

Chapter 2

Digital Computer Systems

This chapter considers issues of emergence and causation in the case of digital computers, as a warm-up example before giving a general viewpoint on these topics in the next chapter. It will be shown that top-down causation is central to their functioning. It develops its themes as follows:

- Section 2.1 discusses the computational basics underlying the functioning of digital computers.
- Section 2.2 discusses how modular hierarchical structures enable complex higher level behaviour to emerge.
- Section 2.3 sets out the implementation and logical hierarchical structures and makes the case that software drives what happens.
- Section 2.4 discusses how both bottom-up and top-down causation happen in these hierarchies, distinguishing five types of top-down causation that have rather different dynamics.
- Section 2.5 discusses the key feature of equivalence classes that underlies the ontological nature of higher level causal elements. It characterizes in precisely what way computer programs are abstract entities.
- Section 2.6 considers the issue of clearing memory and deleting records: a selection process that leads to the irreversibility of computation. This relates to the fact that infinities cannot occur in physical reality.
- Section 2.7 looks at the nature of causation in the light of all the above, making the case for causal effectiveness of non-physical entities in digital computer systems.

2.1 Computational Basics

Digital computers are the embodiment of algorithmic operation, and are nowadays regarded as a fundamental model of causation. Many claim physics can be regarded as a computational process, and indeed that the universe is a computer [44], computational models are proposed for social life [49], and the computer is often used

as a model for how the mind works (some say it *is* a computer, others regard that as an analogy [59]). Accordingly, it is useful to consider how issues of emergence and causation work out in this case, so it serves as a model for effects we may see in other contexts, and in particular in the brain, as the computational metaphor does indeed seem to capture some aspects of what is going on in the brain (even though it is inadequate as a total explanation of the embodied mind).

Turing explains the basic idea as follows [64]:

The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer. The human computer is supposed to be following fixed rules; he has no authority to deviate from them in any detail. We may suppose that these rules are supplied in a book, which is altered whenever he is put on to a new job. He has also an unlimited supply of paper on which he does his calculations. He may also do his multiplications and additions on a ‘desk machine’, but this is not important.

This gives the essential operational idea of algorithmic operation of a computer, also explained nicely by Hofstadter [34, pp. 33–41]. MacCormick states it thus [45, p. 3]:

An algorithm is a precise recipe that specifies the exact sequence of steps required to solve a problem.

Turing explains how arbitrary computations can be realized if a digital computer can be regarded as consisting of three parts [64]:

1. **A Store of Information (Memory).** This stores data that includes an instruction table stating the rules to be obeyed by the computer (nowadays called a *program*).
2. **An Executive Unit (CPU).** This carries out the various individual operations involved in a calculation.
3. **Control (Operating System).** This sees that the instructions are obeyed correctly and in the right order.

The key feature leading to flexibility of use of the computer [21, p. 15] is the *stored program*, a set of symbolically encoded instructions in the machine’s memory. By altering the program (*software*), the same physical apparatus (*hardware*) can be used to tackle many different kinds of problems. Turing demonstrated [64] that, by this means, a single machine of fixed structure is able to carry out every computation that can be carried out by any computer whatsoever. This is the special property of digital computers, namely [64]:

They can mimic any discrete-state machine, [and this] is described by saying that they are universal machines. The existence of machines with this property has the important consequence that, considerations of speed apart, it is unnecessary to design various new machines to do various computing processes. They can all be done with one digital computer, suitably programmed for each case. It will be seen that as a consequence of this all digital computers are in a sense equivalent.

This characterizes the key property of programmable computers [14]:

Universal Logical Capability. The nature of the logical operations that digital computers are able to carry out is not constrained by the specific physical implementation chosen; the underlying physics enables the chosen logic rather than controlling it.

But of course, as Turing himself showed, they are nevertheless limited in what they can do [14, 21], and furthermore, these statements do not by themselves show how to organize a computer to achieve complex behavior.

The first major question is how a combination of such simple operations can enable arbitrary complexity of behavior to emerge. We shall see below that a core requirement is:

Modular Hierarchical Structuring of Both Hardware and Software. This enables structured top-down causation in the hierarchy, in particular allowing software patterns to control hardware. In this way, abstract entities have causal effects in the physical universe.

This is the first key to the emergence of true complexity, and is embodied in all current digital computers in both implementation and logical hierarchies (Sect. 2.3). In his textbook on computing, Robert Keller writes [39]:

An abstraction is an intellectual device to simplify by eliminating factors that are irrelevant to the key idea [...] The idea of levels of abstraction is central to managing complexity of computer systems, both software and hardware. Such systems typically consist of thousands to millions of very small components (words of memory, program statements, logic gates, etc.). To design all components as a single monolith is virtually impossible intellectually. Therefore, it is common instead to view a system as being comprised of a few interacting components, each of which can be understood in terms of its components, and so forth, until the most basic level is reached.

This is particularly clear in the class/object hierarchy of object oriented languages [13].

But a further step is needed: how do we get a computer to carry out computations that are not simply logical implications of what is in the initial data? This is a core requirement on the road towards intelligence: how can we get them to learn? Turing makes a very interesting observation in this regard [64]:

An interesting variant on the idea of a digital computer is a ‘digital computer with a random element’. These have instructions involving the throwing of a die or some equivalent electronic process; one such instruction might for instance be, ‘Throw the die and put the resulting number into store 1000.’

While this breaks the system out of a rigidly determined cycle of deterministic operations, by itself this won’t do the job of creating intelligent behaviour. But it does open the way to programming computers to behave in an adaptive way. The second key feature needed is:

Adaptive Selection. Procedures embodying adaptive selection in the manipulation of data enable the building up of meaningful information from unstructured incoming data streams or randomized internal variables. This is the basis of learning.

This is a kind of top-down causation that is crucial in enabling computers to carry out processes equivalent to learning (Sect. 2.4.4), e.g., through artificial neural networks [11] and genetic algorithms [23], and so allows local processes to flow against the stream of decay embodied in the Second Law of Thermodynamics.

Associated with this is a further crucial idea, omitted in Turing's list of operations above (because he assumed infinite memory was available):

4. **Emptying Memory.** Clearing out or overwriting short term and long term memory locations so that they can be used again.

For one thing, this enables the finite memory of the computer to act as an effectively infinite memory store, thus taming the impractical memory requirements of an infinite tape. For another, it crucially involves the element of selecting what will be kept and what discarded. As just mentioned, such selection processes are the key to building up meaningful information out of a jumble of incoming data: forgetting is the crucial counterpart of remembering! It is also where irreversibility associated with entropy production happens in computations [43].

2.2 Modular Hierarchical Structures

Digital computers involve two orthogonal but interacting hierarchies (Sect. 2.3). This is not by chance. A major theme of this book is that, as pointed out by Simon [61]:

Genuine complexity can only emerge from networks of causation involving modular hierarchical structures.

Note that this principle applies to both physical and logical complex systems. Both kinds occur in digital computers (see the next section).

Each word is important: the physical and logical *hierarchies* (Sect. 2.3) are *structured* by means of carefully configured physical and logical connections [47, 62], and each involve interacting *modules* at many levels [30]. The system is composed of inter-related subsystems that have in turn their own subsystems, and so on, until some lowest level of component is reached where the basic work is done. This structure enables a build-up of genuine complexity if appropriately formed to fulfill some higher level function: as in biology, structure follows function. Examples are sub-routines, procedures, objects. Each has a name, which identifies the specific entity, and a type, which identifies the class of entities it belongs to.

I consider in turn:

- Structures: Combination and abstraction (Sect. 2.2.1).
- Decomposition and modularity (Sect. 2.2.2).
- Encapsulation and information-hiding (Sect. 2.2.3).
- Naming, combination, and recursion (Sect. 2.2.4).
- Hierarchy: Class structure and object structure (Sect. 2.2.5).
- Evolution (Sect. 2.2.6).

2.2.1 Structures: Combination and Abstraction

In digital computer languages, as explained by Abelson and Sussman [1, p. 4], structures are formed in the following way:

Every powerful language has three mechanisms for combining simple ideas to form more complex ideas:

- **Primitive Expressions.** These represent the simplest entities the language is concerned with.
- **Means of Combination.** These serve to build up compound elements from simpler ones.
- **Means of Abstraction.** These serve to name compound elements and are manipulated as units.

A key element here is naming compound entities, indexing them, and having rules about how they can interact:

- **Names.** Any named entity is identified as a potentially causally effective agent, whether it is physical or abstract. Indeed, any entity that is causally effective in a programme has to be given a name so that it can be referred to (Sect. 2.2.4). The name must have attributes identifying whether they refer to objects or processes or something else.
- **Indexes/Pointers.** It then also has to have an index or pointer that shows where the relevant records are stored in memory.
- **Logical Rules.** Abstract rules can then be applied to sets of named entities, these rules embodying the logic of their interactions, and which processes can interact with each object.
- **Action Rules.** Action rules can be signified by the named entity (e.g., print text.pdf).

In the end these are the abstract technologies that enable computation to function. They are causally effective because they result in the computer being able to operate. The foundational key is the ability to give names to recognisable entities—generic (hence necessarily abstract) and specific (whether physical or abstract).

Emergence. Such combinations of parts lead to the higher level functionality of a complex logical system. The behaviour of the whole is greater than the sum of its parts, and cannot even be described in terms of the language that applies to the parts. This is the phenomenon of *emergent order*: the higher levels exhibit kinds of behaviour that are more complex than those the lower level parts are capable of.

In the implementation hierarchy, much the same applies. Emergence of layers of structure and behaviour, one upon the other, lead to hierarchical structuring and this enables a build-up of higher level entities that can be characterised by abstract properties. Not only is the structure hierarchic, but the levels of this hierarchy represent different levels of abstraction, each built upon the other, and each understandable by itself (and each characterised by a different phenomenology).

Phenomenology. All parts at the same level of abstraction interact in a well-defined way, whence they have a causal reality at their own level, and each is represented in a different language describing and characterising the causal patterns at work at that level [62], which may entail their own logical hierarchies. The vocabulary to describe each of the levels in each hierarchy is different at each level, because the nature of the relevant entities at each level is quite different from that at the levels above and below.

2.2.2 *Decomposition and Modularity*

A hierarchy represents a decomposition of the problem into constituent parts and processes to handle those constituent parts, each requiring less data and processing, and more restricted operations than the problem as a whole [12]. The success of hierarchical structuring depends on (i) implementation of modules which handle these lower-level processes, such as the CPU and memory circuits and interconnections between them, (ii) integration of these modules into a higher-level structure, viz., the computer as a whole. The idea is to encapsulate functions in modular units with information-hiding and abstraction, so that named entities can be regarded as functional wholes whose internal functioning is hidden from the outside view. I closely follow Booch's excellent exposition of object-oriented analysis [12], together with Beer's exposition of the principles of decentralized control [7].

Modularity [12, pp. 12–13, 54–59]. The technique of mastering complexity in computer systems and in life is to decompose the problem into smaller and smaller parts, each of which we may then refine independently [12, p. 16]. The basic principle is

Divide and Conquer. Divide a complex overall task into many simpler subtasks, each requiring lesser data and computational power than the whole; then integrate the results so as to attain higher level cohesive behaviour, thus creating complex outcomes.

By organising the problem into smaller parts, we break the informational bottleneck on the amount of information that has to be received, processed, and remembered at each step; and this also allows specialisation of operation. This implies the creation of a set of specialised modules to handle the smaller problems that together comprise the whole: in computer systems these will be subroutines, which Turing called 'subsidiary tables'.

According to Abelson and Sussman, one breaks up a complex problem into subproblems, each accomplished by a separate procedure. The program used can be viewed as a cluster of procedures that mirror the decomposition of the problem into subproblems [1, p. 26]:

The importance of this decomposition strategy is not simply that one is dividing the program into parts. After all, one could take any large program and divide it into parts—the first ten lines, the next ten lines, and so on. Rather it is crucial that each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.

They emphasize that when developing a program like this, we are not initially concerned with how the procedure computes its result, only with the fact that it does so. The details of how this is done can be held back until later on. Thus one actually deals with a procedural abstraction. At this level, any procedure that computes the desired output will do. This is the principle of equivalence classes, showing that top-down causation is taking place (see Sect. 2.5). Intra-component linkages are generally stronger than inter-component linkages. This fact has the effect of separating the high frequency dynamics of the components, involving their internal structure, from the low-frequency dynamics, involving interactions amongst components [61] (and it is for this reason that we can sensibly identify the components).

A further basic principle is that this allows one to:

Adapt and Re-Use [61]. In building complex systems from simple ones, or improving an already complex system, one can re-use the same modular components in new combinations, or substitute new, more efficient components, with the same functionality, for old ones.

Thus we can benefit from a library of tried and trusted components. Complex structures are made of modular units with abstraction, encapsulation, and inheritance, and this enables the modification of modules and re-use for other purposes (Sect. 2.2.6).

2.2.3 *Encapsulation and Information-Hiding*

Named objects carry with them expectations of behaviour that identify abstractions: specific essential characteristics of the object (ignoring other properties as inessential).

Abstraction and Labeling [12, pp. 20, 41–48]. Unable to master the entirety of a complex object, we choose to ignore its inessential details, dealing instead with a generalised idealised model of the object. An abstraction denotes the essential characteristics of an object that distinguishes it from all other kinds of objects. An abstraction focuses on the outside view of the object, and so serves to separate its essential behaviour from its implementation. It emphasises some of the system's details or properties, while suppressing others. A key feature is that compound objects can be named and treated as units (Sect. 2.2.4). This leads to the power of abstract symbolism and symbolic computation.

