II DESIGNING EXPRESSIVE PROCEDURES

The design of digital artifacts rests on our ability to assign symbolic meaning to electrical impulses by building up **abstraction layers**, from the **primitives** of the 00s and 11s of the machine code, applying the principles of **modularity** and **encapsulation.**

A key design goal for procedural systems is to increase human ability to think coherently about processes that have many **variables** that serve as **parameters**, modifying the way in which an action is performed. Designers should know how to represent complex, variable procedures using **flow charts**, **pseudocode,** and **state diagrams**.

Designers should recognize that the same process can be approached through more than one **algorithm** and that any fixed representation of a process, whether it is computer code or an earlier media genre such as a proverb, recipe, or automated mechanical device, reflects cultural biases and values and is subject to human error.

Designers should be conversant with the computational concepts used in **object-oriented** programming, in which objects are described separately from their **instantiation** and within more generalized **classes** from which they can **inherit** qualities and behaviors (**attributes** and **methods)**.

Designers should know how to use **discrete time steps**, **feedback loops**, **resource allocation** structures, and focused **scenarios** to design **simulations** that afford modification and **replay**, so that interactors can increase their understanding of cause and effect within complex systems.

Humanist designers should be aware of what values they are serving and what assumptions they are making about the world when creating the cause-and-effect structures of computer programs and simulations. Wherever possible, they should make these assumptions visible to the interactor.

4 Computational Strategies of Representation

The Analytical Engine has no pretensions whatever to *originate* anything. It can do whatever we *know how to order* it to perform.

—Ada Lovelace (1843)

Computation as Symbol Manipulation

Designers need to understand the ways in which meaning is inscribed and encoded on electronic circuits, and the conceptual strategies by which software engineers represent the world. Computational concepts shape the fundamental structures of the digital medium, and no member of a design team can understand the plasticity of the medium without a conceptual understanding of computation, starting with the fact that the entire system is based on a very simple mechanism: an electrical switch that can be in either the on or off position. The interior of a computer can be thought of as composed of millions of such switches, each of which can change state in a very small fraction of a second. A computer program is a set of conditional rules, an executable script, for turning these switches on and off. Computers do only what they are explicitly instructed to do, and all of their instructions have to be translated into a changing **pattern** of on/off switches. We can imagine the inside of any computer-based device as filled with tiny circuits like Christmas lights, flashing on and off in rapidly changing patterns.

The power of computation rests on a **semiotic** connection similar to the way spoken language associates meaning with arbitrary vocalizations. Just as there is no reason why the sound "Ma" should mean "Mother"—or anything at all for that matter—there is no reason why a particular electrical state should hold meaning, or why one state should be assigned a value of 0, and another state should be assigned a value of 1. In both cases, there has been a cultural consensus—we (as English speakers) all agree to assign the meaning of "Mother" to the sound "Ma" and we (as programmers) agree that we will think of physical electrical switches, like vacuum tubes or silicon transistors, as abstract entities called *binary digits* or ***bits*** that

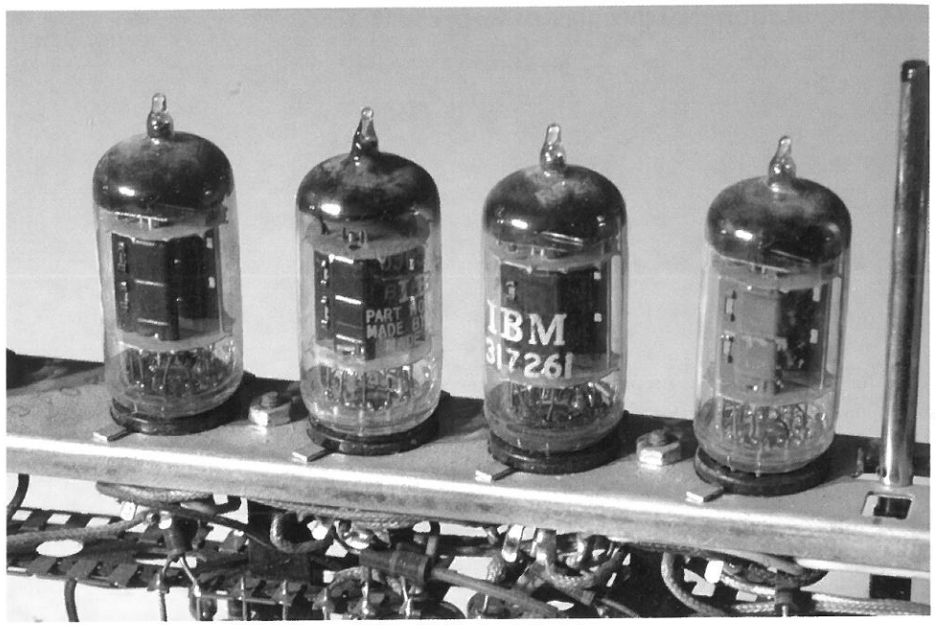

Figure 4.1
Vacuum tubes in a 1950s IBM mainframe computer, the technology that preceded the transistor, as a means of creating variations in electrical current for use in media appliances such as computers, radios, and televisions.

can be arranged in recognizable patterns, usually tracked in eight-bit groups, each of which is called a **byte**.

Everything symbolically represented within the machine—whether it is an urgent bank transfer or a thrilling video game or a mind-numbing game of solitaire—is stored as a pattern of on/off switches. A one-**megabyte** (1 Mb) file in any format is composed of eight million tiny electrical switches (bits), whether that file contains the text of a Pulitzer prize-winning book or a silly cat trick video or the initiation sequence for a nuclear weapon. The encyclopedic capacity and rapid speed of today's computers are the result of the successive miniaturization of these bits, from vacuum tubes to transistors to integrated circuits etched into the silicon (figures 4.1–4.3).

