

Bits don't have error bars: upward conceptualization and downward approximation

Russ Abbott

Department of Computer Science, California State University, Los Angeles, California
Russ.Abbott@GMail.com

Abstract. How engineering enabled abstraction in computer science.

1. *Turning dreams into reality*

What has been is what will be, and what has been done is what will be done; there is nothing new under the sun. Is there a thing of which it is said, "See, this is new"?
– Ecclesiastes 1:9-10

Although Ecclesiastes says “No,” engineers and computer scientists say “Yes, there are things that are new under the sun—and we create them.” Where do these new things come from? They start as ideas in our minds. A poetic, if overused, way to put this is that we—and I’m writing as a computer scientist—turn our dreams into reality. Trite though this phrase may be, I mean to take seriously the relationship between ideas and material reality. Engineers and computer scientists transform subjective experience into phenomena of the material world.

The previous statement notwithstanding, this paper is not a theory of mind. I am not claiming to explain how subjective experience comes into being or how we map subjective experience to anything outside the mind. I do plan to talk about the relationship between ideas and material reality. Section 2 provides more detail on the approach to consciousness I’m adopting. It also provides an introduction to the notion of a level of abstraction and its significance.

Section 3 returns to thought externalization, the ways in which engineers and computer scientists turn ideas into material reality. As Ferguson (1992) says, “The conversion of an idea to an artifact ... is a complex and subtle process that will always be far closer to art than to science.” Section 4 discusses both the bit, the fundamental level of abstraction, and the price computer science pays for working at the bit level and above. Section 5 describes the different approaches engineering and computer science take to design.

Much of this paper talks about how computer scientists and engineers think. For computer science I rely my own intuitions and self-awareness (Abbott 2008a). To a great extent I rely on Ferguson (1992) for the thought processes of engineers.

2. *Subjective experience and levels of abstraction*

A reader of an earlier draft wondered whether I intended to embrace dualism by talking about transforming ideas into physical reality. This section explains why not and goes on to discuss levels of abstraction in general.

The hard problem of consciousness

Subjective experience seems to be universal among human beings.¹ Yet we don't know how it comes about. Presumably it arises from brain activity, but we have no way of explaining how brain activity results in subjective experience. Chalmers (1995) calls this "the hard problem of consciousness."

The really hard problem of consciousness is the problem of *experience*. ... When we see, ... we *experience* visual sensations: the felt quality of redness, the experience of dark and light, the quality of depth in a visual field. Other experiences go along with perception in different modalities: the sound of a clarinet, the smell of mothballs. Then there are bodily sensations, from pains to orgasms; mental images that are conjured up internally; the felt quality of emotion, and the experience of a stream of conscious thought. ... It is widely agreed that experience arises from a physical basis, but we have no good explanation of why and how it so arises.

Chalmers was echoing a problem described much earlier by Leibniz (1714).

Supposing there were a machine, so constructed as to think, feel, and have perception, it might be conceived as increased in size, while keeping the same proportions, so that one might go into it as into a mill. On examining its interior, we would find only parts which work one upon another, and never anything by which to explain [subjective experience].

Levels of abstraction

Leibniz and Chalmers are describing emergence, a problem more general than subjective experience. Here's how O'Connor and Wong (2006) put it.

[Emergent entities and properties] 'arise' out of more fundamental entities and yet are 'novel' or 'irreducible' with respect to them. (For example, it is sometimes said that consciousness is an emergent property of the brain.)

I argue (2006, 2007, 2008b, and 2008c) that emergence is what is known in computer science as a level of abstraction. A level of abstraction is a collection of types (categories of entities) and operations on entities of those types. Every level

¹ Dennett (1988) denies the existence of qualia on "eliminative materialism" grounds. I don't understand that argument. As I discuss later in this paper—and in more detail in (Abbott 2008c)—that something can be reduced to physical primitives doesn't demonstrate its eliminability.

of abstraction has two important properties. (a) It can be characterized independently of its implementation—the (emergent) entity types, properties, and operations can be described abstractly. (b) When implemented, it is reducible to pre-existing levels of abstraction.