Encapsulation and Information-Hiding [12, pp. 49–54] and [57, pp. 233–234, 476–483]. In a hierarchy, through encapsulation, objects at one level of abstraction are shielded from implementation details of lower levels of abstraction. Consumers of services only specify what is to be done, leaving it to the object to decide how to do it: “No part of any complex system should depend on the internal details of any other part.” Encapsulation occurs when the internal workings are hidden from the outside, so its procedures can be treated as black-box abstractions. To embody this, each class of object must have two parts: an *interface* (its outside view, encompassing an abstraction of the common behaviour of all instances of the class of objects)

and an *implementation* (the internal representations and mechanisms that achieve the desired behaviour). This is formalised in declarations of public and private variables. Efficiency and usability introduce the aim of reducing the number of variables and names that are visible at the interface. This involves information-hiding, corresponding to coarse-graining in physics. The accompanying loss of detailed information is the essential source of entropy in the case of physics.

2.2.4 Naming, Combination, and Recursion

When names can be allocated to collections of names, this allows recursion, which is how real complexity gets built up (languages explicitly allow it). Indeed this is the power of *symbolic representation*: the name is a symbol for the thing it represents; and one can give names to patterns of names.

Naming. The key feature in setting up modules is first to identify them as entities by naming them: both classes, with generic features, and particular objects, with specific features, and then to refer to them by that name (an *identifier*) [57, pp. 45–46, 78–80]. Associated with the name is a set of attributes that characterise the object:

- A state embodied in internal variables of specific types (the *arguments*).
- A set of characteristic behaviours that characterise how it can interact with other objects (the *methods*). These are the law-like rules of behaviour that outline the nature of the object and create ordered outcomes.
- An indexed storage location, allowing programs to access this information (involving *pointers*).

The names are referenced in an index, enabling one to locate the physical location of the items referenced by the name, and pointers enable storage in non-contiguous memory locations. Each segment of a stored item must have a clear start address and end address, as well as links to any further parts of the same stored item or memory. So objects have a state, behaviour, and identity [12, pp. 81–97].

Typing and Links. Each object has a type, that is, a precise characterisation of its structural or behavioural properties shared by a collection of entities [12, pp. 65–72]. This includes the scope of its name, i.e., whether it has global validity, or is only valid in some local context. Its possible interactions with other objects are characterised by links between objects [12, pp. 98–102]. Object diagrams show the existence of objects and their relationships in the logical design of a system [12, pp. 208–219].

Coding and Information. From a functional viewpoint, one is involved in *coding* a message from a sender to a receiver. Use of a code involves *two* pattern recognition mechanisms: one for translating an incoming message into the code, followed by some kind of transformation of the coded message, and then one further pattern recognition system for decoding the output message into a usable form that will have the desired effect. From the viewpoint of the information involved, typing also

includes the rules an object has to obey, that is, the *syntax* of its allowed usage. The further aspects of a symbolic system are *semantics* (the meaning they embody) and *pragmatics* (the effect they have within the context of their usage). These aspects relate to the logical and physical effects of symbolic usage.

Collective Names. One can give a name to any pattern of symbols, including a collection of names. This is what enables one to create classification hierarchies and complex sentence structures, because one can refer to complex entities through a single name. This is also a way of reducing complexity: one does not have to deal with the details, but just with aggregate behaviour. The minimal program to solve some problem is reduced from thousands of lines of code to ‘run prog.exe’. The algorithmic complexity is thereby dramatically reduced.

Combination. Given names, they can be combined in grammatical structures indicating relationships between named entities via named operations. Collections of names can be treated as single entities (phrases function as effective words). This is the key property that enables construction of hierarchical structures, i.e., structures made up of parts that are themselves made up of parts, and so on. This is a core feature of natural language [63].

Recursion. This kind of structure enables one to repeatedly call up the same named entity, nesting structures inside each other. In functional terms, the essential requirement is an operation for combining data objects such that the results of the operation can themselves be combined using the same operation. This *closure property*, for example, underlies the importance of the list structure as a representational tool in LISP [1, p. 98]. When the data object is itself an operation, this enables recursion, that is, an operation or evaluative rule that includes as one of its steps the need to invoke the rule itself [1, pp. 9, 31–42].

2.2.5 Hierarchy: Class Structure and Object Structure

The power of a class hierarchy comes from the fact that it shows the relationships between similar kinds of objects, i.e., which are generalizations of others, and which are specializations. It allows one to relate them by inheritance, a feature which often characterises the nature of the hierarchical structure (see [12, pp. 59–65] and [57, pp. 453–476, 484–494]). Thus we don’t have to memorize separately all the properties of each kind of object or action: we can relate them to similar kinds of objects, remembering the class structure as a whole, and then the similarities and differences of specific members of the class (animal, mammal, dog, Dachshund, Fred). One then uses this to relate the properties of specific instances to the generic properties of a class.

To accommodate this in an object-oriented approach, objects occur in hierarchical functional classes, with inheritance of properties modified by specialization and variation. This class structure is related to the object structure because each object in the object structure is a specific instance of some class [12, pp. 14–15]. Together

these form the logical model. In a list-based language, one has a hierarchy of types [1, pp. 197–199]. In both cases, crosslinks may be allowed (reflecting the fact that various hierarchies are in operation, as emphasized above).

A **class** is a set of objects that share a common structure and a common behaviour [12, pp. 103–106]. The structures chosen to define a class depend on the classification scheme used: they embody a view of the world [12, pp. 145–168]. A single object is an instance of a class. Classes are objects that can themselves be manipulated as an entity. A *metaclass* is a class whose instances are themselves classes [12, pp. 133–134], so we can have a hierarchy of classes, and class families [12, pp. 337–340]. *Class diagrams* show the existence of classes and their relationships in a logical view of a system [12, pp. 176–196], and these relationships are formalised in class specifications [12, pp. 196–199], stating their name, responsibilities, attributes, operations, and constraints. *State transition diagrams* show the state space of a class, the events that cause a state change, and the actions that result from such a change [12, pp. 199–208]. *Module diagrams* show the allocations of classes and objects to modules in the physical design of the system [12, pp. 219–223]. *Process diagrams* show the allocation of processes to processors in the physical design of the system [12, pp. 223–226].

The dual hierarchical relations are *aggregation*, denoting which whole is made of which parts [12, pp. 128–130], and *membership*, denoting which parts belong to which whole. Aggregation may or may not imply physical containment: it may just imply a conceptual whole/part relationship [12, pp. 102–103].

Inheritance [12, pp. 59–62, 107–128] and [57, pp. 453–476]. This is the most important feature of a classification hierarchy. It allows an object class, such as a set of modules, to inherit all the properties of its superclass, and to add further properties to them (it is a ‘is a’ hierarchy). This allows similarities to be described in one central place and then applied to all the objects in the class and in subclasses. It makes explicit the nature of the hierarchy of objects and classes in a system, and implements generalisation/specialisation of features (the superclass represents generalised abstractions, and subclasses represent specializations in which variables and behaviours are added, modified, or even hidden). Inheritance with exceptions enables us to understand something as a modification of something already familiar, saves unnecessary repetition of descriptions or properties, and allows nonmonotonic reasoning [48].

Patterns. Particular types of *structural patterns* recur and are worth identifying and codifying in structural classes. They include lists [68, pp. 56–75], stacks [68, pp. 75–88], queues [68, pp. 88–98] and priority queues (heaps) [68, pp. 183–224], trees [68, pp. 99–153], graphs [68, pp. 291–352], and relational databases. Similarly, particular kinds of operations often occur and are worth identifying and naming. These include *date/time operations* and *filters*, i.e., input, process, and output transformations [12, pp. 331–332], *pattern matching*, i.e., operations for searching for structured sequences within sequences [12, pp. 370–372], *searching*, i.e., operations for searching for items within structures, *sorting*, i.e., operations for ordering structures, *utilities*, i.e., common composite operations building on more primitive operations, e.g., iteration [12, pp. 355–360] and statistical analysis.

The point here is that each of these structures and operations are metaclasses that can be identified and given their own name. They then exist, in virtue of this recognition, as entities in their own right that can from now on be accorded an ontological status as effective entities. They are abstract patterns that are causally effective. They are multiply realisable at a detailed level, and hence show some form of top-down causation or influence.

At a higher level, structural patterns that often occur in modular hierarchical structures can be encoded in *design patterns* that name, explain, and evaluate important recurring designs in object-oriented systems [31, pp. 2–3]. They are:

- **Creational Patterns.** Abstract Factory, Builder, Factory Method, Prototype, and Singleton.
- **Structural Patterns.** Adapter, Bridge, Composite, Decorator, Facade, Flyweight, and Proxy.
- **Behavioural Patterns.** Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor.

These are discussed in depth in [31]. They are based on foundation classes List, Iterator, ListIterator, Point, and Rectangle, with operations for construction, destruction, initialisation, and assignment of lists, and for accessing, adding, and removing elements of a list.

2.2.6 Evolution

Modularity underlies the possibility of successful development of truly complex systems [61]. One can adapt working modules for different purposes, without having to start from scratch. Selection of the most successful small variations of such classes enables incremental increase of complexity without the whole system crashing. Booch quotes Gall as follows [12, p. 13]:

A complex system that works is inevitably found to have evolved from a simple system that worked [...] A complex system designed from scratch never works, and cannot be patched up to make it work. You have to start over, beginning with a simple system.

This is an example of adaptive selection, a crucial form of top-down causation, which is the topic of the next section.

2.3 Orthogonal Modular Hierarchical Structures

Digital computers are hierarchically structured modular systems on both the hardware and software sides. Actually, there are two orthogonal kinds of hierarchies. I discuss:

- The two kinds of hierarchies (Sect. 2.3.1).
- The implementation (vertical) hierarchies (Sect. 2.3.2).

- The logical (horizontal) hierarchies (Sect. 2.3.3).
- The relation between the two hierarchies (Sect. 2.3.4).
- Causality in the hierarchies (Sect. 2.3.5).

2.3.1 *The Two Kinds of Hierarchies*

Firstly there are *implementation hierarchies* [62], which one might call *vertical hierarchies*, whereby the logical operations of the computer are implemented. For example, digital computers are constructed of integrated circuits containing a Central Processing Unit (CPU) which in turn contains an Arithmetic Logic Unit (ALU) made of many interconnected transistors, diodes, resistors, and capacitors, each comprised of an atomic lattice infused with electrons. The higher level physical structures are emergent entities, made up of the interconnected lower level components, but each describable and functioning effectively at its own emergent level. Related to these physical components is a software hierarchy: a tower of virtual machines that implement higher level programming languages at each virtual machine level, on the basis of the underlying machine code. An example is the Java Virtual Machine (JVM). Each of these virtual machines is emergent from the one on the next lower level.

Then there are the *logical hierarchies* [12], which one might call *horizontal hierarchies*, and which exist at each higher level of the virtual machine hierarchy. High level programs contain subroutines comprised of procedures set out in program lines which relate the relevant individual operations and variables. They thus represent a hierarchical structure of operations. These programs thereby also implement hierarchical data structures, e.g., a word-processor may edit a book consisting of chapters, paragraphs, sentences, phrases, words, and letters represented as such in a word-processing program. Both the logical and the data structures cascade down the implementation hierarchy through interpreters and compilers, which translate them into combinations of lower level operations and data elements [3, 6].

At the lowest abstract implementation level, both the data and programs will be represented as strings of 0s and 1s, realised as structured electronic states in the underlying physical level. A key feature is that many different implementation hierarchies can be used to realise the same logical hierarchy. The computational process itself is indifferent as to how it is realised at the physical level. This is a core aspect of top-down causation (Sect. 2.5).

2.3.2 *The Implementation (Vertical) Hierarchies*

As regards the implementation hierarchy [62], it has hardware and software aspects. First there is the *hardware hierarchy* shown in Table 2.1. It is modular because a network of many similar identifiable lower level elements such as logic circuits and transistors underlies each of the higher level structures.

Table 2.1 The hardware hierarchy for digital computers. The computer scientist takes level 1 as the base level. However, it is based on the underlying physics hierarchy, which enables its functioning. Layers below level 0 can be taken for granted by a computer engineer: it is the base level he needs to consider

Level 7	Global network
Level 6	Local network
Level 5	Computer
Level 4	Motherboard, memory banks
Level 3	CPU, memory circuits
Level 2	ALU, primary memory, bus
Level 1	Logic circuits, registers
Level 0	Transistors, resistors, capacitors
Level −1	<i>Atomic physics</i>
Level −2	<i>Nuclear physics</i>
Level −3	<i>Particle physics</i>
Level −4	<i>Fundamental theory</i>

The lowest level, i.e., the level where the real physical work is done, is the physical base level (level −4), which is some form of fundamental physics, possibly related to quantum gravity. But we do not know what the relevant physics is, so we cannot reduce the higher level actions to lowest level actions inter alia, because the lowest level is unknown. Of necessity, for practical purposes, we have to take one of the emergent effective levels of physical operation as the base level, assuming it to exist and be real [25]. For computer scientists, this is level 1 (the gate level); for computer engineers, it is level 0 (the transistor and solid-state physics level) which can be regarded as the level where the physical work is done (see Sect. 2.7.6).¹

However, hardware by itself will do nothing: it needs software in order to run. The *software hierarchy* is shown in Table 2.2. There is a tight logical structure at each level, governed by a set of syntactic rules for the language used at that level, and with an associated set of variables defined for that language, with typing and scoping rules. Each higher level language is emergent from the next lower level language through the way the higher level variables and operations are defined in terms of the lower level variables and operations. The magic that makes this happen is compilers and interpreters [3, 6], the foundation of truly complex functioning in computers.