But the invention of the computer predates the invention of the electrical technologies on which it currently rests. The invention of the modern computer as a system of symbolic representation is credited to nineteenth-century mathematician Charles Babbage, who drew up plans for a calculating machine that would be programmable by removable cards, similar to the cards used by the French inventor and weaver Jacquard to produce variable and intricate textile patterns. Ada Lovelace, the daughter

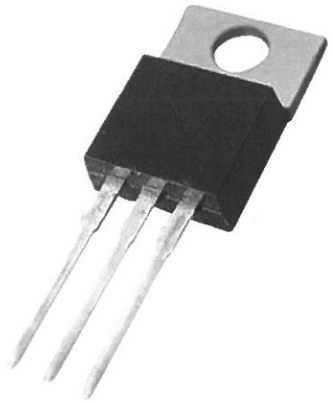

Figure 4.2
A solid-state transistor, introduced in the 1950s, could replace multiple vacuum tubes, making for increasingly portable electronic devices, and increased computational capacity. Transistors are connected in electrical circuits hardwired to execute the primitive (i.e., foundational, building-block) commands of a computer system. For example, an Adder circuit transmits current in a pattern that can be interpreted as binary addition. Starting in the mid-1960s transistors were further miniaturized into integrated circuit chips.

of the English poet, Lord Byron, designed the first such set of cards, thereby becoming the world's first computer programmer. Babbage's "Analytical Engine" was never built, but Babbage and Lovelace developed such detailed plans for it that Lovelace came to understand that such a symbolic logic machine would be more than a mere calculator and "might act upon other things besides number, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations." In particular, since she was trained in music as well as mathematics, she imagined a machine that could generate musical compositions: "Suppose, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expressions and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent" (Lovelace 1843).

At the time that Lovelace was imagining such a machine, the most common form of automated sequence was the simple music box. Music boxes work by the action of a wheel with "pins" embedded in it and turned by a camshaft. The pins strike a keyboard called a "comb," producing tones in the appropriate sequence to form a melody. The physical layout of the pins on the wheel corresponds to a sequence of notes that is executed by the motion of the wheel. What Babbage envisioned for his Analytical Engine was more like a player piano: a machine that could execute fixed sequences

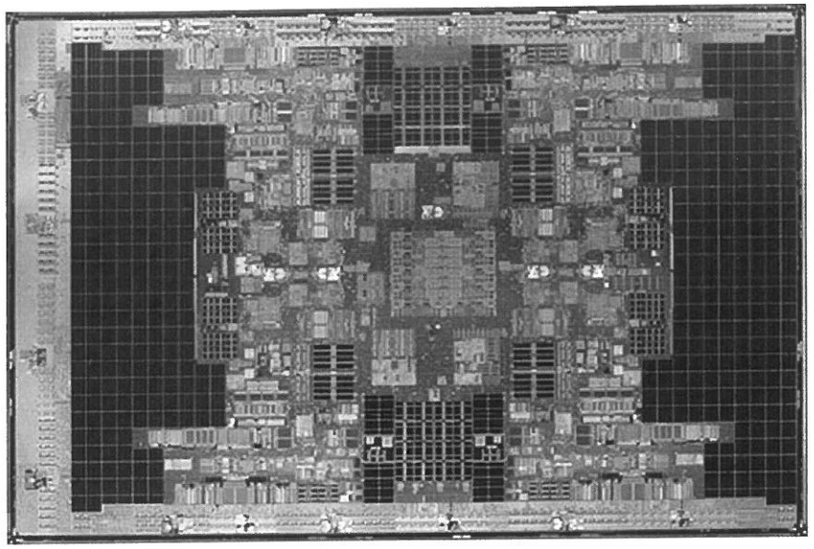

Figure 4.3
The Intel Itanium chip with two billion transistors, each only sixty-five nanometers wide, was announced in 2008, two years after the release of a one-billion-transistor chip, reflecting the company's continued aspiration to fulfill Moore's Law of 1965 which predicts that integrated circuits will double in capacity every two years. In 2010 Intel announced work on a new chip that would have transistors only two nanometers wide.

expressed on swappable cards. But Lovelace realized that such a machine could be programmed not with a particular tune, but with the rules of musical composition in symbolic representation. It could then apply those rules to generate new combinations. Lovelace was therefore the first person to understand the power of symbolic computation. *The conceptual leap from invariant, unisequential, hardwired mechanical processes like music boxes to dynamic, rule-based, multisequential, symbolically coded sequences is what makes the computer a medium of representation.*

It was not until the twentieth century that we had an inscription and transmission technology in the form of electrical current interpreted as bits that could hold such abstract symbols and execute the rules they contained. The state of any set of bits is unambiguous—they are each either on or off, 0 or 1—but the meaning of the bits can change depending on the type of **data** structure of which they are part. The same bit configuration can be interpreted as numbers, letters, pixels on a screen, or as any other kind of data, display, or abstract symbol. For example, consider the following byte:

```
0100 1010
```


Table 4.1
Binary, Hexadecimal, and Decimal (Conventional) Symbolic Representation

Binary	Hexadecimal	Decimal
0000	0	00
0001	1	01
0010	2	02
0011	3	03
0100	4	04
0101	5	05
0110	6	06
0111	7	07
1000	8	08
1001	9	09
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Hexadecimals represent four bits in a single digit. In binary form a byte takes up eight separate bits; the same information can be summarized for humans in two hexadecimal digits.

We can interpret it as a conventional (i.e., base 10) number, one million one thousand and ten: 1,001,010. Or we can read it (as is customary for bits) as a binary (base 2) number, equivalent in quantity to the **hexadecimal** number 4A, or the decimal number 74 (table 4.1). Because it is a number, we can perform arithmetic on it. If we added 1 to it (`0000 0001`), for example we would get

```
0100 1011
```

which is equivalent in quantity to 75 in the decimal system.

But even though it is made up of 0s and 1s, the bit pattern 0100 1010 need not be read as a number. It can also be interpreted as the letter J, within the **ASCII** (American Standard Code for Information Interchange) system (table 4.2).

Or if we were to see the eight bits as part of an array representing one pixel per bit within a blue and red image, with 0 for blue and 1 for red, then 0100 1010 would represent the image shown in figure 4.4.

The bits could also represent true/false answers to a set of eight questions. In that case 0100 1010 might represent a complex set of knowledge such as:

```
Were you born in the South? NO (F)
Were you born in the Northeast? YES (T)
Were you born in Massachusetts? NO (F)
Were you born in Connecticut? NO (F)
Were you born in New York? YES (T)
Were you born upstate New York? NO (F)
Were you born in Queens? YES (T)
Were you born in the Bronx? NO (F)
```

Or `0100 1010` could represent items in a player's inventory within an adventure game:

```
Sword–
Dagger–TAKEN
Bread–
Book of spells–
Poison–TAKEN
Map–
Invisibility Cape–TAKEN
Candy–
```

Bits can also represent instructions to the computer. In fact, all computer instructions, no matter what programming language they originate in (such as Java, C++, PHP) end up being translated into machine code, or patterns of bits, usually in the form of one byte per instruction. For example, the instruction ADD might be written as 1111 0000. Such basic instructions are the **primitives,** or smallest building blocks, of all computer code.