Consider any computer program, say Microsoft Word. It implements a level of abstraction that has such (emergent) entities as words, paragraphs, documents, etc. These entities have such (emergent) properties as font size and style, margins, etc. If one looks at the code that implements Microsoft Word one will not see words, paragraphs, documents, or their components or properties. But we do not consider it mysterious that words, paragraphs, etc. “emerge” from that code.

A specification pins down the entities and properties on a level of abstraction. One may require of Microsoft Word, for example, that when one double clicks the text one “selects” a “word,” and when one triple clicks one “selects” a “paragraph.” These requirements define (abstract) relationships among paragraphs, words, and click operations. They don’t define what a word is in terms of simpler physical elements such as bits or electrons.

That’s how it should be. A level of abstraction may be implemented by software in any number of ways. There is no necessary relationship between words and bits or electrons. All that matters is that words behave as expected with respect to the other elements on its level of abstraction. It’s up to the implementation to decide how to implement words.

Searle (2004) invented a nice phrase to describe entities and entity types on a level of abstraction. He called them causally reducible but ontologically real. Higher-level entities are causally reducible because when a level of abstraction is implemented one can see how it works by looking at the implementation.

Higher-level entities are ontologically real because their functioning can only be described in terms of the entities themselves. When the entities are man-made that description is a specification. But naturally occurring entities must also be described in terms of themselves. Biological organisms, for example, have the properties of being alive or dead, and they perform functions such as eating, sleeping, seeing, moving, and reproducing. These concepts apply only at the biological level; they make no sense at the level of chemistry or physics. The reality of higher-level entities and entity types inheres in their specifications.

Functional decomposition vs. stigmergic design

Where did we get the idea that higher level entities and properties must be reducible to lower level components? Well, how could it be otherwise? If an entity is composed of lower level elements, how could it not be that those lower level elements serve as the components of the higher level entity? This idea is so natural and intuitive that it has been adopted as the design strategy known as *functional decomposition*. Using functional decomposition one designs a system as a collection of independent component subsystems so that the system’s functionality is a composition of the functionalities of the subsystems. Applied recursively this

leads to a hierarchy of functional units. The result is a system whose physical (component) structure corresponds to its functional structure—a very clean design.

But as Microsoft Word illustrates, the functionality that software produces can result from an interaction among elements that do not have any of the properties of the resulting entities. In fact, most software does not lend itself to functional decomposition. (See Section 5.) Yet according to O'Connor and Wong (quoted above) that's why emergence seems so mysterious. Even the brilliant Leibniz was misled. After concluding that one wouldn't find the components of subjective experience in the mechanical workings of a mind, he wrote the following.

Thus it is in a simple substance, and not in a compound or in a machine, that [subjective experience] must be sought for.

In other words, Leibniz concluded that since subjective experience is not composed from lower level elements, it must be primitive, i.e., a “simple substance.”

The design strategy that produces emergent functionality—and that contrasts with functional decomposition—might be called *stigmergic design*.² Software developers use it all the time.³ Nature uses it in biological and social entities. What is philosophically important is the realization (a) that stigmergic design is a valid design strategy and (b) that it can produce entities and properties from apparently unrelated lower level entities and properties. Had this concept been available to Leibniz, he might not have drawn the conclusion he did. I return to the discussion of stigmergic design in Section 5.

Subjective experience as a level of abstraction

It seems reasonable to assume that consciousness is implemented as a level of abstraction by physical processes in the brain. Here's how Searle (2004) puts it.

[Conscious states] are real phenomena in the real world. ... [They] are entirely caused by lower level neurobiological processes in the brain. ... You can do a causal reduction of consciousness to its neuronal substrate, but that reduction does not lead to an ontological reduction because consciousness has a first person ontology, and you lose the point of having the concept if you redefine it in third person terms.