The relation between level 0 and level 1 is where an appropriate physical representation of variables (in digital computers, electronic states) gives rise to a set of simple logical operations on those variables [47].

¹For a hardcore reductionist, it is illegitimate to regard these levels as real: they are epiphenomena arising from the underlying physics. This viewpoint provides no useful understanding of the causation in action.

Table 2.2 The software hierarchy for digital computers, based on the physics of the transistors at the device level. From Tanenbaum [62]

Level 7	Applications programs	Data and operations
Level 6	Problem-oriented language level	Classes, objects
Level 5	Assembly language level	Symbolic names
Level 4	Operating system machine level	Virtual memory, paging
Level 3	Instruction set architecture level	Machine language
Level 2	Microarchitecture level	Microprograms
Level 1	Digital logic level	Gates, registers
Level 0	<i>Device level</i>	<i>Transistors, connectors</i>

Principle C1. Information is not causally effective unless it has a physical representation, and some handles whereby this representation can (i) be inserted, altered, or deleted, and (ii) be read. The relation between levels 0 and 1 is where this happens.

Virtual Machines. A key point then is that Table 2.2 represents virtual machines at every level, except the lowest (level 0). Each of them runs on top of the next lower level virtual machine [62] (see Table 2.3). The lowest level is shown as level 0, which is of a completely different character to the others: it is physically based. The relation between level 0 and level 1 is where the transition between physical and abstract causation takes place: virtual machines (level 1 up) are based on real physical entities at the bottom (level 0).

Principle C2. All the higher levels in the software hierarchy are virtual machines. They are not physical systems.

Table 2.3 A multilevel machine. From Tanenbaum [62, p. 4]

Level n	Virtual machine M_n	Machine language L_n
	\vdots	\vdots
Level 3	Virtual machine M_3	Machine language L_3
Level 2	Virtual machine M_2	Machine language L_2
Level 1	Virtual machine M_1	Machine language L_1
Level 0	Actual computer M_0	Machine language L_0

Table 2.4 The logical hierarchy that determines which operations happen when

Level 7	Design patterns	Data structures	Methods of argumentation
Level 6	Programs	Classes, methods	Overall purpose
Level 5	Subroutines	Objects	Steps to overall purpose
Level 4	Algorithms	Data records	Implementation methods
Level 3	Program lines	Data items	Implementation units
Level 2	Operations	Variables	Entities interacting
Level 1	Representation atoms	Entity components	Logical base level

2.3.3 The Logical (Horizontal) Hierarchies

The *logic hierarchy* (Table 2.4) structures what happens at each level of the software hierarchy, in any specific class of applications [12]. Associated with it is a *data hierarchy* as shown here, which gives the specific data related to a specific class of applications.

The key thing here is the algorithms that act on the data, specifying precisely what operation is to be performed. As explained by Turing (see the quotes in Sect. 2.1), it is these algorithms that shape the computation. What also matters then is the order in which they are implemented, and on what specific data, something which is determined by the operational context of the program as a whole, which implements the computational logic used to tackle the problem of interest.

This generic logical structure enables the logic of any specific application domain to be represented by specific variables and associated operations. Thus one might be engaged in word-processing, numerical calculations, digital image manipulation, digital sound operations, computer-aided design, and so on. A specific high level language will enable modeling of each such domain, representing the hierarchical relations of its specific structure and appropriate operations on them. For example, in the case of word-processing, one might have the data structure in Table 2.5.

The word-processor program enables insertion, edition, deletion, copying, and pasting at any level of the data hierarchy, thus enabling manipulation of the parts (words), their components (letters), and their integration into higher order entities

Table 2.5 The hierarchical structure of specific applications

Book	Specific purpose
Chapters	Major themes
Paragraphs	Subthemes
Sentences	Logical units
Phrases	Logical subunits
Words	Representational variables
Phonemes	Variable components
Letters	Logical atoms
Binary code	Digital representation

such as paragraphs and chapters. It also allows detailed formatting of the resulting text. Because of the power of language, this will enable representation of anything humans can think about, and the associated logic of that domain (science, art, philosophy, whatever).

Application Domain Logic. Word processors, music programs, image manipulation programs and so on handle general classes of data in an appropriate way: all that is required is data entry, storage, recall, editing, and deletion, with appropriate application programs and hardware to output the result. But there are many application domains with their own specific logic: mathematics, engineering, environmental issues, ecological modelling, computer-aided design, statistical data analysis, and so on. Examples chosen at random can be found in [15, 55].

These are logical hierarchies which apply generically to a class of applications. Finally, there is the *systems hierarchy* (Table 2.6) showing how these hierarchies relate to each other whenever an application program is utilised in a specific operational context. This is where *systems analysis* [9, 12] comes in: structuring the hardware, programmes, and data to suitably model some specific real world problem that needs to be solved. Design patterns [31] characterize the high level possibility structures. Thus the program structures and data model the logic of many application areas. A key issue is where these multifold logical structures come from. I consider this in Sect. 2.7.5.

2.3.4 The Relation Between the Two Hierarchies

How do the implementation and logical hierarchies relate to each other? When we load and then run a high level program, these input operations take place at the uppermost level of the implementation hierarchy (Table 2.2), representing the problem logic at that level. Compilers or interpreters [3, 6, 67] then cause all the lower implementation levels to spring into action in accord with the logic and data that has been loaded at the top level. When this occurs, the same hierarchical logical structure is represented at each of the levels of the implementation hierarchy, written in a different language at each level, using quite different kinds of commands and data representation. The operating system orchestrates the way this happens [60].

Table 2.6 The systems hierarchy: the flow of causation when tackling a specific problem. It is driven by the nature of the user's problem, which gets translated into a computer application analysing specific data according to the internal logic of the problem

User level	Specific purpose	Goal
Logical level	Problem structure	↓
Programme level	Particular programmes	↓
Data level	Specific data	↓
Physics level	Hardware	Electrons

Thus the logical hierarchy of Table 2.4, as related to the specific problem at hand, recurs at each of the implementation levels of Table 2.2 in different forms. At the lower levels they are based on a set of simple logical operations, and it is very difficult to see the higher level logic which is shaping what is happening. For example:

- A word processor has high-level commands for insert, delete, copy, paste, italic, bold, change font, and so on. These are the user interface commands.
- An underlying Java Virtual Machine has instructions for the following groups of tasks: load and store, arithmetic, type conversion, object creation and manipulation, operand stack management (push/pop), control transfer (branching), method invocation and return, throwing exceptions, and monitor-based concurrency. The word-processing requirements are implemented in terms of these operations.
- These are implemented in the assembly language by commands such as MOV AL, 61h [Load AL with 97 decimal (61 hex)].
- These are implemented at the machine code level by commands such as 000000 00001 00010 00110 00000 100000 [add the contents of registers 1 and 2 and place the result in register 6].

When the machine code is run, it implements the commands at the binary level, and that induces more complex higher level commands at each higher level, thus causing the desired emergent behaviour. Why does it happen that the desired behaviour emerges? Because the system has been set up to ensure that this will be so!

Each higher level behaviour emerges from the lower level ones. But what ultimately determines what happens? The higher levels drive the lower levels. First, compilers or interpreters [3, 6] translate the higher level languages to the lower level languages. Then the lower levels implement the compiled program in a purely mechanistic bottom-up way and the desired higher level behaviour emerges from the combination of lower level operations. But those lower level states would not be there if they had not previously been determined top-down by the process of compiling a set of algorithms written in a higher level programming language. Their logic determines what happens.

Principle C3. The software drives the hardware. What specific physical interactions take place at the hardware level is controlled by specific data entered, in accord with the logic of the relevant algorithms.

It is this logical structure that is the key causal element in the sense of determining what happens next at each instant. Physical interactions in the computer are controlled by the logic of the algorithms. For example one might have an accounting system, in which case the logic of accounting systems determines what happens, or one might be modeling a chemical engineering system, in which case the logic of chemical interactions drives the system. At the lower levels, the logic of operations such as copying, deleting, and sorting determines what happens. Algorithms such as Quicksort replace physics equations as the driving logic.

The specific physical realisation is what enables this to work, but a different realisation could have been used. The essential nature of the program driving the computer is the equivalence class of all such functionally equivalent realisations (see Sect. 2.5).

2.3.5 Causality in the Hierarchies

Software determines what specific currents flow where and when in the hardware circuits, implementing the specific abstract logic that applies to the issue at hand. This logic is coded in a hierarchical fashion through the program and its subroutines or procedures, which embody its modular structure (Sect. 2.2).

The underlying abstract high level logic, for example, that of data compression or numerical analysis or pattern recognition, shapes the algorithms used. This determines what happens at the lower levels. This is clearly top-down causation from the higher to the lower levels. It will be explored further in Sect. 2.4. The specific outcome depends on the data supplied, which has to be hierarchically structured as required by the applications software.

Software S is not a physical thing, neither is data. They are realised, or instantiated, as energetic states in computer memory. The essence of software does not reside in their physical nature: it is the *patterns* of states, instantiated by electrons being in particular places at a particular time, that matters. These are not the same as those electrons themselves (just as a story is not the same as the paper on which it is written). Given the set of connections in the CPU, the pattern of electrons represents the logical structure of the program.

Programs and data together determine what specific electrical operations take place in the transistors and other physical components (level 0) in the chosen hardware, which is the context within which the software is causally effective. Thus the conclusion is:

Causal Effectiveness of Non-Physical Entities. In digital computers, non-physical entities control the behaviour of physical systems.

This will be explored further in Sects. 2.5 and 2.7.

2.4 Bottom-Up and Top-Down Causation

True complexity emerges through the interplay of bottom-up and top-down effects in the hierarchies of structure and causation [26, 27].

Bottom-Up Action. A fundamental feature of the structural hierarchy in the physical world is bottom-up action: what happens at each higher level is based on causal functioning at the level below, so what happens at the highest level is based on what happens at the bottommost level. This is the profound basis for reductionist world views. The successive levels of order entail chemistry being based on physics, material science on both physics and chemistry, geology on material science, and so on. In the case of computers, such bottom-up action is the basis of the emergence of high level languages and applications from the underlying physical and logical components. However, this only takes place once the scene has been set by processes that design the structure and so channel the lower level interactions.

Top-Down Action. The feature complementary to bottom-up action is top-down action. This occurs when the higher levels of the hierarchy direct what happens at the lower levels in a coordinated way [28]. For example, pressing a computer key leads to numerous electrons systematically flowing in specific gates and so illuminating specific photodiodes in a screen.

Generically, specifying the upper state (for example, by pressing a computer key) results in some lower level state that realises this higher level state, and then consequent lower level dynamics ensues to produce a new lower level state in a way that depends on the boundary conditions and structure of the system. The lower level action would be different if the higher level state were different. It is both convenient and causally illuminating to call this top-down action, and to represent it explicitly as an aspect of physical causation. This emphasizes how the lower level changes are constrained and guided by structures that are only meaningful in terms of a higher level description.

There are five different types of top-down causation (TD1–TD5) in the logical hierarchies, with differing characteristics. The following subsections consider them in turn. I look successively at:

- The combination of bottom-up and top-down action (Sect. 2.4.1).
- TD1: Deterministic top-down processes (Sect. 2.4.2).
- TD2: Non-adaptive feedback control systems (Sect. 2.4.3).
- TD3: Adaptive selection (Sect. 2.4.4).
- TD4: Feedback control with adaptive goals (Sect. 2.4.5).
- TD5: Adaptive selection of adaptive goals (Sect. 2.4.6).
- Goals and learning in relation to these kinds of causation (Sect. 2.4.7).

2.4.1 The Combination of Bottom-Up and Top-Down Action

In the implementation hierarchy, algorithmic processes in the bottom layers enable what happens, through suitably structured electronic interactions at the machine level combining to create the emergent stack of virtual machines (Table 2.3). But top-down control determines what happens according to the logic of the high level programs that happen to be running (music programs, image manipulation, numerical analysis, pattern recognition, or whatever). The mechanisms enabling this to happen are compilers and interpreters, as explained in Sect. 2.3.4: they transfer the application logic down from the higher to the lower implementation levels, which are all virtual machines except for the bottommost level (Table 2.2). At that level, this logic is represented as patterns of electronic excitations.

This combination of bottom-up and top-down actions enables complex higher level behaviour to emerge from simpler lower level processes, which are orchestrated from above by entering suitable data at the keyboard. That action directly alters specific memory registers, which either contain data for the program, or instructions as to what should happen next, as in Turing's description (see Sect. 2.1). Which it is

Table 2.7 The hierarchy in data communications

Level 7	Application	Message, HTTP/ SMTP /FTP
Level 6	Presentation	Data compression/encryption
Level 5	Session	Data delimitation, synchronisation
Level 4	Transport	Segments, TCP
Level 3	Network	Datagrams, IP
Level 2	Link	Frames, Ethernet/WiFi/PPP
Level 1	Physical	Individual bits, protocols

Table 2.8 Bottom-up and top-down action in the hierarchy of data communications

Source			Destination
Level 7	Application	⇒ ⇒ ⇒ ⇒ ⇒	Application
Level 6	↓ Presentation		↑ Presentation
Level 5	↓ Session		↑ Session
Level 4	↓ Transport		↑ Transport
Level 3	↓ Network	Routers	↑ Network
Level 2	↓ Link	Link layer switch	↑ Link
Level 1	Physical	⇒ ⇒ ⇒ Cable/wireless ⇒	Physical

depends on the context of the physical system and the pattern of excitations in all the other gates that embodies the system logic.