The literal mindedness of computers is discouraging to novice programmers, but it is helpful to remember that the arbitrariness of the machinery, its ignorance of the world, is an **affordance** as well as a **constraint**: it allows us to employ these readily available, easily changed bits to mean anything we want them to. *As designers approaching the computer not merely as a tool for number crunching but as an expressive medium capable of many kinds of symbol processing, it is our job to expand the repertoire of meaningful qualities, ideas, and experiences that can be mapped onto those very simple but infinitely reconfigurable on/off switches.*

Abstraction of Processes into Flow Charts and Pseudocode

The principle strategy of representation in computer science is **abstraction**, a practice that underlies all representational media. Whenever we describe the world in any code we focus on part of it and not all of it, abstracting the thing we want to focus on away from the booming, buzzing chaos of our total experience. When the baby says "Mama" or "Dada" she is abstracting her parents away from the totality of sensation. When we use the word "mother" to refer to the category of people rather than to any

Table 4.2
ASCII (American Standard Code for Information Interchange) Representations for English Language Capital Letters

Letter	ASCII Code
A	0100 0001
B	0100 0010
C	0100 0011
D	0100 0100
E	0100 0101
F	0100 0110
G	0100 0111
H	0100 1000
I	0100 1001
J	0100 1010
K	0100 1011
L	0100 1100
M	0100 1101
N	0100 1110
O	0100 1111
P	0101 0000
Q	0101 0001
R	0101 0010
S	0101 0011
T	0101 0100
U	0101 0101
V	0101 0110
W	0101 0111
X	0101 1000
Y	0101 1001
Z	0101 1010

The Unicode standard is more inclusive; see <http://unicode.org>.

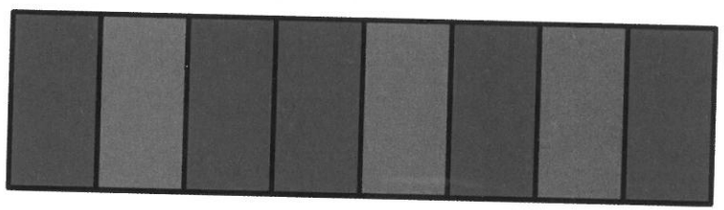

Figure 4.4
Pattern of pixels represented by the array of bits 0100 1010.

particular mother, we are abstracting in another way: generalizing from common elements in our experience to the general case that applies to all.

Computation, like language, rests on practices of abstraction. We find ways of identifying discrete items by giving them unique **labels**, addresses, descriptions; and we find ways of referring to multiple similar items with a single descriptor. Most important, since the computer is procedural we can specify a process once and then apply it iteratively to many similar items. For example, we refer to some people as "employees," a socially defined abstraction that identifies people by a single role and allows us to treat different members of this class in the same way. We are familiar with the abstract **genre** of representation called a "paycheck" based on a standardized transmission and inscription format (paper of a certain size, with fixed arrangement of fields for date, payee, amount, signature, routing number, bank account number, etc.). We can build on these socially established abstractions to tell the computer to issue "paychecks" to all "employees" following the same general rules (a procedural abstraction) for calculating and issuing each individual paycheck.

Computer code is a system for capturing similarity and variation in abstract terms. We capture variation of processes through conditional statements, and we capture variation in data by assigning parts of the data to different **fields** of a **database.** For example, our data about employees might be assigned to the general categories (fields) of first name, last name, address, hourly rate of pay, hours worked. Taken together the set of categories makes up an abstract representation of any employee. Any individual employee would be an **instance** of this abstract representation, and would have appropriate information populating each of the fields.

Programmers specify instructions in computer code, which is written so that it is unambiguous for a computer. It is very beneficial for designers to understand the syntax of programming languages, but the coding step can be separated from the conceptual work of deciding how a program should work. Computational processes can be specified in two human-readable formats that are precise but independent of the coding layer: **flow charts** and **pseudocode.** Designers should be able to create and read documents in both of these genres in order to have a common language with the other members of the design team and in order to think more clearly about the computational representation of processes. For example, the flow chart for paying an employee might look like figure 4.5.

Pseudocode captures the step-by-step logic of a process, the **algorithm** or abstract procedure, for performing it without worrying about the unforgiving syntax of any particular programming language. We can use pseudocode to describe the components of data structures like this:

```
Define Datastructures:
  timecard with name, hours, rate
  paycheck with name, pay
```


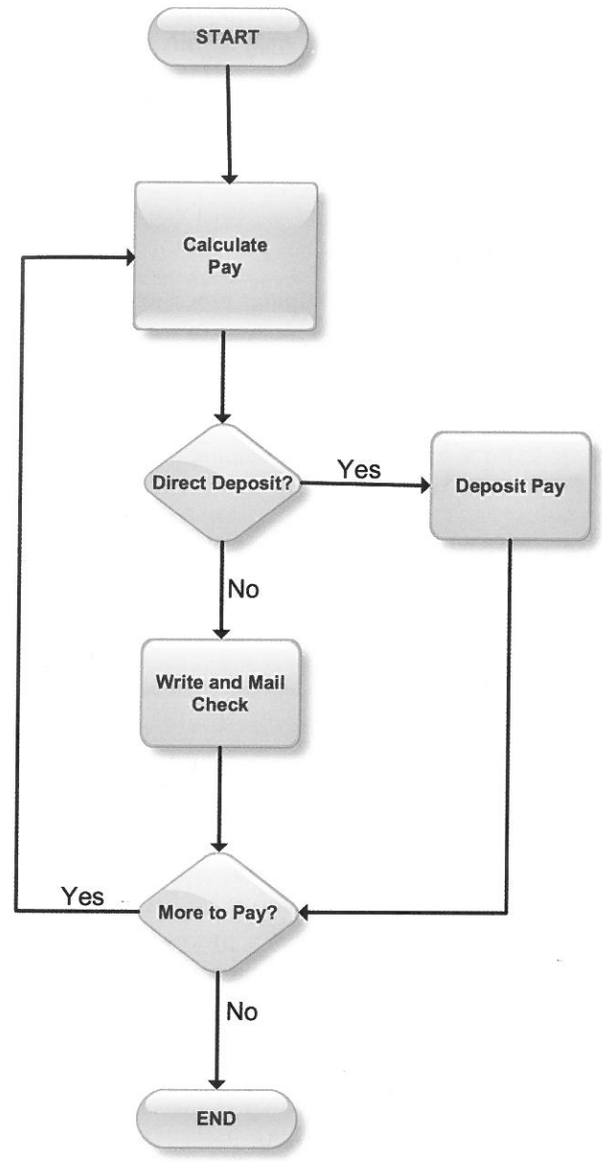

Figure 4.5
Flow chart for a program that calculates pay and either writes and mails a check or does a direct deposit. There are two decision points, shown as diamond shapes, and one **loop,** which iterates the same sequence until the program runs out of timecards. A flow chart is a useful way to display the logic of a conditional process.