The problem is that we don't know how the brain does it. We can explain how Microsoft Word produces words, paragraphs, and documents, but we can't yet explain how brain functioning produces subjective experience. Nonetheless, from here on I will assume (a) that subjective experience is a level of abstraction implemented by the brain, (b) that we each have the experience referred to as having an idea, (b) that we are aware of ourselves as having ideas, and (c) that we understand in more or less the same way what it means to say that one has an idea.

Given the preceding, I will refer to the phenomenon of having an idea non-dualistically and without further explanation or apology and will return to the per-

² The term *stigmergy* was coined by Pierre-Paul Grassé (1959) to explain how social insect societies operate. Social insects interact in part by leaving markers in the environment.

³ Related software terms include object-oriented design, service oriented architecture, tiered design, platform-based design, and (of course) levels of abstraction.

spective that engineers and computer scientists turn ideas into material reality.⁴ Although engineering and computer science are similar in that way, there is a difference in the kinds of realities created. Computer scientists create (physically implemented) symbolic reality. Engineers create material objects that act in the physical world.⁵ The consequences of this difference are far-reaching.

3. *Thought externalization: engineering is to sculpture as computer science is to music*

If I could say it in words there would be no reason to paint. — Edward Hopper

The first step in turning an idea into reality is to externalize the idea. By externalizing an idea I'm referring to the conversion of the thought from something completely subjective to an external representation—a representation outside the mind in a form that allows the thought to be examined, explored, and communicated. The pervasive example is natural language. Most disciplines use natural language to externalize thought. Diagrams and equations are other examples.

Most externalized thought is intended for human consumption. Expressions in natural language as well as equations and diagrams are meaningless except to other human beings. An externalized thought has the same intentional property as thought itself. It is about something, and for an externalized thought to be about something requires that it be re-internalized, i.e., understood and converted into (intentional) thought. Here's how Ferguson puts it.

Engineers start with visions of the complete machine, structure, or device. ... [They first] convert the visions in their minds to drawings and specifications, [which] are expressed in a graphic language, the grammar and syntax of which are learned through use [and which] has idioms that only initiates will recognize.

But as illustrated by Hopper's remark sometimes the thought is transformed directly into the thing itself. A Hopper painting is the thing itself. No intermediate form can adequately represent the idea—or there would be no reason to paint. Along the same lines Ferguson distinguishes between engineers and artisans.

If the idea [of the thing to be made] is in the head of an artisan, he can make the thing directly. ... If the idea of the thing to be made is not in the artisan's head but in the engineer's, the engineer [uses] drawings to convey to workers what is in [his] head. ... The difference between the direct design of the artisan and the design drawing of the engineers are differences of format rather than ... conception.

Thought externalization in computer science is different. Computer scientists externalize thought as software. Software has the important property that it is *both*

⁴ Mitchum (1978) made this point 3 decades ago. He also noted that science turns reality into ideas. Debora Shuger (personal communication) added that humanists turn reality into dreams. And Paul Erdos is widely quoted as saying that mathematicians turn coffee into theorems.

⁵ In this article I won't be discussing hybrid systems—physical systems with embedded software.

intentional and the thing itself. It refers in the same way that expressions in natural language refer: one can read it and understand it as having meaning. Software is also the thing itself—almost. With the help of a computer, software acts without the need for human understanding. (See Abbott (1998b).) To the best of my knowledge, music is the only other discipline that can externalize thought in a form (a) that is intentional/symbolic and (b) that may be actualized without further human participation. Even before computers, player pianos were able to convert the symbolic expression of musical thought into music. Engineering is approaching this capability, but it isn't there yet. A fully automated computer aided manufacturing capability—insert a design; get a product—would be equivalent.

4. *The bit: where thought and matter meet*

The term *bit* is used in three overlapping ways.