Emergence of Same-Level Action. The emergence of same-level action through this combination of bottom-up and top-down effects is particularly clear in the case of computer networking [40]. The internet protocol stack/OSI model is shown in Table 2.7 (levels 5 and 6 are in the OSI model). Sending a message from the source to the receiver, the process is top-down at the source: the message gets sent down from level 7 to level 1, the representation being transformed on the way down from alphabetic at level 7 to binary at level 1, and also split into packets with headers and tailers. It is sent in this form to the receiver.

A reverse bottom-up process takes place at the destination: the binary digital level 1 form gets transformed to a properly formatted output form at level 7. Encapsulation takes place: extra information is added at each level on the way down, and stripped on the way up [40]. Thus the result (Table 2.8) is effective same-level action: the message sent by the source is received in the same form at the destination. This is a good model of how same-level action emerges in general from a combination of top-down and bottom-up action.

2.4.2 TDI: Deterministic Top-Down Processes

In deterministic processes in a computer, the outcome is uniquely determined by initial and structural conditions. Data must be chosen to respect the logical conditions specifying legal data and item length limits, but it can vary arbitrarily within those

Table 2.9 The basic features of deterministic causation in a digital computer. Given the context (structural conditions and data constraints), the initial data leads to a unique final state if the calculation halts

Context	Date typing	Structural conditions
<i>Constraints</i>	↓	↓
Data	Initial data	Computation
<i>Closed system</i>	↓	↓
Outcome	Final state (deterministic)	

constraints. In fact, it varies between different runs, but is fixed for each run, and it cannot change once the run has started. That is why the dynamics is deterministic: the outcome is fixed by the input, with no uncertainty, providing the calculation halts. This is top-down causation, because the outcome depends on the context: alternative higher level states (structuring or input data) lead to different outcomes (Table 2.9).

Such computations simulate many different aspects of reality, e.g., predictions of stock control in a shop or factory, nucleosynthesis in stars, aircraft paths, weather patterns, future activity in the stock exchange. Initial data plus the algorithm determines the outcome at each time step $t_{i+1} = t_i + \Delta t$: for a system of variables $y_j(t)$,

$$y_j(t_{i+1}) = f_j(y_1(t_i), \dots, y_N(t_i), \Delta t) \; ,$$

(2.1)

there is no uncertainty in the model. But there are rounding errors affecting the outcome if it involves continuous variables, depending on the size of the step Δt . The stability of the outcome depends on whether the system modelled is stable or chaotic: attractors stabilize outcomes, strange attractors destabilize. In simulating real world systems such as stock in a shop, an aircraft in flight, or the weather, one can make the model more accurate by updating the data on an ongoing basis. Then the outcome is no longer a unique outcome of the initial data. This is the route to feedback control (TD2), discussed below (Table 2.10).

Random Initial Data. An interesting twist is to add in a random number generator to vary initial data. One uses a random seed to initialize the generator. This is a new number, unrelated to the problem domain, e.g., the time of the start of the program, or data from the weather or the atmosphere. It is chosen separately for each run and is then fixed for that run. This enables Monte Carlo simulations by choosing a whole series of runs where the seed is varied randomly but the rest of the data is fixed.

This can simulate a set of objects with varying unknown properties. Each run is deterministic, but the overall run is not. However, it shows statistical trends, which are then themselves deterministic at a higher level. These are emergent properties of

Table 2.10 Randomness and determinacy in statistical investigation

Statistical description	Statistical laws	Deterministic
↑	<i>Coarse grain</i> ↑	
Ensemble	Many cases	<i>Random</i>
↑	<i>Repeat with variation</i> ↑	
Individual case	Dynamics	Deterministic

an ensemble of individual lower level systems. But in the bigger scheme of things, this is still deterministic: the random number generator is not random if we take into account causal processes in the environment that determine the seed.

Indeterministic Processes. There is, however, the possibility of introducing genuine randomness into algorithmic computational systems. Here one uses quantum uncertainty to generate the seed: detection of radiation resulting from radioactive decay of atoms is used to generate a random number.² Then it is truly random: there is no cause for its value, provided that standard quantum theory is correct (see Sect. 6.1). The specific result of each run is not predictable from the initial data, although it must lie in the possibility space set by the algorithms. Thus it is algorithmic, but not deterministic: it is not mechanistic in the classical sense.

2.4.3 TD2: Non-adaptive Feedback Control Systems

By contrast, goals are the essence of feedback control systems. Non-adaptive control systems compare the actual present state of the system with a desired goal and feed information back to a controller to correct the system state (see Table 2.11). This is the essence of cybernetics: feedback control corrects any error in the system state (i.e., any deviation from the desired goal) by observation and measurement, continually using new data to keep it on track.

In contrast to the case just discussed (TD1), the initial data is irrelevant here. It is the full set of goals g_n that determine the outcome, through the differences $\Delta y_n(t_i)$ between the goals and the actual values. Instead of (2.1), we have

$$y_j(t_{i+1}) = f_j(\Delta y_1(t_i), \dots, \Delta y_N(t_i), \Delta t) \text{ , } \Delta y_n(t_i) := y_n(t_i) - g_n \text{ . } \tag{2.2}$$

Examples are thermostats, an elevator taking one to the desired floor in a building, speed controllers in engines, fully automated electric trains, and so on. In many engineering applications, there will be computer control systems that will implement this logic of deciding what to do next on the basis of the current system state, embodied at the microscale in WHILE and IF THEN loops [14, p. 29].

Table 2.11 The basic features of a feedback control system. The goals lead to a specific final state via feedback of an error signal to an actuator. The initial state of the system is irrelevant to its final outcome, provided the system parameters are not exceeded

	Controller	⇐ Correction signal	
Noise ⇒	Action ↓	Feedback ↑	
	State	⇔ Comparator ⇔	Goal

²The Hotbits random number generator uses this technique: see <http://www.fourmilab.ch/hotbits/>.

In advanced systems (automatic pilots, control systems in chemical plants), the controller will act not on the basis of the present physical state of the system but on the basis of predicted future states as determined by the latest updates of the current system state. It is this continual updating of predictive data that gives the process its power. This is also the core principle of numerous homeostatic control systems in physiology and cell biology. This is top-down causation because the goal determines the outcome, and hence is at a causally higher level than the system controlled. It is an emergent property of the system, enabling sophisticated behaviour. But this process cannot innovate: the outcome is predictable from the outset, as it is determined by the explicit or implicit goals of the system. Like predictive algorithmic processes, non-adaptive feedback control systems cannot learn. That requires adaptive selection.³

2.4.4 TD3: Adaptive Selection

The basic feature of adaptive selection [36] is that a process of variation generates an ensemble of states, from which a best outcome is selected according to some selection criterion (see Table 2.12).⁴

The reason this is classed as a form of top-down action is that the nature of the higher level environment is crucial to using selection criteria. The outcome would be different if either the environment or the criteria were different. Its great power in evolutionary biology is due to the continued repetition of the adaptation process, with the best variant being passed on from one generation to the next by a hereditary mechanism. But that repetition is not essential to the basic process.

The basic dynamics is first a randomisation process, and then a selection process

$$y_j(t_{i+1}) = \Xi_j(y_1(t_i), \dots, y_N(t_i), c_j, E) \; ,$$

(2.3)

Table 2.12 The basic features of adaptive selection. Selection takes place from an ensemble of states, the selection being based on the outcome of some selection criteria in the context of the specific current environment. Unwanted states are discarded

System states	⇐	Selection agent selects state		Meta-goals
Variation ↓		↑		↓
Ensemble of	⇒	Preferred	⇐	Selection
System States		states		criteria
		↑		
		Environment		

³I am aware that some present day feedback control systems use principles of adaptive control. I believe they should be labeled as such, to distinguish them from the basic cybernetic processes identified by Wiener, in which the goal is fixed.

⁴This is what Penrose identifies as bottom-up organisation [53, p. 18], but this is incorrect, because he fails to recognise the top-down nature of the decision process via higher level selection criteria.

Table 2.13 The basic function of adaptive selection processes: they select what is useful or meaningful from an ensemble of mainly irrelevant stuff and reject the rest, thus creating order out of disorder by selecting states conveying meaningful information

Final data set	Meaningful information	\Rightarrow	Rejected set: noise
	Selection \Uparrow	\Leftarrow	Selection principle
	Varied set		
	Variation \Uparrow	\Leftarrow	Variation principle
Initial data set	Ensemble of states	Random	

where Ξ_j is a projection operator selecting one of the $y_n(t_i)$ and rejecting the rest, on the basis of the selection criterion c_j evaluated in the environmental context E . It is a non-deterministic process: because of the random element in generating the ensemble selected from, one cannot predict the outcome before the selection process takes place.

It is also for this reason that it can innovate. The process generates new information that was not there before—or rather, finds information that was hidden in noise (Table 2.13). That is the general process whereby adaptive selection generates useful information: it finds what is relevant and works from an ensemble of stuff that is mainly irrelevant or does not work, hence allowing a local flow against the general tide of increasing disorder. Inter alia, this is the process underlying learning.

Many computational processes build on this possibility. These include:

- *Artificial neural networks* [11], where selection of node weights occurs through the training process. The resulting set of node weights is not predictable. (If it were, one would not need the training process.)
- Many *optimization procedures* are of this nature, as they search the possibility space and choose the best outcome encountered. Randomness comes because one cannot explore the whole space, and we have to choose a subset of points to investigate, and steps away from these points: the result might depend on this choice, if local maxima occur.
- *Evolutionary computation* (EC) [23, 24] encompasses genetic algorithms (GA), evolution strategies (ES), evolutionary programming (EP), genetic programming (GP), and classifier systems (CS).

These are all examples of non-deterministic computing [1, pp.412–413]:

The key idea is that expressions in a non-deterministic language can have more than one possible value [...] our non-deterministic programme evaluator will work by automatically choosing a possible value and keeping track of the choice. If a subsequent requirement is not met the evaluator will try a different choice, and it will keep trying new choices until the evaluation succeeds, or we run out of choices. [...] the non-deterministic evaluator will free the programmer from the details of how the choice is made [...] it supports the illusion that time branches, and that our programmes can have different possible execution histories. When we reach a dead end, we can revisit a previous choice point and proceed along a different branch.

This is just a version of adaptive selection.

Table 2.14 Adaptive selection of goals

Level 3	Selection criterion	Meta-goal
	↓	
Level 2	Goal	Adaptively selected
	↓	
Level 1	Feedback control	⇒ Output

2.4.5 TD4: Feedback Control with Adaptive Goals

Higher level innovation becomes possible when one combines TD2 and TD3 to obtain TD4: feedback control with adaptive learning. Unlike TD2 where goals are fixed, these are feedback control systems that select their goals by a process of adaptive selection: equation (2.3) is applied to a set of goals g_n in (2.2) to get

$$g_j(t_{i+1}) = \Xi_j^g(g_1(t_i), \dots, g_N(t_i), c_j^g, E), \quad (2.4)$$

where c_j^g are criteria for feedback control goals (see Table 2.14).

This is a higher level form of top-down action, as it involves both goals in a homeostatic system (TD2) and adaptive selection criteria (TD3). It is used in engineering in adaptive forms of feedback control, which can be implemented through suitable digital computer systems.

2.4.6 TD5: Adaptive Selection of Adaptive Goals

One issue inevitably arises: where do the selection criteria in adaptive selection systems come from? In fact, they, too, may be adaptively selected, giving TD5: the case where adaptive selection criteria are determined by adaptive selection. Hence, (2.3) is applied to the criteria c_n [guiding selection in (2.3)] in the form

$$c_j(t_{i+1}) = \Xi_j^c(c_1(t_i), \dots, c_N(t_i), c_j^c, E), \quad (2.5)$$

where c_j^c are criteria for selective criteria (see Table 2.15).

This is a higher form of top-down causation, because adaptive selection is itself a form of top-down causation. It is of importance in determining strategy in every area of personal and communal life, e.g., business, education, politics, social policy. It can be exemplified by ranking systems in search engines [45], where the key element is

Table 2.15 Adaptive selection of selection criteria

Level 3	Selection criterion 2	Meta-goal
	↓	
Level 2	Selection criterion 1	Adaptively selected
	↓	
Level 1	Adaptive selection	⇒ Output

Table 2.16 The hierarchy of selection criteria

Level $N + 1$	Selection criterion N	Non-algorithmic choice
	\Downarrow	
Level N	Selection criterion $N - 1$	Adaptively selected
	\Downarrow	
	\vdots	\vdots
	\Downarrow	
Level 3	Selection criterion 2	Adaptively selected
	\Downarrow	
Level 2	Selection criterion 1	Adaptively selected
	\Downarrow	
Level 1	Adaptive selection	\Rightarrow Output

selection of criteria for ranking, and the second order adaptive outcome is successful ranking of web pages (selection of the most relevant according to the chosen criteria).

Closing the Hierarchy. Adaptive selection of adaptive criteria involves choosing a set of criteria c_j^c for suitability of adaptive criteria c_j . This appears to be the start of an infinite recursion: where do these next higher level selection criteria c_j^c come from? Are they, too, selected adaptively? How do we close the logic? (see Table 2.16).

At some point we have to stop and accept a set of highest level selection criteria as an a priori choice, otherwise we cannot close the system. (if we consider criteria for this choice and evaluate it, then through that act it is shown not to be the uppermost level). Any attempt to determine these criteria algorithmically, heuristically, or by adaptive selection will of necessity introduce a further set of selection values: it will just postpone the final decision level and choice by adding in a further level to Table 2.8. Naturally, the same issue arises in relation to adaptive selection of goals (TD4). There, too, there has to be an uppermost level which is just taken as given and sets the overall direction and purpose of the dynamics. The meta questions are:

- **Meta-analysis.** How many levels up do you go?
- **Choice.** How do you decide which criteria to use at the top?

