The large-scale structure of software-intensive systems

Grady Booch*

IBM, 6533 West Prentice Avenue, Littleton, CO 80123, USA

The computer metaphor is dominant in most discussions of neuroscience, but the semantics attached to that metaphor are often quite naive. Herein, we examine the ontology of software-intensive systems, the nature of their structure and the application of the computer metaphor to the metaphysical questions of self and causation.

Keywords: software-intensive system; software architecture; computer metaphor

1. INTRODUCTION

In the unfolding of the human experience, the meaning of the self, the nature of sentience, the dichotomy of the brain and of the mind are issues that have been deeply pondered and yet that remain profoundly without closure. One question in particular stands out: is the self emergent, or is the self a product of external causes? The journey of consideration for this question quickly leads to two other questions, even more general in nature: Can any significantly complex system exhibit interesting behaviour without some element of externally applied top-down causes? Indeed, is there even such a thing as a closed system?

This paper examines these issues from the perspective of the computer metaphor.

2. THE NATURE OF METAPHOR

As Dr Ellis has noted, ‘the computer metaphor is dominant in most discussions of neuroscience’1. Now, I am not a neuroscientist: the only defensible qualifications I have in that domain are the possession of a brain, a mind and a singular instance that I self-identify as an I. However, I do have a modest understanding of computing, and thus as an outsider (coming from the domain of software-intensive systems) looking in (to the domain of the neurosciences), I suggest that one must be careful not to apply that metaphor too naively. To do so, if I may be direct, would be akin to my trying to reason about the New York Stock Exchange by studying the traces from a few digital logic analysers scattered about the globe. These may be the only things that I think I can measure. I undoubtedly would learn some things. But I doubt I would learn much about macroeconomics.

In every age, the then contemporary context of human life frames how we approach the universe and offers us a language whereby we may describe and reason about the structure and the dynamics of that universe. Indeed, it is likely that we are mental prisoners of our own making, for it is unusual—if not extraordinary—to be able to transcend the shackles of the metaphors that surround us. As Lakoff [1] so beautifully describes in Women, Fire, and Dangerous Things, ‘to change the very concept of a category is to change not only our concept of the mind, but also our understanding of the world’. Lakeoff & Johnson [2] elaborate in Metaphors We Live By by noting, ‘the way that we think, what we experience and what we do every day is very much a matter of metaphor’.

The construction of Maus’ historical timeline is open for considerable debate. His is a particular dead-white-guy-Western-centric point of view that sidesteps the rich legacy of Asian cultures and religions, for example; also, he stops short of exploring the idea of self-as-a-machine, a concept that flowed first from the Industrial Revolution and now finds an anchor in our self-styled Information Revolution. However, this debate is not the focus of our current discourse. Without fussing over the details, it is fair to say that in different times and in different cultures, our understanding of self has changed, and that our understanding is generally always couched in the language of the then contemporary metaphors.

3. THE COMPUTER METAPHOR

In early-recorded history, citizen scientists such as Yan Shi and Hero of Alexandria, and then Al-Jazar

*gbbooch@us.ibm.com

1Dr George Ellis, letter to author, 21 June 2009.

One contribution to a Theme Issue ‘Top-down causation’.
built automata that mimicked some aspect of nature or of the human figure. In the flowering of the Renaissance and then the Enlightenment, catalysed by a deeper understanding of the mathematical and scientific foundations of the universe, the concept of man as a machine emerged. As the infrastructure for creating mechanical devices of greater precision matured, cunning devices such as von Kempelen’s Mechanical Turk (which played a fair game of chess albeit by fully human means) and de Vaucanson’s Canard Digerateur (which simulated the process of ingestion and excretion in a duck) were created. While these were little more than amusing objects of conversation, de la Mettrie’s work, Man a Machine, introduced the highly controversial (at the time) philosophical idea that man was indeed nothing more than a machine.

This concept—of man as nothing more than a biological mechanism, without spirit—and its corollary—the computer-as-a-brain—gained considerable traction as computers first entered public view in the 1940s and 1950s. In particular, Grey Walter’s autonomous robots and Norbert Wiener’s writings on cybernetics brought substance to this point of view. Even more recently, this concept is echoed by many in contemporary philosophy and physics. Stephen Hawking, for example, writes ‘I regard the brain as a computer which will stop working when its components fail. There is no heaven or afterlife for broken down computers; that is a fairy story for people afraid of the dark’ [4].

Ignoring the philosophical or emotional reactions, one might have to such a proposition, a calm analysis would suggest that one might reasonably ask what semantics Hawking attaches to the term ‘computer’.

Turing [5] pondered that issue quite thoroughly in his exploration of the computability of the mind, propelled by a thought experiment he called the ‘imitation game’. That work in turn was an outgrowth of his thoughts on computability and the limits of what could and could not be calculated automatically. Turing’s investigation led him to devise a Turing machine whose structure consisted of an infinite memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol and its behaviour is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behaviour of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings.