1. *Bit* can refer to a binary value, i.e., either **true** and **false** as in Boolean logic. A bit in this sense, i.e., as the notion of true or false, is a thought, an idea in our minds just as **true** and **false** are.
2. *Bit* can refer to a mechanism or means for recording such a value. A bit in this sense is part of the material world, typically a unit of computer storage. It is a physical device that (a) is capable of being in either of exactly two states and (b) can be relied on always to be in one of them.
3. *Bit* can refer to a unit of information as in information theory. I'm not using *bit* in this sense in this paper—although a *bit* in the second sense can be used to represent a *bit* in this third sense.

The bit is a fundamental example of externalized thought. It is both a Boolean value (a thought) and a physical device (a part of the material world). Circuits that when run can be understood as performing Boolean operations provide a way to glide gracefully back and forth between *bit* as thought and *bit* as material device. Bits and the physical machinery that operate on them enable us to externalize Boolean operations and values (thoughts) and to manipulate them in the material world. The bit is where thought and matter meet, an extraordinary achievement.

Engineering is a physical discipline. Physical disciplines involve physical devices and materials, which are never perfect and never the same from one instance to another. To determine how a physical device behaves, one measures it—often multiple times. The resulting sets of data points typically contain ranges of values. Such data sets have average values and error bars.

The physical devices used to store and manipulate bits have the same properties as any other physical device. In particular, they have error bars. Yet computer scientists don't see these aspects of physically implemented bits. Machinery built by engineers hides the reality of analog values and error bars from those of us who use bits. It is because of this machinery that as far as computer science is concerned bits don't have error bars. As the interface (a) between engineering and

computer science and (b) between thought and matter the bit serves as the foundation upon which all other levels of abstraction can be implemented.

As externalized thought, the bit also externalizes the forbidden fruit of the tree of knowledge. Like thought itself it both gives us a way of gaining leverage over reality while at the same time separating us from it. Because every conceptual model implemented in software is built on bits, every software-based conceptual model has a fixed bottom level, a set of primitives. Whatever the bit—or the lowest level in the model—represents, those primitives serve as the model’s floor. It’s not possible to decompose them and model how they work.⁶

Because software models have a fixed set of primitives, it is impossible to explore phenomena that require dynamically varying lower levels. A good example of a model that needs a dynamically varying lower level is a biological arms race. Imagine a plant growing bark to protect itself from an insect. The insect may then develop a way to bore through bark. The plant may develop a toxin—for which the insect develops an anti-toxin. There are no software models in which evolutionary creativity of this richness occurs. To build software models of such phenomena would require either that the model’s bottom level be porous enough that entities within the model are able to develop capabilities that sabotage operations on that lowest level or that the lowest level include all potentially relevant phenomena, from quanta on up. Neither of these options is currently feasible.⁷

Another example of a model that needs a dynamically varying lower level involves the gecko, a macro creature, which uses the quantum van der Waals force to cling to vertical surfaces (Kellar, et. al. 2002). Imagine what would be required to build a computer model of evolution in which that capability evolved. Again, one would need to model physics and chemistry down to the quantum level.

The inability to develop models with dynamically varying lower levels is not limited to software. It is a problem with any finite conceptual model. We don’t seem capable of building models that at the same time (a) have a lowest level and (b) can be extended downward below that lowest level. Yet engineering continually extends its models downward. How? The next section answers that question.

5. *Static and functional structures*

To a first approximation most systems may be understood from two complementary perspectives: static and functional. A static view describes the system’s fixed skeletal structure—if it has one. A functional view describes how the system’s

⁶ One might object that when bits are used to represent numeric values that are derived from an equation-based model, they don’t represent primitive elements. I wouldn’t dispute that. But equation-based models are science and engineering models, not computer science models.

⁷ This poses a nice challenge to computer science: develop modeling mechanisms in which the lowest level of the model can be varied dynamically as needed.

functionality is implemented by the functioning of and interactions among its components. In medicine, for example, these two views are referred to as anatomy (structure) and physiology (functioning).