These are philosophical issues, to be chosen according to one’s philosophical position. This is where values and purpose come in: this highest level is the level of meaning (‘telos’), perhaps involving ethics or aesthetics. This choice gives shape to all the rest, for it transfers down to affect choices made and outcomes at all the lower levels.

2.4.7 Goals and Learning in Relation to These Kinds of Causation

This section has looked at five distinct types of top-down causation (TD1–TD5) that can occur in computer systems. Three key points to notice are the following:

- Goals versus attractors.
- Learning and adaptive selection.
- Intelligent top-down causation.

2.4.7.1 Goals Versus Attractors

Dynamical systems with attractors (TD1) can look like feedback systems with goals (TD2) because initial conditions anywhere in a wide basin of attraction can lead to the same result. In particular this happens if there is friction (motion dies away as energy dissipates). Nevertheless, they are completely different in terms of mechanism: the second (TD2) involves active collection and use of information, while the first (TD1) does not. The second involves the causal effectiveness of goals as in (2.2), the first just the flow of the dynamical system according to the initial data as in (2.1).

2.4.7.2 Learning and Adaptive Selection

Learning, and associated collection of new information, is not possible via bottom-up action alone, or via dynamical systems (TD1) or non-adaptive feedback control (TD2). TD1 proceeds simply on the basis of information that is available at the beginning, as in (2.1), while TD2 compares updated information with goals as in (2.2). Neither generates any new information that was not there to start with. In order for new information to be acquired, and hence in order that learning can occur, one needs adaptive selection to take place, that is one needs TD3 as in (2.3), TD4 as in (2.4), or TD5 as in (2.5).

2.4.7.3 Intelligent Top-Down Causation

Intelligent top-down causation is the special case of any of TD1–TD5 where symbolic systems are used in the analysis, based on using some entity to represent something else. This is what characterizes intelligent thought: systems and situations are modeled in a symbolic way through use of language, diagrams, maps, physical models, or mathematical models. In particular higher level goals and selection criteria are analysed through use of symbolic systems and then adapted to get optimal results.

This use of symbols is an abstract technology that enables us to transcend the boundaries of what actually exists and consider what might be, what it might mean, and what methods to use when investigating these issues. The use of symbolic systems—particularly language—is a key characteristic of being human [22].

Now all digital computer systems are symbolically based—that is the core of how computers function—so their use to assist decision-making is in a sense automatically of this kind. However, sometimes computers act as explicitly symbolic computational systems, rather than just carrying out data analysis or numerical computations. Computer languages such as LISP can be used to perform logical operations and so can

be used to investigate goal choice and decision-making algorithmically. Their mathematical derivatives, such as MATHEMATICA and MAPLE, are able to perform algebraic operations (solving an equation symbolically, for example) and symbolic integration and differentiation, as opposed to numerical differentiation and integration. The former hold for generic functions whereas the latter hold only for specific functions. Such languages are of considerable use in evaluating goals and adaptive criteria.

At a deeper level, computer systems act crucially as extensions of human capacity in investigating policy options symbolically: computers are used as interactive aids in decision or design systems, and it is in the human–computer interaction that the real creative capacity lies. Computer models are used to simulate reality, e.g., computer-aided design systems for houses or aircraft: the human mind intervenes and tries new options, the best one being selected. Examples are health policy, housing policy, energy policy, environmental policy. In each case examining what is possible when physical and economic constraints are taken into account can play a key role in determining what are suitable tactical and strategic goals, and indeed in working out what are the best criteria for such goals. This is particularly because of the unintended consequences that can arise in complex systems such as ecosystems: you aim for one effect, but a completely unexpected side-effect dominates the outcome. Neither the options nor the selection process can be fully algorithmic, because the former involves imagination and understanding of causal possibilities, and the latter involves decisions that cannot be reliably reduced to a numerical algorithm, for example, an architectural design involves aesthetic as well as functional features. When they are so reduced (as in the case of automated stock options), disaster may ensue.

The core causal feature is the interaction of the user and the machine, the resulting evaluations being based on models of the target area embodied in suitable symbolic systems. These evaluations then become the high level causal feature underlying our plans and consequent actions that are physically effective in the real world. One attains new patterns that were not there before by optimization and selection of goal choices, selection criteria, and methods used.

A key feature of such reasoning is that it is recursive: it can be turned on itself, to adapt the method of reasoning. An open question concerns the degree to which intelligent computer systems can capture the kind of human reasoning involved in such analysis. This is of course the contentious area of artificial intelligence [48, 56, 58]. I will not enter the fray except to give the following quote from McCarthy [48, p. 18]

Formalizing common-sense reasoning needs contexts as objects, in order to match human ability to consider context explicitly. [...] We propose the formula *holds*(p, c) to assert that the proposition p holds in context c . It expresses explicitly how the truth of an assertion depends on context.

Thus a key to success is adapting the logic to take contextual effects into account, in line with the central argument of this book. There is, however, a specific open question as regards TD5:

Open Question. Is adaptive selection of adaptive goals *only* possible through use of symbolic systems? Or can it be possible without symbolic reasoning?

I suspect the answer is that symbolic reasoning is essential for meaningful TD5 processes. Then TD5 is necessarily a subclass of intelligent top-down causation.

2.5 The Core Feature: Equivalence Classes

The central feature of all forms of top-down causation in general is multiple realization and the associated equivalence classes [5]. This applies in particular to digital computation. I consider in turn:

- Multiple realization (Sect. 2.5.1).
- The link to top-down causation (Sect. 2.5.2).
- The ontological nature of computer programs (Sect. 2.5.3).

2.5.1 Multiple Realization

The core feature of top-down causation is the way higher level elements can emerge from many different variants of lower level ones, in both the physical and the logical context:

Multiple Realisability. Higher level structures and functions can be realised in many different ways through lower level entities and interactions.

In general many lower level states correspond to a single higher level state, because a higher level state description is arrived at by averaging over lower level states and throwing away a vast amount of lower level information (coarse-graining). Hence, specification of a higher level state determines a family of lower level states, any one of which may be implemented to obtain the higher level state (a light switch being on, for example, corresponds to many billions of alternative detailed electron configurations). The specification of structure may be loose (attainable in a very large number of ways, e.g., the state of a gas) or tight (defining a very precise structure, e.g., particular wiring of a VLSI chip in a computer). In the latter case, both description and implementation require far more information than in the former. Equivalence classes of lower level operations give the same higher level effect. Some examples in the case of digital computers are:

- At the circuit level, one can use Boolean algebra to find equivalent circuits to any circuit [47].
- At implementation level, one can compile or interpret a high level program, giving a completely different lower level process producing the same higher level outcome [67].

- One can run the same high level software on different microprocessors, using different instruction sets [47].
- One can run the same algorithms in different programming languages (Basic, Fortran, Pascal, Java, for example).
- Generic procedures can operate on data represented in different ways [1, pp. 170, 187].
- At implementation level, there is an equivalence of hardware and software. One can decide to imbed developed software in a dedicated hardware chip, giving a completely different nature of lower level physical entities for the same higher level outcome.
- At the highest level, specific tasks can be allocated either to the user or to the computer to give the desired high level output (e.g., focusing and exposure in digital cameras).

At the foundations of computing, the notion of a computable function is extremely robust and can be defined in many seemingly different, but equivalent terms [14]⁵:

- One of these definitions is Turing’s original definition via Turing machines that can encode numbers in the form of digits on infinite tapes that the machine can manipulate according to actions specified by its program.
- An equivalent definition is via register machines that can directly manipulate natural numbers with arithmetic operations. This is close to the (assembler) programming language.
- Another completely different but equivalent definition is purely number theoretic, avoiding reference to any kind of seemingly obscure ‘machinery’: a computable function is a function whose graph is a Diophantine set
- Another common characterization used in logic is via certain recursive equations, which is why the word ‘recursive’ is used synonymously with ‘computable’ in this field.

This variety of ways expresses the notion of a computable function from quite different, but nevertheless equivalent viewpoints.

2.5.2 *The Link with Top-Down Causation*

The connection with top-down causation is that we only normally have access to the higher level variables: these are the handles we have to affect the system state. When we change them we change numerous lower level states in accordance with the chosen higher level state, that is, we instantiate an instance of the equivalence class. It does not matter which specific one we instantiate. What matters is which equivalence class it belongs to, because this determines which higher level state it represents.

⁵I thank Vasco Brattke for these characterisations.

In the implementation hierarchy, once a particular lower level method of implementation has been chosen, that is the one that exists physically and drives the higher level dynamics.

In logical hierarchy, the higher level function drives the lower level design and hence the lower level operation. This is embodied in the nature of modularity, involving encapsulation and information hiding [57, pp. 233, 476–483]: one can change the nature of the private methods, while overall function and interface remain unchanged:

Equivalence Classes. Top-down causation takes place by instantiating a specific lower level instance of an equivalence class representing a higher level variable. This happens by giving a higher level variable a specific value, an action which sets specific values for all the relevant lower level variables. Which specific such values are set is not determined by the chosen high level value.

2.5.3 *The Ontological Nature of Computer Programs*

Because of this multiple realisability, a higher level element is not ontologically the same as any specific lower level realization. It is the equivalence class of all of them [5]:

The Ontological Nature of a Computer Program. In terms of the lower level elements that represent or instantiate it, this is nothing other than the functional equivalence class of such lower level elements that give the desired high level function.

This characterizes in precisely what way computer programs are abstract entities. They are not the same as any specific physical state: they are in essence equivalent to the set of all physical states that embodies their logic:

Reality of Computer Programs. They are real and exist as higher level entities, because the equivalence class of lower level elements exists, and is causally effective. It determines uniquely what happens at the macro level.

The same is true for data: it can be represented logically in many different ways, e.g., binary or hexadecimal. It can be instantiated physically in electronic states or in printed or spoken form. The essence of the data is not any specific representation of either equivalence class: it is the equivalence class itself.

2.6 Resources: Memory and Deleting

Formal language theory proposes that there are an infinite number of possible statements in any language [38, p.320]. This is based on the idea that statements can have an unbounded length: one can always add another clause to them. In the case of computers, the tape in a Turing machine is supposed to be infinite: it can store an infinitely long programme and an infinite amount of data. But infinities cannot occur

in physical reality: resources are limited and in reality infinity is unattainable. This has important practical applications for computing. I consider in turn:

- The unphysical nature of infinity (Sect. 2.6.1).
- Deletion and garbage collection (Sect. 2.6.2).
- The memory hierarchy (Sect. 2.6.3).
- Modular hierarchical structure and scoping of variables (Sect. 2.6.4).
- Deletion, adaptive selection, and irreversibility (Sect. 2.6.5).

2.6.1 *The Unphysical Nature of Infinity*

Turing states [65]:

Some years ago I was researching on what might now be described as an investigation of the theoretical possibilities and limitations of digital computing machines. I considered a type of machine which had a central mechanism, and an infinite memory which was contained on an infinite tape. This type of machine appeared to be sufficiently general [...] It was essential in these theoretical arguments that the memory should be infinite. It can easily be shown that otherwise the machine can only execute periodic operations.

But an infinite memory or an infinite tape cannot be read. Infinity is not just a very large number: it is a magnitude that is never attained. It is always beyond reach. That is its most essential feature. No matter how much has been read, there will always be more to read, because that is what infinity means—something that is never completed, it is always unattainable. David Hilbert remarked [35]:

The infinite is nowhere to be found in reality, no matter what experiences, observations, and knowledge are appealed to.

A real computer has finite storage capacity and only survives for a finite length of time, and so can only carry out a finite number of operations in its lifetime.

One can calculate an absolute limit to what a computer can possibly read in its lifetime by estimating how many bytes can be read by a machine that reads continuously for 24 hours a day, every day for say 1200 years at a rate of say 10^9 bytes a second, giving $10^9 \times 60 \times 60 \times 24 \times 365 \times 1200 = 378\,432\,000\,000\,000\,000$ bytes: a large number but obviously not infinite. No real computer can exceed this limit in its lifetime (inter alia because it will need maintenance, and will not in fact last that length of time). Indeed the computational capacity of the entire universe is finite [44]. Hence, there is a finite limit to the length of any statement that could be read by a computer in its entire lifetime in a physically realistic setting. And anyway, sentences actually usable for computational purposes, the *raison d'être* of computers, are very much shorter:

Computational Finiteness. The set of possible computable programs Ω_p and the set of potentially associated data Ω_d are both large but finite.

The implication is that there are a finite number of possible computer languages, programmes, and data, whence the possibility space for computer operations is finite. The idea of a computer that can process an infinite tape or read an infinite amount of data does not make physical sense. Formal language theory should take this into account.

What about Turing's comment that if a machine has finite memory then it can only execute periodic operations? This is in principle true, as the operation space is then compact, and if the machine continues to operate for an unlimited time, Poincaré's eternal return theorem applies: eventually all possible states will have been visited and the next and all subsequent ones will be repeats of ones already utilised, so cycles will occur. But this assumes that the machine will continue operating for an infinite time, something which cannot happen *inter alia* because the Earth will come to an end in a finite time, when the Sun comes to the end of its life. The alleged problem arises because of this implicit infinity, which is unphysical. Computer memories are now so large that this will not be an inevitable outcome in practice.

2.6.2 *Deletion and Garbage Collection*

In practical terms, this limitation on memory has important implications for how memory is handled, and leads to the need for garbage collection and the ongoing deletion of records, freeing up memory space for reuse.