We can incorporate aspects of programming syntax that are self-explanatory or commonly understood by team members. For example, a data field within a larger data object (like a name field on a timecard) is often referred to like this in programming syntax:

entity.attribute

So we can refer to fields in the timecard like this:

```
timecard.name
```

We can use pseudocode to specify instructions to the computer processor as a series of functions. In the case of the payroll program we need functions to read timecards, copy information, calculate pay, and print out paychecks:

```
READ next timecard
COPY timecard.name to paycheck.name
CALCULATE paycheck.pay = timecard.hours x timecard.rate
PRINT paycheck
```

Or if we were to write it closer to the actual code we might use the = sign without the "copy" or "calculate" command words:

```
READ next timecard
timecard.name = paycheck.name
paycheck.pay = timecard.hours x timecard.rate
PRINT paycheck
```

There are many correct ways to write out the same procedure in pseudocode. It is only important that it be self-consistent, logical, and readable by all the appropriate team members.

Let us assume that some employees have direct deposit of their pay. In that case we want to calculate the pay in the same way but we do not want to print the paycheck. Let us assume there is a separate **subroutine,** a process that handles direct deposit, called DEPOSIT. We would just have to change the last instruction like this:

```
COPY timecard.name to paycheck.name
CALCULATE paycheck.pay = timecard.hours x timecard.rate
DEPOSIT paycheck
```

To be efficient we want to combine both possibilities—PRINT and DEPOSIT— into a single program. To do that we have to add another field to our TIMECARD data structure:

timecard with name, hours, rate, directDeposit

The **variable** directDeposit can have a **value of true or false**. The pseudocode for processing either case could use the IF/THEN/ELSE structure that is a foundational convention of computer programming:

```
START
READ next timecard
COPY timecard.name to paycheck.name
CALCULATE paycheck.pay = timecard.hours x timecard.rate
IF directDeposit = true
THEN DEPOSIT paycheck
ELSE PRINT paycheck
END IF
```

Furthermore, we need an instruction that will allow us to repeat this process in a **loop** until there are no more timecards to read. To do that we can create another IF/THEN/ELSE statement, building up complexity of processing by nesting the original program within it:

```
START
READ next timecard
IF no more timecards
THEN QUIT
ELSE
  COPY timecard.name to paycheck.name
  CALCULATE paycheck.pay = timecard.hours x timecard.rate
  IF directDeposit = true
  THEN DEPOSIT paycheck
  ELSE PRINT paycheck
  END IF
END IF
LOOP BACK to "READ next timecard"
```

The ability to keep track of nested processes is important as the conditions proliferate, so we use indenting to indicate the **hierarchy**, and we end each IF/THEN/ELSE sequence with an END IF command to keep the logic clear. We can also simplify the structure by expressing the outer loop as a WHILE DO statement, which can be written in pseudocode like this:

```
WHILE any timecard is left in the stack
DO
    COPY timecard.name to paycheck.name
    CALCULATE paycheck.pay = timecard.hours x timecard.rate
    IF deposit = true
    THEN PRINT paycheck
```

```
   ELSE DEPOSIT paycheck
   END IF
END WHILE
QUIT
```

The instruction after the WHILE DO loop executes when the WHILE condition is no longer met. In this case, the program quits.

Conditional choices often have more than two possibilities, making it cumbersome to express them using IF/THEN/ELSE. Programming languages use case statements to express multiple conditions. The pseudocode for a CASE statement looks like this:

```
CASE
TODAY= Monday DO Monday_process
TODAY= Tuesday DO Tuesday_process
TODAY= Wednesday DO Wednesday_process
TODAY= Thursday DO Thursday_process
TODAY= Friday DO Friday_process
TODAY= Saturday DO Saturday_process
TODAY= Sunday DO Sunday_process
ELSE RETURN_ERROR
END CASE
```

Conditional instructions can be expressed in many ways, but the logic is the same. The computer is told to apply a test to some item, and to execute different instructions based on the results of this test. A computer program is usually made up of multiple such tests and a key part of computational abstraction is determining all the conditions that must be recognized and treated differently.

Part of the power of computational representation comes from our ability to abstract parts of a process as **variables**, as entities whose value will change during the execution of the code. For example, in the payroll example there were variables for:

- the name of the employee
- the number of hours worked
- the rate of pay
- the direct deposit indicator
- the amount of the paycheck

We were able to describe how to deal with these variable items by labeling them and referring to them by an abstract label, ignoring the **value** the variable might have in any particular instance.

The power of computing is the power of abstraction, of specifying how to handle many instances of something like an employee record in the same general way, with appropriate specific customizations. Thinking like a programmer means identifying

the appropriate generalizations and customizations necessary to perform a sequence of actions that accounts for all the likely variations of the target situation.

Faced with a large set of similar but different items and a system of complex processes, such as filing tax returns or registering students for classes or selling books over the Internet, programmers impose order by *abstracting away from the messy individual cases* to the appropriate level of generality. They do this by asking themselves questions like these:

- What are the most important categories of items in this domain?
- Within any category, what are the important qualities that all these items share?
- What actions have to be taken for every item in this category?
- How do these entities, attributes, and actions differ with individual cases?
- How do we decide which action to take in any individual case?

Since the answers to these questions will determine the representational structure of the digital artifact—how the processes in the real world are coded within the machine—they cannot be left to the programmers alone. *They must be part of the fundamental process of design.* Using pseudocode and flow charts, all team members can communicate and discuss these crucial abstractions. The more explicitly designers analyze such decisions, the better they will be able to anticipate the many variations that actual users will bring to the delivered system.

Scripting Behaviors

Human culture can be seen as a progression of ever more elaborate strategies for scripting individual and group behaviors and for developing uniform behaviors within larger and larger groups. Rhythmic clapping and movement—the rudiments of music and dance—can be thought of as programmed behaviors, sufficiently general and recognizable actions that can be replicated by a wide range of performers in a varied but uniform manner. Hunting, fishing, gathering and sharing food, religious **rituals**, weddings, funerals, technologies of tool-making, child care, clothing, cooking, agricultural festivals of sowing and harvesting all require the transmission and reproduction of patterns of behavior that incorporate conditional rules for appropriate variations based on the immediate context. Symbolic forms of representation allow us to describe shared patterns of behavior without acting them out in real time, so that they can be remembered and reproduced with greater accuracy and complexity.