Although structure vs. function makes a nice division of labor, most systems are not so easily decomposable. Biological organisms may have what we think of as an anatomy, but most of the atoms making up that anatomy are regularly replaced. Part of the physiology of biological organisms is to repair and replace their anatomical structures. The same is true for social organizations. The anatomy of an organization such as a government consists of its institutional organs—a legislature, a judicial system, etc. But these are composed of people, who are replaced regularly. In addition, even the constitutional structure of most social organizations is subject to change—although usually with significant constraints.

Stigmergic design and upward conceptualization

The rest of this section continues the discussion of functional decomposition and stigmergic design begun in Section 2.

What about software? Does it have a static structure? The static structure of interest is the structure in place when the software executes. Krutchen (1995) suggests that it is useful to understand software from 5 perspectives. Only the *Process View*—what happens during execution, including the creation and destruction of variables, objects, stack frames, tasks, etc.—captures the static structure of executing software. Clearly this is not static. Yet at any instant, it is those elements and their interrelationships that make up the static structure of the executing software.

A good way to think about executing software is as a continually changing collection of interacting entities—variables, objects, tasks, etc. The job of a software developer is to write software that creates and controls software entities. Software creates functionality through the interaction of these entities. Section 2 referred to this as *stigmergic design*. Since each software entity exists on some level of abstraction, computer scientists tend to understand software as the art of using one level of abstraction to implement another. Just as multi-celled organisms result from the patterned interactions of individual cells and social organizations result from the patterned interactions of biological organisms, much of software grows in an equally organic manner. The result is a series of upwardly conceptualized levels of abstraction—starting with the bit.

Functional decomposition and downward approximation

Unlike computer science, engineering builds objects whose design must work in the material world. Ferguson quotes the definition of engineering from the 1828 charter of the British Institution of Civil Engineers. “Engineering is ‘the art of directing the great sources of power in nature for the use and convenience of man.’” Petroski (1996) adds, “[Although] engineering is the art of rearranging the materials and forces of nature, the immutable laws of nature are forever constraining the engineer as to how those rearrangements can or cannot be made.”

Engineering is both cursed and blessed by its attachment to physicality. It is cursed because in molding and modeling physical reality one can never be sure of the ground on which one stands. All engineering models are approximations. But engineering is also blessed by its attachment to physicality. For any issue one can decide how deeply to dig for useable physical bedrock. Engineering design rests on two techniques: physical approximation and functional decomposition.

Physical approximation. To reduce model complexity, engineers chose models that provide the necessary assurances on as high a level as possible—a civil engineering model may include the load bearing properties of a steel beam rather than the chemical bonds that produce those properties. Ferguson puts it this way.

The engineering sciences ... differ from pure science in that they have an array of abstract concepts, independent of science, that serve as a framework within which technical problems can be analyzed. ... Informed judgment must decide to what extent calculations involving idealized processes can be depended upon ... because mathematical models are always less complex than [nature].

But engineering is also blessed by its attachment to physicality. If the accuracy of a model is inadequate, engineers can replace it with one that models more primitive physical phenomena. The secret of engineering's ability to dig beneath the floor of its models is that as humans they can create new models when needed.

Functional decomposition. Functional decomposition works well when functional design can be mapped to static structure. Many engineered systems may be decomposed into component subsystems. If the subsystems can be modeled independently, then the system as a whole can be modeled as the collection of subsystems along with a model that ties the subsystems together. This often works quite well. But it is not infallible. To paraphrase the Commission on Engineering and Technical Systems of the National Academy of Engineering (2000),

When engineering systems fail it is often because of unanticipated interactions (such as acoustic resonance) among well designed components that could not be identified in isolation from the operation of the full system.

Problems often arise (a) at a level below the system primitives (approximation fails) and (b) only when the system components are joined to make the entire system (functional decomposition fails). So engineering design is often a continuing search for models of reasonable complexity that provide an acceptable approximations to reality. Engineering is continually approximating downwards.

