete, Chapter 1
Turing completeness, Halo's complexity held near it, Chapter 10
Turing machine, Chapter 3, Chapter 3
unicity, Chapter 7
uniquity, Chapter 4
USI number, Chapter 11
variable prefixes, Chapter 7
version suffix, Chapter 11
von Neumann, John, Chapter 3
WebAssembly, Chapter 11