For people living in an oral culture, proverbs are a suitably brief genre, inscribing and transmitting useful general rules for behavior and for understanding the world by coding them in familiar language with conventions like rhyme, rhythm, and metaphor to make them memorable without having to write them down, for example:

The early bird catches the worm.
A stitch in time saves nine.
Leaves of three, let it be. (For avoiding poison ivy.)
The race is not always to the swift.
Do unto others as you would have others do unto you.

In a culture that has a writing system, people can inscribe behavioral scripts in stone or place them on sacred scrolls so they can be read out regularly as a guide to shared rules of behavior:

For six days you shall labor and do all your work. But the seventh day is a Sabbath to the Lord your God; you shall not do any work. (Exodus 20:9–10)
You shall keep the festival of unleavened bread. For seven days you shall eat unleavened bread, as I commanded you, at the time appointed in the month of Abib; for in the month of Abib you came out from Egypt. (Exodus 34:18)

Writing fixes behavior with greater uniformity for more people over longer periods of time, especially when it is accompanied by ritualized enactments that recur at predictable intervals. Time-keeping media technologies are also important in scripting behaviors. Synchronizing our reading of where we are in the sequence of natural events—for example, the daily, monthly, and yearly rhythms of sun and moon—allows us to coordinate our cultural patterns, like eating meals or planting crops, with nature and with one another.

The invention of the printing press helped us to standardize our observations and performance of complex processes—like the experiments that are the foundation of the scientific method or the best methods for dairy farming—and to share our strategies for improving them. The Industrial Revolution transformed many processes that are foundational to human culture by automating them, leveraging smaller amounts of human effort for vastly greater productivity. Manufacture of clothing and shelter and transportation was transformed by using compressed air, waterpower, carbon-burning engines, and eventually electricity to perform tasks that were formerly done by hand.

The invention of recording technologies for images and sounds has expanded our ability to observe and document processes, allowing us to repeat and segment events that would otherwise be lost in time, as well as to preserve and distribute our recordings.

The computer has augmented all of these modes of synchronizing, standardizing, automating, and recording sequences of behavior. It has increased scripting by word of mouth by facilitating voice, text, and image transmission—the equivalent of proverbial wisdom transmission for ever-larger social networks. Computer networks have also increased the dissemination, retrieval, and refining of more complex written rules, from legal contracts to the demanding rituals of French cooking. They have increased our control

over mechanized processes, from automobiles to nuclear power plants to remote-controlled weaponry and even to telescopes and probes in space by allowing us to control them through automated conditional processes. And they have augmented our ability to observe and record events over time.

Computers have given us the ability to fix, share, and refine procedural knowledge. A recipe in a book is descriptive (or declarative) knowledge. A recipe enacted by an expert is procedural knowledge. A computer displaying coded instructions is imparting descriptive knowledge; a computer running those instructions as a set of actions is demonstrating procedural knowledge. When we read a set of instructions in a book we can imagine what will happen when they are executed. When we run a set of instructions on a computer, we can explore what will happen under multiple conditions. It is descriptive like a recipe in a book; but it is also procedural like a chef executing a recipe—or like 1,000 chefs performing 1,000 variations of the same recipe. *The power of procedural representation is a tremendous resource for design, a cultural power continuous with dances, proverbs, and rituals as well as with scientific experiments and mechanical engineering.*

Computer programmers think of the procedures for doing things as separate from the particular code with which the procedures are implemented and they refer to these more abstract procedures as algorithms. A key part of computational analysis is identifying the appropriate algorithm, the optimal sequence of instructions for performing a particular task or part of a task. Some algorithms are usable in multiple situations for the same kind of process, such as sorting a list, searching a database or archive, compressing files, or generating random numbers. The field of computer science is concerned with exploring the properties of algorithms and inventing new algorithms for processing **information**. The field of computer programming is concerned with applying and refining known algorithms for practical purposes. These algorithms are part of the cultural conventions that shape a particular software environment, although they may be hidden from the user or nonprogramming designer.

Mathematical formulas, like the ones for determining the area of geometrical shapes, are algorithms that always give the single correct answer, and that we therefore follow in precisely the same way each time we execute them. Much of computer science is concerned with mathematical precision in algorithms. But in daily life we generally make use of rougher algorithms called **heuristics**, sometimes called "rules of thumb." Heuristics tells us in a mostly reliable way how to get to an acceptably close-to-accurate answer, such as estimating relative distances on a map by comparing them to the size of our thumb. Often the most advanced computational solution to a problem—the standard algorithm—is only a rough heuristic rather than a more precise methodology because the computer only has partial information, the outcome of a process is basically unpredictable, or a more mathematically accurate answer would take too much processing power for current machines. The proverbs discussed

earlier in this chapter are heuristics that guide the thoughts and behavior of an individual toward the cultural norms and received wisdom of the larger group, without aiming for literal truth or mathematical accuracy. But a human being can compensate for the approximation of a proverb by not expecting " a stitch in time" to save exactly "nine" other stitches. Humans can also apply the same process description metaphorically, recognizing that other problems besides fabric tears are best taken care of before they get much bigger. Computer programs, on the other hand, need algorithms that explicitly state every condition they should be aware of and every action they should take. So human users need to be told how to gauge the reliability of the computer's methods, by designs that bring **visibility** to the underlying algorithms.

Although the algorithms on which computer programs are based are crucial parts of the structure of digital artifacts, they are often invisible to designers and interactors. Sometimes proprietary rights keep them invisible since the best search engine or recommendation system algorithm is a valuable invention in the marketplace. We also do not have many conventions for illustrating algorithms outside of mathematical notations. Algorithm visualization is a focus of research in information visualization (figure 4.6). It is useful as an aid to the teaching of computer science, and it should also be a focus for interaction designers. The more we understand about the way information is represented and processed within the machine, the greater our power to design it. And the better we can communicate the basis of the machine's decision making to the interactor, the greater the experience of agency.