Garbage Collection. During a program run [1, pp.540–546], this is a key strategy for handling memory limits, giving the illusion of infinite memory even though in fact the memory space is finite. Memory cells used to hold intermediate results during a calculation can be cleared at the end of the calculation, freeing up memory space to be reused in the next calculation.

This is related to *persistence* [12, pp.75–77]: keeping in memory objects and names across different contexts. Objects take up some amount of space and exist for a particular amount of time. But one has to clear them out to make room for new objects, or memory will fill up and operations will cease.

Deleting Records. As regards long term memory, deletion of records to free up memory is a key requirement, not just because storage space is limited, but also because otherwise we simply cannot handle the vast amounts of data we accumulate. We eventually forget we have stored specific data, or cannot locate the relevant records in the fog of data clogging up our machine. The key strategy here is that the user deletes all those records they don't want to keep and puts the rest into suitably formatted short term or longer term storage, depending on their usage needs. This process of sorting emails, music, digital images, and so on, deleting those that are unwanted and keeping those that are still useful, refines and organises our files into meaningful collections suited to our purposes.

Together with the organisational methods discussed in the following sections, deletion and reuse of memory is the key to handling memory limitations resulting

from finite resource availability, giving an illusion of infinite memory space, despite the available space being strictly limited.

2.6.3 *The Memory Hierarchy*

Given the hierarchy and memory limits, one still has to handle the practical limits on memory. This is done through the memory hierarchy. Turing states the problem as follows [65]:

A problem might easily need a storage of three million entries, and if each entry was equally likely to be the next required, the average journey up the tape would be through a million entries, and this would be intolerable. One needs some form of memory with which any required entry can be reached at short notice [...]. Another desirable feature is that it should be possible to record into the memory from within the computing machine, and this should be possible whether or not the storage already contains something, i.e., the storage should be erasable.

Even with more modern forms of memory, memory bottlenecks are the key design issue for computers. This breeds the memory hierarchy of short term, medium term, and long term memory. Thus one has [39]:

- **Main memory.** DRAM semiconductor memory in which most of the program and data are stored when the program is running (short term memory).
- **Cache memory.** Very high-speed semiconductor memory that caches frequently-used programs and data from main memory (storing them in a quick access area of medium term memory).
- **Paging memory.** Slower memory, usually disk, which provides swap files as an extra area for the main memory (medium term memory not used so often).
- **Hard drives.** Disk or tape memory for files (long term memory).

There is an entire science of how to design caches [37], and special languages designed handle the memory hierarchy efficiently [29]. As is clear from the above, a key issue is what to delete and what to keep. But additionally, a suitable hierarchical structure makes a big difference.

2.6.4 *Modular Hierarchical Structure and Scoping of Variables*

The fundamental principle is *locality of reference*, realised in modular hierarchical structures, with related aspects of temporal locality and algorithmic locality. One limits applicability of a variable both in logical space and in time. This is done by the mechanism of *scoping*, i.e., specifying the context within which it will be valid.

Scoping as Regards Context. Algorithmic locality happens via the distinction between local and global variables, embodied in the scope of a variable. Local variables must be readily available when a module is run, but can be cleared when another one is run. Global variables must be available all the time. The existence of the modules enables this distinction and so clarifies which variables can be cleared when the active module is changed.

Scoping Variables in Time. This follows from the fact that local variables are only valid for a certain time and cease to be needed when other local variables become relevant because another module is run. But there is another aspect: a key idea is that of a function $f(t)$ with an unchanging name, which keeps its identity as we evaluate it at different times, rather than regarding each of its values as separate ontological entities $x := f(t_1)$, $y := f(t_2)$, $z := f(t_3)$, etc. This allows one to overwrite old values of the variable as new values are calculated. One can discard the old value because it is no longer needed: what matters in most cases is just the value of the function at the present time, and perhaps a few times steps before that (if we are taking numerical derivatives). Exceptions are when the records are needed in the long term (financial or medical records for example), but then they can be transferred from short term memory to long term memory and stored on hard drives for later recall if necessary. Short term memory is freed up for reuse.

Streams. A related concept is the idea of delayed evaluation of streams [1, pp. 316–330]. These are lists which can be used to represent sequences that are infinitely long (such as the set of integers), even though in fact we only compute as much of the stream as we need to access [1, p. 326]. This is done by constructing streams partially and passing the partial list to the program that uses the list. Thus one writes the program as if the entire sequence was being processed, but interleaves the construction of the stream with its use. In this case, at the end of the calculation, there is no obligation to delete the variables that are part of the list but were never needed, because they were never activated in the first place.

2.6.5 *Deletion, Adaptive Selection, and Irreversibility*

The big picture is that (see Table 2.13):

One Creates Order by Deleting. Adaptive selection of what is meaningful, and hence creation of ordered meaningful information, is centrally based on deleting what is not wanted.

Examples are deleting old files and emails, as well as deleting old values of variables, and indeed no longer used variables themselves. This is what allows the freeing up memory for reuse, and so creates the illusion of infinite memory.

Irreversibility. As pointed out by Landauer [43], these processes are where irreversibility, associated with physical entropy production, happens in computations (quoted by Bennett [8]):

Any logically irreversible manipulation of information, such as the erasure of a bit or the merging of two computation paths, must be accompanied by a corresponding entropy increase in non-information bearing degrees of freedom of the information processing apparatus or its environment.

One is creating logical order by deleting (see Table 2.13), as regards information locally going against the overall flow of increase of entropy. There is a consequent physical energy cost characterized by the Landauer limit: the minimum amount of energy required to change one bit of information is given by $kT \ln 2$, where $k \sim 1.38 \times 10^{-23} \text{ J/K}$ is the Boltzmann constant and T is the temperature of the circuit in kelvin. This principle linking information and entropy creation has been experimentally verified by Bérut et al. [10]. Hence, there is an energy cost to generating useful information.

Ladyman et al. [41] analyse in detail what it means for a physical system to implement a logical transformation L , and make this precise by defining the notion of an L -machine. They show that logical irreversibility of L implies thermodynamic irreversibility of every corresponding L -machine. This relates in particular to the operation *Reset* which clears a logical system to its original state by replacing all variable values generated in the previous cycle with default values and so freeing it up to start a new cycle of operation. Overall, the conclusion is that dealing with logical infinity in a system of finite size is irrevocably tied to physical irreversibility.

2.7 The Outcome: Causation in Digital Computers

Even though computers are the epitome of algorithmic machine operations, they are also systems where non-physical entities (programs, algorithms, data) are causally effective, and enable symbolic operations to take place that are independent of the underlying physics. Here we consider:

- Computer programs are non-physical, but causally effective (Sect. 2.7.1).
- Computer programs embody abstract logic, and act top-down (Sect. 2.7.2).
- Room at the bottom (Sect. 2.7.3).
- Predictable explanation (Sect. 2.7.4).
- Possibility spaces and their causal effects (Sect. 2.7.5).
- Top-down action from the mind (Sect. 2.7.6).
- Genuine emergence (Sect. 2.7.7).

2.7.1 *Computer Programs Are Non-physical, but Causally Effective*

Virtual machines are the core of computing systems (Table 2.3), and although they do not exist as physical entities, they are real: they exist as causally effective entities.

2.7.1.1 The Non-physical Nature of Computer Programs

Computer programs are not the same as what is printed in a listing, or stored in a disc, or saved in computer memory, or presented on a blackboard, and neither are they what exists in a programmer's mind. These are all instantiations of an entity that is not itself a physical thing. It is not fully realised in any of these instantiations: precisely because it can be realised in the others. It is not the same as any of its instantiations. Rather, it is essentially equal to all of them:

- When considered in lower level terms, the real nature of a program is that it is an equivalence class of such representations (Sect. 2.5).
- When considered in higher level terms, it is an abstract entity obeying rigidly prescribed syntactic laws, and through a combination of bottom-up and top-down causation, it is causally effective at its own level.

It is not equal to any particular physical manifestation, e.g., on a CD disk or as electronic states in a computer. These are just vehicles whereby it is instantiated.

2.7.1.2 The Causal Effectiveness of Computer Programs

Given the physical computer, a loaded program, and input data, the output is uniquely determined:

$$(\text{physical structure, program, data}) \implies \text{output} . \quad (2.6)$$

The first two will be fixed and unchanging in a given run (with the same high level software loaded) and can be taken for granted then. So, within this context, the given constraints imply

$$(\text{data}) \underset{\text{program}}{\implies} \text{output} , \quad (2.7)$$

showing that abstract information is causally effective in the given context of a specific program, which determines in a top-down way the family of results obtained from arbitrary data. But as we have seen, the program is an abstract entity. According to Abelson and Sussman [1, p. 1]:

Computational processes are abstract beings that inhabit a computer. As they evolve, processes manipulate other abstract things called *data*. The evolution process is directed by a pattern of rules called a *program*. People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells.

That gets it just right. Abstract entities produce concrete results. They are causally effective through the computer hardware. The ultimate reason this is so is because they were designed to do so: they are an example of the causal efficacy of the human mind. Consequently:

Causal Effectiveness. Computer programs are not physical entities, but are nevertheless causally effective in numerous ways.

For example they can do engineering and calculations that result in specific physical structures such as aircraft and automobiles coming into existence. Furthermore, they facilitate economic interactions such as shopping and banking, social interaction through internet applications such as email and facebook, and education through the internet in conjunction with search engines such as Google and encyclopedias such as Wikipedia. They make a real difference in the real world. A final note for the philosophically cautious:

Existence. Because computer programs are causally effective, they clearly exist.

Here I use as a criterion that whatever is causally effective in the physical world must certainly exist (Sect. 1.3.5). If this is not true, we will have to face existence of uncaused entities or events in the physical universe.

2.7.2 Computer Programs Embody Abstract Logic, and Act Top-Down

This is possible because logical entities can cause physical effects, enabled by the interaction of bottom-up emergence and top-down causation. In particular, this happens in the interaction between the logical and physical systems. These systems are emergent systems based on the underlying physics, but then acquiring an abstract character at the higher levels.

2.7.2.1 The Implementation Hierarchy: Logical Levels and Descriptions

A series of interlocked computer programs, each representing the same logical structure, power the virtual machines at each level in the implementation hierarchy (Table 2.2). They are what give the system its dynamics. The downward link is via compilers and interpreters (Sect. 2.3.4). The upward link is via implementation, in essence according to Turing's prescription of reading a tape and performing the next logical operation specified thereon (Sect. 2.1).

The physical system is designed to embody logical relations, which are coded in a hierarchical manner through the interaction between system hardware and software. There are different layers in the description of computers and, in particular, the following⁶:

1. Digital circuits that can be directly implemented using certain physical devices.
2. Register machines that describe computation on a higher level of abstraction (in terms of very simple arithmetic operations).
3. Object-oriented programming languages that offer very abstract ways to describe data structures and operations on them.

⁶I am indebted to Vasco Brattke (private communication) for the following comments.

One of the questions seems to be: how is it possible to implement on one relatively primitive level a layer that seems to offer a much higher degree of abstraction?

The emergence from layer 1 to 2 above happens on the level of microcode, which is implemented in digital circuits and offers the first layer of programming. Microcode operations are very simple and they are actually used to implement assembler languages that offer pretty much the same type of instructions as register machines. On the level of microcode, one still reasons in terms of digital circuits and very elementary operations that transfer content from one position in the memory to another.

On the level of assembler languages one no longer has to think in terms of digital circuits, but the reasoning happens on the higher level of registers and certain arithmetic and logical operations. On this level one can actually implement abstract object-oriented programming languages such as Java (although in practice there are several intermediate layers, such as the operating system). In particular, all such things as indirect addressing, pointers, etc., can be implemented easily on the level of register machines.

In fact, as shown in [14], all these ‘programming languages’, register machines, recursive functions, Java programs, and so on, satisfy the so-called *SMN properties* (*Kleene’s translation theorem*) and *UTM properties* (*Turing’s universal function theorem*). Hence, it follows from the equivalence theorem of Rogers that each of them can be simulated in any of the others [14]. The level of description and abstraction is very different, but the power of expressiveness is essentially the same. Already at level 1 in Table 2.5, the zeros and ones are conceptual representations of physical states. The actual physical state is a charge or current [47]. It is conceptually referred to by binary notation: an abstraction that is the effective language of the logic that is built into the gates by their properties and connectivity in logical circuits.

Given this structure, the hierarchy of languages can be constructed, with compilers and interpreters [3, 6] acting top-down to link the levels. But they are just computer programs. Abelson and Sussman [1, p. 360] state the following:

Metalinguistic abstraction—establishing new languages—plays an important role in all branches of engineering design. It is particularly important to computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing evaluators. An evaluator (or interpreter) for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression. It is no exaggeration that this is the most fundamental idea in programming: the evaluator, which determines the meaning of expressions in a programming language, is just another program.

This enables the emergence of higher level entities such as the higher level systems programs and application programs, both realised when the low level systems programs are run. They subsequently exert top-down effects on lower level dynamics (Sect. 2.4). Universal computation is then possible, able to model arbitrarily complex systems.

2.7.2.2 The Logical Hierarchy

To enable high level computation additionally requires modular hierarchical structuring of a logical hierarchy (Table 2.4) at each level of the implementation hierarchy, enabling abstraction, information-hiding, and so on (Sect. 2.2.2). This structure enables contextual information processing. James McClelland describes it thus [46]:

Interactive models of language processing assume that information flows both bottom-up and top-down, so that the representations formed at each level may be influenced by higher as well as lower levels. I describe a framework called the interactive activation framework that embeds this key assumption among others, including the assumption that influences from different sources are combined non-linearly. This non-linearity means information that may be decisive under some circumstances has little or no effect under other conditions. [...] feedback from higher levels is computationally desirable [because] it allows lower levels to be tuned by contextual factors so that they can supply more accurate information to higher levels.