6. *Summary*

Engineers and computer scientists say there are new things under the sun. Nature agrees—new viruses, new bacterial, new species, lots of new things. Is nature a blind engineer or a blind computer scientist? Engineers build systems top-down. But like computer scientists, nature builds phenomena bottom-up, level of abstraction by level of abstraction. Nature is a blind programmer.

References

- Abbott, Russ, 2006, "Emergence explained," *Complexity*, Sep/Oct, 2006, (12, 1) 13-26. Preprint: http://cs.calstatela.edu/wiki/images/9/95/Emergence_Explained-Abstractions.pdf.
- Abbott, Russ, 2007, "Putting Complex Systems to Work," *Complexity*, 2007, (13, 2) 30-49. Preprint: http://cs.calstatela.edu/wiki/images/c/cb/Putting_Complex_Systems_to_Work.pdf.
- Abbott, Russ and Chengyu Sun, 2008a, "Abstraction abstracted" In *Proceedings of the 2nd International Workshop on the Role of Abstraction in Software Engineering*, (Leipzig, Germany, May 11, 2008). ROA '08. ACM, New York, NY, 23-30. Preprint: http://cs.calstatela.edu/wiki/images/5/56/Abstraction_abstracted.pdf.
- Abbott, Russ, 2008b, "If a tree casts a shadow is it telling the time?" *Journal of Unconventional Computation*, Vol. 5, No. 1, pp. 1-28, 2008. Preprint: http://cs.calstatela.edu/wiki/images/6/66/If_a_tree_casts_a_shadow_is_it_telling_the_time.pdf.
- Abbott, Russ, 2008c, "The reductionist blindspot" *Draft*. http://cs.calstatela.edu/wiki/images/9/93/The_reductionist_blind_spot_and_Levels_of_Abstraction.pdf.
- Autumn, Kellar, et. al. 2002, "Evidence for van der Waals adhesion in gecko setae," *Proceedings of the National Academy of Sciences*, August 27, 2002, 10.1073/pnas.192252799. <http://www.pnas.org/cgi/reprint/192252799v1>.
- Chalmers, David J. 1995 "Facing up to the problem of consciousness," *Journal of Consciousness Studies*, Volume 2, Number 3, 1995, pp. 200-219(20).
- Commission on Engineering and Technical Systems, National Academy of Engineering, 2000, *Design in the New Millennium*, National Academy Press. http://books.nap.edu/openbook.php?record_id=9876&page=R1.
- Dennett, Daniel 1988 "Quining Qualia," in A. Marcel and E. Bisiach, eds, *Consciousness in Modern Science*, Oxford University Press 1988. <http://ase.tufts.edu/cogstud/papers/quinqual.htm>.
- Ferguson, Eugene. 1992, *Engineering and the Mind's Eye*, MIT Press.
- Grassé, Pierre-Paul 1959 "La Reconstruction du nid et les Coordinations Inter-Individuelles chez *Bellicositermes natalensis* et *Cubitermes* sp. La théorie de la Stigmergie: Essai d'interpretation du Comportement des Termites Constructeurs," *Insectes Sociaux*, 6:41-81.
- Krutchén, Phillippe 1995 "Architectural Blueprints--The '4+1' View Model of Software Architecture," *IEEE Software*, 12(6) Nov. 1995
- Liebniz, Gottfried Wilhelm 1714 *Monadology*.
- Mitcham, Carl, (1978), "Types of Technology," *Research in Philosophy and Technology*, Vol. 1, 229-294.
- O'Connor, Timothy and Hong Yu Wong 2006 "Emergent Properties", *The Stanford Encyclopedia of Philosophy (Winter 2006 Edition)*, Edward N. Zalta (ed.), URL = <http://plato.stanford.edu/archives/win2006/entries/properties-emergent/>.
- Petroski, Henry 1996, *Invention by Design; How Engineers Get from Thought to Thing*, Harvard University Press.
- Searle, John 2004 *Mind: a brief introduction*, Oxford University Press.

All Internet accesses are as of May 26, 2008.