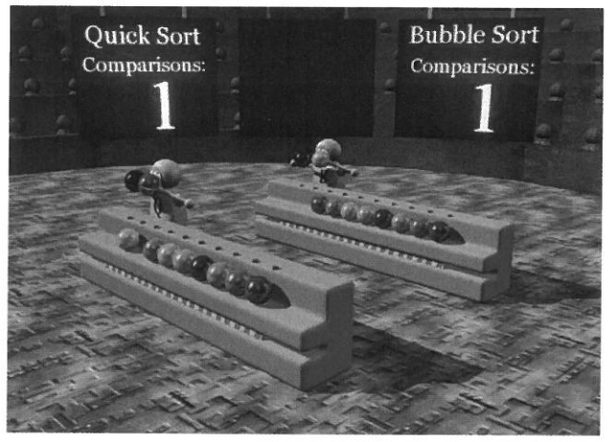

Figure 4.6
A comparative visualization of two sorting algorithms in the form of an animation with robots representing the actions of the processor. From a video by <http://zutopedia.com>, available on YouTube.

State

Since computer environments are always changing, we speak of them as having **state.** It is useful to remember that when we talk about state as a powerful abstract concept it is grounded in the physical reality of the underlying machine code and the millions of electrical switches set to 0 or 1 in patterns that we can interpret as representing higher-level codes. Programs have state, and parts of programs can also have state. The state of a program is the state of all the bits in which it is instantiated. In early programming environments programmers made use of a "core dump," a printout with a 0 or 1 representing every bit in computer memory. When a program crashed—when it could not figure out how to execute an instruction because of a typo or logic error of some kind—the program displayed its state at the moment of failure as a long list of 0s and 1s formatted into bytes. Programmers would find the bytes that indicated the location of the machine code instruction being executed at the time of the crash and the values of the variables, translating them from bits into ASCII characters, binary numbers, or primitive instructions, in order to determine what went wrong. We glimpse this underlying layer in error messages, usually expressed not in 0s and 1s but in hexadecimals (figure 4.7). We can think of an executing program as always in flux but capable of being captured in such snapshots when we need to, such as when we want to save a dynamic environment like a game or a tax return editor and restore it at a later time.

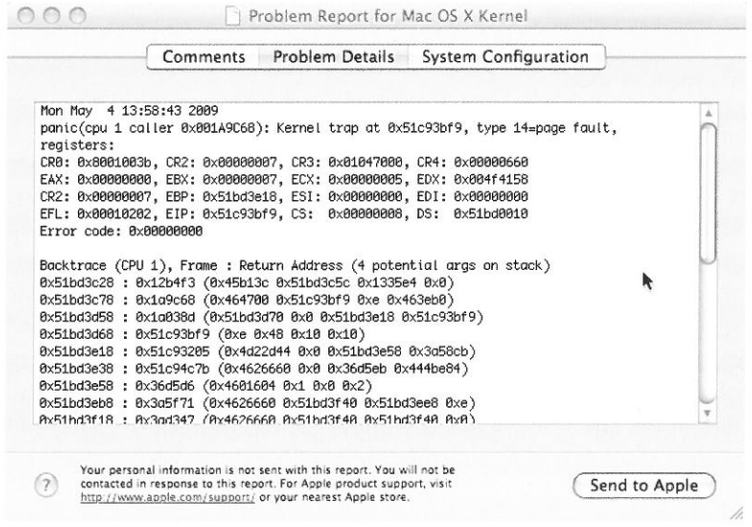

Figure 4.7
System errors producing problem reports like this one provide end-users with a rare glimpse into the bit abstraction layer that underlies all digital representation.

Because computer applications are procedural, we do not have to reproduce the entire program and all the associated data structures to capture the state of a system. *We only have to capture the values of all the variables.* A "saved game" is not a bit-for-bit duplicate of the entire game; it is a file with the values of all the game variables. The master program can procedurally reproduce the desired state by reading this small data set and reloading these saved values into the appropriate variables. In a multiplayer world each player sharing the same virtual space has local copies of the large image files and the rules for generating the world, and the central server only needs to receive and send a limited number of variable values in a standard common format to keep everyone's picture of the state of the world up to date. An image editor or word processor is similarly able to undo changes because it can keep track of the previous states of a document without having to save multiple copies. *State is a useful concept for designers in any situation where users will want to save or share a procedural environment, or to undo/redo changes to a digital artifact.*

State can also belong to entities within a world constructed of code. For example, all the characters in a multiplayer world can have individual states consisting of variables, like strength or intelligence, and possessions, like food and weapons. State is also a powerful conceptual framework for thinking about conditional processes. Instead of regarding a process as a unidirectional **branching tree**, with processing proceeding from beginning to end, we can see it as an interconnected **network**, with each **node** representing a distinct state and the lines or paths between them showing the preconditions for moving from one state to another. For example, figure 4.8 shows a **state diagram** of a virtual dog. If he is hungry he can pass to one of two states:

sated if he is given a meal of food
cranky if a second mealtime passes while he is already in a hungry state

The sated dog can become *playful* in the presence of a playmate or *curious* in the presence of a scent. He may run away if he is cranky, but he can be distracted by curiosity. We can represent all of these states as nested if/then statements or as a multibranching flow chart, but it makes more sense to represent them in a state diagram.

Many objects in computational systems can be understood in terms of a state diagram, such as a cursor that can change to a point, hand, highlighter, brush, and so on, with different functionalities associated with each state. The concept of state can summarize multiple variables in a complex system. For example, Will Wright's original design for The Sims included motives such as hunger, hygiene, and entertainment that were summarized in an overall state of happiness of the character (figure 4.9).

State distinctions in real life and in virtual environments often reflect power relationships or special privileges. For example, credentials of various kinds (such as college degrees; birth, death, marriage, and divorce certificates; drivers licenses; and visa and citizenship documents) can also be understood in terms of state, with preconditions officially monitored, legal rituals accompanying the state transitions, and state-specific constraints and affordances. A power-up in a platform video game is a similar kind of state transition. For example, in the Super Mario Brothers video game, the Mario figure can become huge enough to stomp on obstacles he has to jump over in his usual state (figure 4.10) or small enough to fly into tiny places to find hidden treasures. State distinctions can provide a useful framework for distinguishing between stages of a process, and sometimes a master state can affect multiple variables.