The 5 different types of top-down causation (Sect. 2.4) can be implemented and enable complex behaviour to emerge on the basis of purely algorithmic operations at the bottom.

2.7.2.3 Symbolic Logic Independent of the Underlying Physics

It is clear from Turing's work (Sect. 2.1) that what one can do symbolically via digital computers is not in any way restricted or constrained by the lower level physical implementation [14]. It is determined by the logic of the higher level possibility space (the effective laws of logic, mathematics, and semiotic representation), not by the underlying laws of physics that enable the computer to function.

2.7.3 Room at the Bottom

How is there room at the bottom for top-down action in a mechanistic system, where the low level operations are completely deterministic?⁷ The main way higher level structures exert an effect on lower levels is by setting various constraints on their functioning:

- The physical structuring of the computer (hardware) embodies patterns of connection that constrain what happens at gate level.
- The loaded high level software establishes further constraints on the logical structure of the lower level interactions.

⁷I only consider classical computers here, where quantum uncertainty in the underlying physics has no effect on microcomputer operations because they have been carefully designed so that this will be the case. Quantum computing raises many further possibilities I do not engage with in this text.

- Finally, the data establishes sufficient further constraints on the lower level interactions to give a unique output.

This works out in the following ways (discussed further in Sect. 5.3).

2.7.3.1 Context

Firstly, the context determines what algorithmic operations take place. The *physical context* of computer structure does not alter the lower level physics: it constrains its actions. Paradoxically, constraint creates the possibility of complexity. For example, the wiring in a computer means that a specific gate G_1 is connected only to further gates G_2 and G_3 and not to any other gates in the system, and this is what enables these three gates to produce a specific logical operation, such as AND-OR-INVERT [47]. This would not be possible if inputs from other randomly selected gates were also connected. More generally, motifs occur in complex systems and shape their behaviour by constraining interactions [2].

The *logical context* of loaded programs also constrains what happens. Gate operations at the bottom are individually identical, whether a music program, a spreadsheet, a word processor, or an image-processing program is running. The specific sequence of low level operations that takes place, and the consequent high level output, is completely different depending on the higher level context of what program is running and what data are entered.

2.7.3.2 Environment

Secondly, part of the context is the environment, which lies outside the control of the algorithmic system and exerts a causal influence on operations. In many computer applications, new data comes in during a run that was not present at the start: so the computer is not a closed system, it is influenced by the environment—a top-down effect. This happens, for example, in continually updated weather forecasting systems, online stock control systems, ATM operations, and feedback control systems.

2.7.3.3 Randomness and Adaptive Selection

Thirdly, processes of adaptive selection allow learning to take place, with new information being garnered by selection processes whereby masses of irrelevant information are discarded as irrelevant. This is non-deterministic, and hence not uniquely implied by the initial data, because the variation processes include random elements (Sect. 5.6.6). It is top-down because the outcome depends on the choice of selection criteria at higher levels in the hierarchy of causation. It may also happen in adaptive selection processes where non-algorithmic higher level criteria are used on the fly during the selection process. This occurs, for example, in the use of spreadsheets, and

all those computer-aided design processes in which the operator chooses between options.

2.7.3.4 Mutable Lower Level Elements

Fourthly and crucially, the behaviour of lower level elements is not generally immutable, but depends on context: they are adapted to their role in the hierarchy (see Sect. 5.4). Put briefly:

Contextually Determined Nature. The nature of the lower level entities—the way they respond to events—is often determined by context.

In digital computers this occurs through the late time binding that enables polymorphism in object-oriented systems [57, pp.506–531]. More generically, parameters are passed down from the higher level to set or alter the data-handling method used by modules at the lower level, thereby determining the specific outcomes. The lower class functions can in this way underlie many different higher level functions, through the setting of parameters that control function at the lower level.

2.7.3.5 The Enabling Role of Physics

One cannot derive algorithmic logic from physics: e.g., one cannot derive Quicksort either from the physical operation of electromagnetic interactions, or from the logical form of Maxwell’s equations. Yet it is algorithmic logic that drives what happens at the higher levels in a computer, and hence at the lower levels.

The underlying physics enables this to happen: it dances to the tune of this abstract logic, which gets embodied in particular patterns of energy states at the micro level. They are the outcome of the logic, not its cause. The logic of the algorithms derives from the nature of what is possible in logical terms.

2.7.4 Predictable Outcome?

Computers are the epitome of algorithmic operations: is the outcome predictable? There are three ways in which the outcome may not be implied by the initial data:

1. It is not predictable because of the complexity.
2. It can have new input: data fed in during the runtime (open systems).
3. It can have a random element inserted (by a random generator or clock time or radioactive decay).

The first is non trivial, as remarked by Turing[aut] [66]:

The view that machines cannot give rise to surprises is due, I believe, to a fallacy to which philosophers and mathematicians are particularly subject. This is the assumption that as

soon as a fact is presented to a mind, all consequences of that fact spring into the mind simultaneously with it. It is a very useful assumption under many circumstances, but one too easily forgets that it is false. A natural consequence of doing so is that one then assumes that there is no virtue in the mere working out of consequences from data and general principles.

Indeed, if the outcome were predictable, we would not need the computer!

The second case is logically obvious, but operationally important: the cases of stock control, weather forecasting, and aircraft automatic pilots are examples.

As regards the third, unpredictable effects occur despite algorithmic operation in the case of adaptive selection, based on random lower level processes plus higher level selection effects. This results in accumulation of unpredictable information, and build-up of effective structures adapted to higher level function and environment, not uniquely determined by the initial data. Genetic algorithms and neural nets are examples. They can learn only because they get input from their environment in their training phase, enabling them to use high order selection criteria in the context of this specific environment—a form of top-down action. Then the outcome is not determined, even though the process is.

To Be Done. There is an interesting issue that arises here: such programs need a source of randomness so that the outcome is not predictable, allowing genuine learning. One can use a pseudo-random number generator, or a genuine random number generator (see the discussion in TD1 above). Both generate outcomes not implicit in the initial data, but the first is a disguised algorithmic process, while the second is not: it is truly non-deterministic. The issue is whether this makes a genuine difference to the outcome: does it really matter which choice is made? The answer is not clear.

2.7.5 *Possibility Spaces and Their Causal Effects*

What can be done by computers is characterized by a possibility space: the space of all possible computations Ω_c . This in turn is based on the set of all possible algorithms Ω_a , which includes the set of possible computer programs $\Omega_a(\text{prog})$.

2.7.5.1 Possible Algorithms

What is possible algorithmically is based on the space of logically possible algorithms Ω_a . This can be thought of as an eternal unchanging space of what is and what is not logically possible. We *discover* these possibilities, that is, we work out that they are indeed possible and valid first by inspiration or invention (imagining the possibilities), then by working out the details by logical argumentation (development), and then by checking that they are indeed valid (verification), again by logical argumentation).

The same algorithms are valid anywhere in the universe: near Alpha Centauri and in the Andromeda galaxy, and at any time. They were valid before humans

existed and will remain valid after we are long gone. For example, there are various possible ways to sort a list: shellsort, heapsort, mergesort, bubble sort, quicksort, library sort, and so on [42, 68]. These have been discovered by human beings over the course of history, and indeed some were known long before computers existed. The corresponding subset $\Omega_a(\text{sort})$ of Ω_a is finite (a typical list of sort algorithms will mention about 20 possibilities), as is each algorithm itself (an infinite algorithm would be of no use whatever, as discussed in Sect. 2.6.1).

The space Ω_a is hierarchically structured: more complex algorithms such as the Google search algorithm and pattern recognition algorithms [45] build on combinations of simpler ones such as quicksort. Although this logical space is progressively explored by the human mind as we discover more and more algorithms, it is independent of the mind: the logical possibility and validity of those algorithms is true independently of what we think. Like the mathematics possibility space Ω_m , the space Ω_a embodies eternal truths independent of place and time and culture, and so can be thought of as an abstract Platonic space, as is argued in the case of Ω_m by Penrose [54] and Connes [17]. In summary:

The Space of Algorithmic Possibilities Ω_a . This is a hierarchically structured abstract Platonic space. We explore it through logical analysis by the action of the mind [19]. Instances of algorithms existing in Ω_a are causally effective when we implement them in computer programs [42, 45, 68].

This space is not implied by physics or physical laws, but by logic. Our understanding of this space cannot be tested by physics laboratory experiments (although these may possibly give hints as to how some algorithms operate). This understanding can, however, be tested by running computer programs embodying specific algorithms we have discovered and developed. They either work to give the desired results, or they don't!

2.7.5.2 Possible Computations: Limits of Computability and Applicability

Because computer programs are in essence just high level algorithms made by combining lower level algorithms in a structured way so as to produce a complete calculation, the space of possible computer programs is in essence a subspace $\Omega_a(\text{prog})$ of Ω_a . But this is not the same as the space of possible computations Ω_c . Various issues intervene.

What can be computed and what cannot? There are four aspects here:

1. What kinds of problems are algorithmically expressible?
2. What algorithmic problems can be computed in principle by a physical device?
3. What is algorithmically computable by programs in a finite time?
4. What is computable in a realistic time?

These are deep issues, which I will only touch upon in the briefest of ways.

1. What kinds of problems are algorithmically expressible? How much of what humans understand can be algorithmically encoded? The brain does not naturally work in an algorithmic way, although it can be trained to do so. It operates by pattern recognition, enabled by the overall pattern of neural connections in the cortex [33], the connection weights in these neural networks (Churchland [19]), and synchronized patterns of oscillations between them (Buzsáki [16]).

These are not at all like the algorithmic operations of a digital computer, so it is not obvious that all that they can do can be represented by algorithmic processes (Penrose [52, 53]), unless those processes mimic the adaptive properties of neural networks [11], that is, they don't model the pattern of understanding attained, rather they model the process by which it is attained.

2. What kinds of algorithmic problems can be computed in principle by a physical device? This is the subject of the Church–Turing thesis, stated by Brattke [14] as follows:

Church–Turing Thesis (1936). A function $f : \subseteq N^k \rightarrow N$ is computable in the formal sense if and only if it can be computed by some physical device.

This form of the thesis is not a mathematical statement since it relates the mathematical concept of computable functions to the question of what it means to compute something with a physical device. Copeland states it this way [20]:

Thesis M. Whatever can be calculated by a machine (working on finite data in accordance with a finite program of instructions) is Turing-machine-computable. Thesis M itself admits of two interpretations, according to whether the phrase “can be generated by a machine” is taken in the narrow, this-worldly, sense of “can be generated by a machine that conforms to the physical laws (if not to the resource constraints) of the actual world”, or in a wide sense that abstracts from the issue of whether or not the notional machine in question could exist in the actual world. Under the latter interpretation, thesis M is false. It is straightforward to describe notional machines, or ‘hypercomputers’ that generate functions not Turing-machine-computable. It is an open empirical question whether or not the narrow this-worldly version of thesis M is true.

The latter is the case of physical interest.

3. What is algorithmically computable by programs in a finite time? This is the issue of the *halting problem* [21]: given a valid program, will the computation come to an end in a finite time? The algorithmic structure of the program may be logically correct, but the computation may never conclude, and no algorithmic computation can determine whether this will happen or not. Chaitin states this as follows [18]:

Turing's train of thought now takes a very dramatic turn. What, he asks, is impossible for such a machine? What can't it do? And he immediately finds a problem that no Turing machine can solve: the halting problem. This is the problem of deciding in advance whether a Turing machine (or a computer program) will eventually find its desired solution and halt.

A solution to the halting problem would determine the space of possible computations Ω_c as a subset of $\Omega_a(\text{prog})$, but this is unsolvable by any Turing Machine.

4. What is computable in a realistic time? This is the whole subject of computational complexity and computation times. Issues occurring include time functions,

complexity measures, and complexity classes [14, Sect. 3.6]. The necessary amount of auxiliary storage, stability, and effects on indexing keys are also important when comparing algorithms. Together these determine a subspace Ω_c (realisable) of Ω_c representing those possible algorithms that can be effectively implemented. This is a very context-dependent concept: as computer memory size and speed increase, what was previously impractical becomes possible. This is of great practical importance.

2.7.5.3 The Causal Effectiveness of Platonic Possibility Spaces

Overall, the key issue is the causal effectiveness of algorithms. This is what enables computer applications in engineering, science, and commerce, which cause real changes in the physical world. So where do they come from? The chain of causation is shown in Table 2.17. As explained above, algorithms ultimately originate in the Platonic space of logically possible algorithms Ω_a . Thus the conclusion is as follows:

Causal Effectiveness of Platonic Spaces: The abstract possibility spaces Ω_a and Ω_c are the ultimate source of the causal powers of digital computers in the physical world.

Three-dimensional printers are able to create physical objects because the algorithms that enable this are valid algorithms, and that fact is a consequence of the nature of the Platonic space Ω_a .

Their Existence. The claim that all these spaces exist, i.e., that they are ontologically real, rests upon a philosophical analysis of what kinds of things must be recognised as existing. The view taken here (see [30] and Sect. 1.3.5) is that we must recognise the existence of any kind of entity that demonstrably has a causal influence on physical systems.

The possibility spaces discussed here are certainly causally effective, even though non-physical, so they must be realised as existing. They are the ultimate source of computational power.