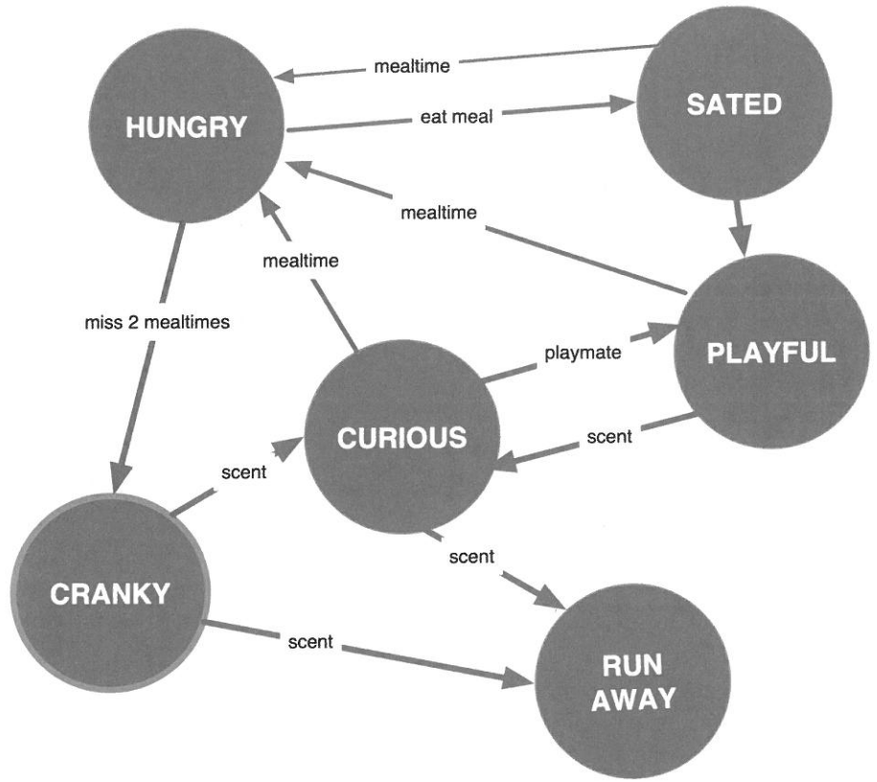

Figure 4.8
State diagram of a virtual dog. The hungry state occurs at time steps representing meal times. The conditions for advancing to the other five possible states are indicated on the arrows. Drawing a diagram like this can suggest new possibilities or refinements of behaviors, such as considering the conditions under which the dog can be distracted from crankiness by curiosity.

```
enum
{
    mHappyLife    =0,
    mHappyWeek    =1,
    mHappyDay     =2,
    mHappyNow     =3,

    mPhysical=4,
    mEnergy       =5,
    mComfort=6,
    mHunger       =7,
    mHygiene =8,
    mBladder =9,

    mMental       =10,
    mAlertness    =11,
    mStress       =12,
    mEnvironment=13,
    mSocial       =14,
    mEntertained=15
};

#define DAYTICKS  720        // 1 tick = 2 minutes game time
#define WEEKTICKS  5040
```

```
// calc and average happiness
// happy = mental + physical

    Motive[mHappyNow] = (Motive[mPhysical]+Motive[mMental]) / 2;
    Motive[mHappyDay] = ((Motive[mHappyDay] * (DAYTICKS-1)) + Motive[mHappyNow]) / DAYTICKS;
    Motive[mHappyWeek] = ((Motive[mHappyWeek] * (WEEKTICKS-1)) + Motive[mHappyNow]) / WEEKTICKS;
    Motive[mHappyLife] = ((Motive[mHappyLife] * 9) + Motive[mHappyWeek]) / 10;

    for (z = 0; z < 16; z++) {
        if (Motive[z] > 100)  Motive[z] = 100;        // check for over/under flow
        if (Motive[z] < -100)  Motive[z] = -100;
        oldMotive[z] = Motive[z];                     // save set in oldMotives (for delta tests)
    }
}
```

Figure 4.9
Excerpts from Will Wright's early draft code for The Sims, showing values for computing physical and mental motives and for combining multiple motives into a calculation of the character's average state of happiness over multiple time steps.

Cyberspace is increasingly populated by complex dynamic systems composed of thousands of variables, such as:

- vast exploratory game worlds with hundreds of thousands of players
- social networking sites with thousands of associated media players and interactive games
- stock trading systems providing access to globally available securities
environmental monitoring systems
- Slide presentation systems that offer one-button changes of color scheme, formatting, and background for the same text content
- control systems for automated manufacturing processes

Figure 4.10
Video game character Mario undergoing a state-transforming power-up that increases his size and the range of actions available to him.

Each of these systems exists in a huge possibility space, making it increasingly difficult to keep track of the big picture, to know what variables to watch in order to understand the state of the system as a whole or to accurately track its individual parts. *We are creating complex digital environments faster than we are creating the means of conceptualizing and overseeing them.* The abstraction and visualization of the state of complex computational systems will therefore be an active area for design in the coming decades.

Modularity and Encapsulation

Modularity and **encapsulation** are key programming strategies for representing complex sequences of behavior by breaking them up into smaller parts that work independently but know how to exchange information with one another.

The more modular a program, the better it can survive the inevitable changes of a development process, the more ***extensible*** *it will be, and the more likely it will continue to be adaptable as new technologies emerge.* Modularity allows programmers to change part of a large system without affecting the other parts, while sharing important elements across the whole system as appropriate. One way to think of modularity is to imagine a software program as an organization with employees, like a storefront bakery. The bakery may have one group of employees to make the pastries, one to wait on customers, and another to work the cash register. Each group understands what a cupcake is and how

to pass it through the system, performing the appropriate actions at each stage. If the pastry chef is experimenting with a new cupcake recipe, it should not distract the other employees from doing their jobs. Similarly, a software program that supports Internet sales of baked goods might be written with a front-end module for customers, a back-end module for employees, and shared database for keeping track of inventory. Redesigning the appearance or navigation of the customer interface should not disrupt the underlying database or the data entry done by the employees.

One of the most useful applications of the strategy of modular design is the separation of the appearance of digital artifacts from their inner structure, of the user interface from the back end. For example:

- Databases that offer multiple views of the same data, such as graphs, pie charts, tables
- Slide presentation systems that offer one-button changes of color scheme, formatting, and background for the same text content
- Separation of display from structure with cascading style sheets in web environments
- Delivery of the same news and entertainment content on multiple-size screens and multiple hardware/software platforms

Modularity of design also allows developers to release expansion packs and level modifiers for games, by adding separate but compatible code onto an existing structure.

Modularity is a fundamental principle of software engineering. It can be implemented in many ways. For example, in the payroll program we might make a subroutine called PAYCALCULATOR that would be called by the main program. Here it is in pseudocode:

```
PAYCALCULATOR (hours, rate) // calling program, passing parameters
CALCULATE calculatedPay = hours x rate //plugging in values passed as
                                                    parameters
RETURN (calculatedPay) //returning a value for calculatedPay
```

In the first line of the program we tell the subroutine PAYCALCULATOR what values to use to calculate the pay. When the program runs, the subroutine will receive two numbers separated by a comma. It will assume the first number represents a particular employee's hours and the second represents the employee's rate, and it will then plug these values into the equation hours x rate and come up with a new value, which it will send back to the program that called it in a variable called calculatedPay. In this example (hours, rate) and (calculatedPay) are variables that are passed as **parameters** between the master program and the subroutine.

Having made this subroutine, we can change the master program so that it does not have to know how the pay is calculated. The relevant portion would read like this:


```
COPY timecard.name = paycheck.name
CALL PAYCALCULATOR (hours, rate) to get (calculatedPay)
Copy calculatedPay = paycheck.pay
IF DIRECTdeposit = true
THEN DEPOSIT paycheck
ELSE PRINT paycheck
END IF
```

The formula for calculating the pay has been encapsulated in the subroutine.

In general, programmers try to divide up the functional components of a software system so they can be tested separately and modified without affecting the other components. This also allows multiple people (or subteams) to work on the same system. As long as all the developers have a common representation of the data structures they will be sharing and the parameters they will be passing, each subteam can develop its own piece of the larger system independently of one another. For example, to handle the complexity of a real corporate payroll program, the functions for figuring out pension deductions or income taxes or charitable contributions can be treated as separate modules from one another and from the module that calculates net pay. As with many aspects of digital systems, standardization allows for coordination among multiple contributors.

When the payroll program calls the CalculatePay subroutine by name it is taking advantage of the computational strategy of **encapsulation,** by which we hide the specific functional details of a module, drawing on it as if it were a **black box** known only by its name and the relevant protocols for sharing information. For example, we do not want to have to look at the binary machine code instruction for contacting the printer every time we want to print something. We encapsulate all of that within the PRINT command of a higher-level programming language or the PRINT menu item in a computer application. Encapsulation is sometimes spoken of as "information hiding" but it is better understood as standardizing the topmost level of an operation, so that other parts of the system can invoke the function without worrying about the details of how it is performed. Encapsulation reinforces modularity. By making one command PRINT for every printer we do not have to think about the individual print drivers for every operating system and every printer family each time we make an application. We just pass the PRINT command on to the system software and let the system find the appropriate device and driver.

Often design of a complex software system starts with the topmost level and black boxes the constituent processes, leaving the details unspecified while the big picture is sorted out. For example, if we are making a procedure for preparing breakfast, we want the top level, the most general part of the program, to make sense regardless of what we are cooking. It might look like this:

```
PLAN menu
LIST ingredients
GATHER ingredients
COOK meal
SET table
SERVE meal
```

In listing the general steps we do not have to think about the details of performing any of them. We have encapsulated the multistep process of cooking the meal into a single "Cook" command. Because these routines are modular, the team that does the cooking does not have to think about gathering the ingredients. Each part of the larger program can be designed independently as long as the coders know how to pass information (shopping lists, groceries, prepared food) from one module to another. Encapsulation helps us to keep track of a large sequence of instructions without getting overwhelmed by the details.

Within each module we would specify the actions in greater detail, continuing to look for opportunities to encapsulate further, building up complexity with multiple **abstraction layers**. At the level of gathering the ingredients, we can begin to think about what generalized actions (**functions**) we would perform regardless of what we are cooking. For example, we have to

```
GO to Refrigerator
OPEN Refrigerator
SELECT item
PLACE item on counter
GO to Pantry
OPEN Pantry
SELECT item
PLACE item on counter
```

We notice that there is some parallelism here. The Pantry and the Refrigerator are similar, and they belong to a category that could have other elements, such as ShoppingBag and BreadBox. All are FoodContainers. So we could save ourselves the trouble of making individual routines for each by making a generalized `GATHER` routine that would work for all of them, and that would also allow for other places besides the counter, as follows:

```
From [FoodContainer] select [Item] and Place [WorkLocation]
```

To identify components of any process requires brainstorming all the examples of specifics while constantly thinking about how to abstract the general case. Food preparation immediately brings to mind refrigerators and pantries, which in turn suggest the general category of FoodContainer. Once we think of FoodContainer, shopping bags and breadboxes come to mind. Once we generalize the container, we realize we could also generalize the work location. Meanwhile other teams are thinking in more detail about the components of other parts of the process. The process of abstraction is iterative and collaborative. Although the advantage of modularity in development is information hiding, it is important for teams to establish clear standards and to remain in touch about shared structures. In 1999 NASA lost a $125 billion Mars orbiter because "one engineering team used metric units while another used English units for a key spacecraft operation" (CNN 1999).

Encapsulation is an important enabling strategy for the Internet (table 4.3), allowing a programming team to make significant changes to modules at any point in the hierarchy without having to worry about affecting or being affected by changes going on in the levels above and below, as long as the innovators observe the standard

Table 4.3
The Open Systems Interconnection (OSI) Basic Reference Model

Layer	Function	Example of Components
7. Application	Provides network services to applications	File Transfer Protocol (FTP) Hypertext Transfer Protocol (HTTP)
6. Presentation	Standardizes and encrypts data to pass it between application and session layers	Multipurpose Internet Mail Extensions (MIME)
5. Session	Establishes connections between computers, ends sessions gracefully	Domain Name Server (DNS)
4.Transport	Transfers data between users, keeping track of segments of data and whether they transmitted correctly	Transmission Control Protocol (TCP)
3. Network	Routing of data chunks of varying lengths through one or more networks	Internet Protocol (IP)
2. Data Link	Arranges bits from the physical layer into logical sequences called frames	WiFi, Integrated Services Digital Network (ISDN)
1. Physical	Provides the standardized electrical and physical specifications for devices, such as pins, voltages, cables, adapters, creating the function of modulation—conversion of digital data from user equipment to signals that can be transmitted by a communications network	Modems, Coaxial Cable, Fiber Optic Cable

The OSI Basic Reference Model, describing the system of enabling technologies of the global information network, is a good example of the principle of encapsulation, which simplifies interaction in a nested hierarchy of functions by hiding information so that each level only sees what is relevant to its own process; for example, programs that are transferring files cannot see anything of the underlying hardware or radio waves, but they can rely on them working as needed because each component is following the official protocols for passing information up and down the hierarchy.

protocols for passing information up and down the line. The reliability of the Internet, which it is easy to take for granted on a day-to-day basis, can be seen as a triumph of computational abstraction.