Table 2.17 The origin of algorithms and programs in the abstract possibility spaces Ω_a (possible algorithms) and Ω_c (possible computations). These lead to real world effects such as 3D printing of physical objects

Level 4	Possibility space Ω_a	Possible algorithms
	↓	
Level 3	Possibility space Ω_c	Possible computations
	↓	
Level 2	Written programs p_i	Selected algorithms a_j
	↓	
Level 1	Computer run	Selected program and data
	↓	
Level 0	Output data/actions	⇒ Real world effects

2.7.6 *Top-Down Action from the Mind*

Computer programs based in the possibility spaces Ω_a and Ω_c are not physical entities, but are nevertheless causally effective in numerous ways [45]. The final puzzle is this: how are these possibility spaces causally effective in this way? How do they influence what gets realised in computers?

The answer is through the human mind, which explores these spaces by logical reasoning. This is enabled by the ability of neural networks to learn about such abstract spaces through processes of pattern recognition based on the operation of neural networks in our brains, as explained clearly by Churchland [19]. Hence, I emphasize:

Causal Effectiveness of Platonic Spaces. It is through adaptive selection processes in the mind, enabled by the neural circuits in the brain, that the possibility spaces are understood and hence causally effective.

This enables not only the existence of operational programs and algorithms, but also computers themselves: the physical entities that make this all happen. They ultimately originate, not only from our exploration of possible algorithms Ω_a , but also from our explorations of the physical possibility space Ω_{ph} restricting what is physically possible due to the nature of physical interactions (described by the laws of physics). Their development embodies the combined experience of numerous workers in aspects ranging from basic concepts to solid state physics to system design to effective algorithms to high level design patterns. This leads to the extraordinary ability of digital systems to represent language, pictures, sound, mathematical relationships, and indeed all human knowledge. Overall, this is the effect of intelligent top-down causation from the human mind to physical systems (the computer itself) and abstract systems (the set of programs that make a computer work).

At a higher level, the existence of computers is an outcome of the human drive for meaning and purpose: it is an expression of the possibility space of meanings, the higher levels whereby we guide what actions take place. This will be discussed in Chap. 8.

2.7.7 *Genuine Emergence*

Although they are the ultimate in algorithmic causation, as characterized so precisely by the concept of Turing machines, digital computers embody and demonstrate the causal efficacy of various kinds of non-physical entities—algorithms, programs, data—which enable truly complex behaviour to emerge from simple constituents.

It is noteworthy here that one is able to regard level 0 in Table 2.1 as the bottommost level, the level ‘where the work is really done’, even though this is not in fact the case if one takes a strict reductionist viewpoint: that level emerges from lower physical levels, which are *really* where the work is done!

Why is it then legitimate to regard the emergent level 0 as real, as is taken for granted by all computer scientists and engineers? The answer is that this level does indeed do real work, as do *all* the levels in Table 2.1:

Genuine Emergence. Each of the levels in Table 2.1 is a causally effective emergent level of structure. They are all equally real.

Just as in the case of neurons and the mind, and indeed biology as a whole [50], this is the only approach that makes sense. And it is valid because of the reality of top-down causation in the hierarchies, as discussed in this chapter. I revisit this issue in Sect. 8.1.

The operations at each level in both the logical and implementation hierarchies are realizations of possibilities occurring in abstract Platonic spaces such as Ω_a (Sect. 2.7.5), and these are the ultimate source of the possibility of computation. Their implementation in physical terms is possible because the human mind is able to comprehend the nature of these possibility spaces [19].

References

1. H. Abelson, G.J. Sussman, J. Sussman, *Structure and Interpretation of Computer Programs* (MIT Press, Cambridge, 1996)
2. U. Alon, *An Introduction to Systems Biology: Design Principles of Biological Circuits* (Chapman and Hall/CRC, London, 2007)
3. A.W. Appel, *Modern Compiler Implementation in Java* (Cambridge University Press, Cambridge, 2002)
4. W. Ross Ashby, *An Introduction to Cybernetics* (Chapman and Hall, London, 1957). <http://pcp.lanl.gov/books/IntroCyb.pdf>
5. G. Auletta, G.F.R. Ellis, L. Jaeger, Top-down causation: From a philosophical problem to a scientific research program. *J. R. Soc. Interface* **5**, 1159–1172 (2008). [arXiv:0710.4235](https://arxiv.org/abs/0710.4235)
6. A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, and Tools Paperback* (Pearson, 2013)
7. S. Beer, *Brain of the Firm* (Wiley, Chichester, 1981)
8. C.H. Bennett, Notes on Landauer's principle, reversible computation and Maxwell's demon. *Stud. History Philos. Modern Phys.* **34**, 501–510 (2003)
9. S. Bennett, S. McRobb, R. Farmer, *Object-Oriented Systems Analysis and Design* (McGraw Hill, Maidenhead, 2010)
10. A. Béruit, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, E. Lutz, Experimental verification of Landauer's principle linking information and thermodynamics. *Nature* **483**, 187–190 (2012)
11. C.M. Bishop, *Neural Networks for Pattern Recognition* (Oxford University Press, Oxford, 1999)
12. G. Booch, *Object-Oriented Analysis and Design with Applications* (Addison Wesley, New York, 1994)
13. G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide* (Addison Wesley, New York, 1998)
14. V. Brattka, *Computability Theory* (University of Cape Town Notes, 2011)
15. V. Brilhante, Computer modelling hierarchy: the model reflects the hierarchy of the system being modelled. *J. Braz. Comp. Soc.* **11**(2), Campinas (2005)
16. G. Buzsáki, *Rhythms of the Brain* (Oxford University Press, Oxford, 2006)

17. J.-P. Changeux, A. Connes, *Conversations on Mind, Matter, and Mathematics* (Princeton University Press, Princeton, 1998)
18. G.J. Chaitin, Computers, paradoxes and the foundations of mathematics. *Am. Sci.* **90**, 164–171 (2002)
19. P. Churchland, *Plato's camera: how the physical brain captures a landscape of Abstract Universals* (Cambridge) (The MIT Press, Cambridge, 2012)
20. B.J. Copeland, The Church–Turing Thesis. *The Stanford Encyclopedia of Philosophy*, (Fall 2008 edition), ed. by E.N. Zalta (2002). <http://plato.stanford.edu/archives/fall2008/entries/church-turing/>
21. J. Copeland, *The Essential Turing* (Oxford University Press, Oxford, 2004)
22. T. Deacon, *The Symbolic Species: The Co-Evolution of Language and the Human Brain* (Penguin, London, 1997)
23. K.A. De Jong, *Evolutionary Computation: A Unified Approach* (MIT Press, Cambridge, 2006)
24. G. Dyson, *Darwin Among the Machines* (Penguin, London, 1997)
25. G.F.R. Ellis, True complexity and its associated ontology, in *Science and Ultimate Reality: Quantum Theory, Cosmology and Complexity*, ed. by J.D. Barrow, P.C.W. Davies, C.L. Harper (Cambridge University Press, Cambridge, 2004), pp. 607–636
26. G.F.R. Ellis, On the nature of causation in complex systems. *Trans. R. Soc. S. Africa* **63**, 69–84 (2008)
27. G.F.R. Ellis, Top-down causation and emergence: some comments on mechanisms. *J. R. Soc. Interface Focus* **2**, 126–140 (2012)
28. G.F.R. Ellis, D. Noble, T. O'Connor (eds.), Top-down causation: An integrating theme within and across the sciences? *R. Interface Focus Spec. Issue* **2**, 1–140 (2012)
29. K. Fatahalian, T.J. Knight, M. Houston, M. Erez, D.R. Horn, L. Leem, J.Y. Park, M. Ren, A. Aiken, W.J. Dally, P. Hanrahan, Sequoia: Programming the memory hierarchy, in *SC 2006 Conference, Proceedings of the ACM/IEEE* (2006)
30. R.L. Flood, E.R. Carson, *Dealing with Complexity: An Introduction to the Theory and Application of Systems Science* (Plenum Press, London, 1990)
31. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software* (Addison Wesley, New York, 1995)
32. P. Gray, *Psychology* (Worth Publishers, New York, 2011)
33. J. Hawkins, *On Intelligence* (Holt Paperbacks, New York, 2004)
34. D. Hofstadter, *Godel, Escher, Bach: An Eternal Golden Braid* (Penguin, London, 1980)
35. D. Hilbert, On the infinite, in *Philosophy of Mathematics*, ed. by P. Benacerraf, H. Putnam (Prentice Hall, Englewood Cliff, 1964), p. 134
36. J.H. Holland, *Adaptation in Natural and Artificial Systems* (MIT Press, Cambridge, 1992)
37. B. Jacobs, S.W. Ng, D.T. Wang, *Memory Systems: Cache, DRAM, Disk* (Elsevier, Burlington, 2008)
38. N.L. Kamorova, M.A. Nowak, Language, learning, and evolution, in *Language Evolution*, ed. by M.H. Christensen, S. Kirby (Oxford University Press, Oxford, 2005), pp. 317–337
39. R.M. Keller, *Computer Science: Abstraction to Implementation*. <http://www.cs.hmc.edu/~keller/cs60book/>
40. J.F. Kurose, K.W. Ross, *Computer Networking: A Top-Down approach* (Addison-Wesley, New York, 2012)
41. J. Ladyman, S. Presnell, A.J. Short, B. Groisman, The connection between logical and thermodynamic irreversibility (2006). <http://philsci-archive.pitt.edu/id/eprint/2689>
42. R. Lafore, *Data Structures and Algorithms in Java* (SAMS, Indianapolis, 2002)
43. R. Landauer, Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.* **5**, 183–191 (1961)
44. S. Lloyd, Computational capacity of the universe (2001). [arXiv:quant-ph/0110141](https://arxiv.org/abs/quant-ph/0110141)
45. J. MacCormack, *Nine Algorithms that Changed the Future: The Ingenious Ideas that Drive Today's Computers* (Princeton University Press, Princeton, 2012)
46. J.L. McClelland, The case for interactionism in language processing. Technical Report AIP-2 (Department of Psychology, Carnegie-Mellon University Pittsburgh, PA 15213 USA, 1987)

47. M.M. Mano, C.R. Kime, *Logic and Computer Design Fundamentals* (Pearson/Prentice Hall, 2008)
48. J. McCarthy, Artificial intelligence, logic and formalizing common sense (1990). <http://www-formal.stanford.edu/jmc/>
49. J.H. Miller, S.E. Page, *Complex Adaptive Systems: An Introduction to Computational Models of Social Life* (Princeton University Press, Princeton, 2007)
50. D. Noble, A theory of biological relativity: no privileged level of causation. *Interface Focus* **2**, 55–64 (2012)
51. Object Management Group (OMG), *OMG Unified Modeling Language (OMG UML) Superstructure Version 2.2*. <http://www.omg.org/spec/UML/2.4.1/>
52. R. Penrose, *The Emperor's New Mind: Concerning Computers, Minds and the Laws of Physics* (Oxford University Press, New York, 1989)
53. R. Penrose, *Shadows of the Mind: A Search for the Missing Science of Consciousness* (Oxford University Press, Oxford, 1994)
54. R. Penrose, *The Large, the Small and the Human Mind* (Cambridge University Press, Cambridge, 1997)
55. J. Porway, Q.C. Wang, S.C. Zhu, A hierarchical and contextual model for aerial image parsing. http://vcla.stat.ucla.edu/Aerial_Image_Parsing/index.html
56. S Russell and P Norvig (2009) *Artificial Intelligence: A Modern Approach* (Prentice Hall)
57. W. Savitch, *Absolute Java* (Pearson, Boston, 2010)
58. S.C. Shapiro, Artificial intelligence, in *Encyclopaedia of Artificial Intelligence*, ed. by S.C. Shapiro (Wiley, New York, 1992), pp. 54–57
59. J.R. Searle, Is the brain a digital computer? https://mywebspace.wisc.edu/lshapiro/web/Phil554_files/SEARLE-BDC.HTM
60. A. Silberschatz, P.B. Galvin, G. Gagne, *Operating System Concepts* (Wiley, New York, 2010)
61. H.A. Simon, *The Sciences of the Artificial* (MIT Press, Cambridge, 1992)
62. A.S. Tanenbaum, *Structured Computer Organisation* (Prentice Hall, Englewood Cliffs, 2006)
63. R.L. Trask, *Language and Linguistics: The Key Concepts* (Routledge, Abingdon, 2007)
64. A.M. Turing, On computable numbers, with an application to the Entscheidungsproblem, in *Proceedings of the London Mathematical Society*, vol. **42**, pp. 230–265 (1936) (Reprinted in J. Copeland, *The Essential Turing* (Oxford University Press, Oxford, 2004), p. 58)
65. A.M. Turing, Lecture on the automatic computing engine (1947) (Reprinted in J. Copeland, *The Essential Turing* (Oxford University Press, Oxford, 2004), p. 378)
66. A.M. Turing, Computing machinery and intelligence. *Mind* **59**, 433–460 (1950) (Reprinted in J. Copeland, *The Essential Turing* (Oxford University Press, Oxford, 2004), p. 433)
67. D.A. Watt, D.F. Brown, *Programming Language Processors in Java: Compilers and Interpreters* (Prentice Hall, Harlow, 2000)
68. M.A. Weiss, *Data Structure and Algorithm Analysis in Java* (Addison Wesley/Longman, 1999)

<http://www.springer.com/978-3-662-49807-1>

How Can Physics Underlie the Mind?
Top-Down Causation in the Human Context

Ellis, G.

2016, XXVI, 482 p. 36 illus., Hardcover

ISBN: 978-3-662-49807-1