The importance of Turing’s contribution to theoretical computing cannot be understated. Among other things, the notion of Turing completeness gives rise to a grounded meaning of computability. Specifically, the Church–Turing thesis asserts that any decidable effect that can be computed can always be computed by a universal Turing machine. Furthermore, no matter the degree of perceived complexity at the top, if at the bottom of the abstraction stack there exists Turing completeness, all layers of abstraction above may be considered computable.

Ultimately, Turing drew the conclusion that it would indeed be possible to devise a machine that could pass the imitation test. By his semantics, the brain was indeed computable.

Obviously, however, his prediction of this coming to pass by the year 2000 has not been made manifest.

The computer metaphor is thus an understandable one—it does have many appealing elements that offer a predictive framework for reasoning about a variety of non-computer systems such as the mind, but the reality that underlies that metaphor is far more richly textured that one might first realize. Indeed, I assert that software-intensive systems are among the most complex artefacts ever created.

4. SYSTEMS

Note that I use the term systems rather than programs. In my world, all economically interesting computing thingies are very much systems—they have a unique identity, they have discernible boundaries, they are composed of reducible constituent parts, they exhibit certain behaviours as a whole and—unless they are closed systems—they live as a society of systems with which they dance in context with their peers. By software-intensive, I mean to suggest that these thingies are manifest in the co-dependent and co-evolved artefacts of software and hardware. While hardware represents the physical form of a computing thingy, any such device would be no more powerful than a rock were it not for the software that directs its actions; at the same time, any chunk of software without a hardware platform on which it may execute would be a spirit in search of a body, neither alive nor dead but rather an unrealized entity. Hardware is the visible platform, the place where computation takes place; software is the collection of instructions, hidden from view, that animate that hardware and draw it to action.3

In brief, a Turing machine is one that supports conditional branching and the ability to perform state-changing operations. Early computers were often not Turing complete (for many lacked the ability to branch) but virtually all contemporary computers and programming languages are Turing complete. A large class of languages—regular, context-free and context-sensitive ones, in particular—are recursively enumerable and thus may be processed by a Turing machine. A decidable problem is thus one for which an algorithm (ultimately manifest in one of these classes of languages) will eventually terminate. The formal semantics of Turing machines (and the nature of decidability) are well understood and are well documented elsewhere. As a start, one might begin with the Stanford Encyclopedia of Philosophy’s entry on the subject, see http://plato.stanford.edu/entries/turing-machine.

By economically interesting, I mean that it presents some measurable value in some context; by thingie, I mean just that—a thing of some vague sort; I purposely chose this ambiguous word, so as to not suffer the consequences of any other overloaded word that carries historical or emotional baggage such as component or program or object. Even so, the word program is largely meaningless in the context of contemporary software-intensive systems.

Still, from a systems perspective, that software is a part of the system, and while we speak of software as a thing unto itself, any such software is ultimately made manifest in the physical world of that...
Stated in a more lyrical fashion, one might say that software is the invisible thread, and hardware is the loom on which we weave the artefacts of computing. 5 From the perspective of computability, whether something is made manifest in software or in hardware is irrelevant. As long as the programming language is Turing complete and as long as the hardware is Turing complete, all software-intensive systems constructed upon such a foundation are themselves Turing complete. Furthermore, it is possible to take a software artefact and move it to hardware and vice versa without impacting the completeness of the system as a whole.

The semantics of the term system is one that may exhibit dissonance when explained from the point of view of different domains, and so a more precise statement of its meaning here is in order.

In this context of this current investigation, a system has been defined as a collection of individuals in which some of the intrinsic properties of the collection and their joint results are not strictly deducible, in principle, from the properties of an spatio-temporal relations among the composing individuals. Some feature or features of the interrelations among the parts give rise to novel systemic properties and forms of causal efficacy.

In systems engineering, we use a subtly different definition. As adopted by the International Council on Systems Engineering, using the definition from Rechtin a system is a construct or collection of different elements that together produce results not obtainable by the elements alone. The elements, or parts, can include people, hardware, software, facilities, policies, and documents, that is, all things required to produce systems-level results. The results include system level qualities, properties, characteristics, functions, behaviours and performance. The value added by the system as a whole, beyond that contributed independently by the parts is primarily created by the relationship among the parts; that is, how they are interconnected (http://www.incose.org/practice/fellowconsensus.aspx).

I prefer the latter definition; whereas both emphasize the importance of relationships in the semantics of a system (the whole is greater than the sum of its part, because of the reification of the relationships), the latter is more general (it attends to other than individuals) and more comprehensive (especially in regard to the meaning of system level behaviours).

5. PEOPLEWARE AND BITS AT THE BOTTOM

I also use the terms artefact and created to acknowledge that humans directly or indirectly (though other computing things) make a computing thing. 6 Developing hardware is fully an engineering activity, involving today the creation of digital circuitry containing hundreds of millions of transistors, each transistor forged from a hundred or so atoms. Similarly, developing quality software thrives on the cusp of engineering and art, of mathematical formalisms and creative writing; it is not unusual to find tens of millions of lines of code in any given software-intensive system, each line of code originally crafted by some person. Furthermore, a quick back of the envelope calculation 7 suggests that, worldwide, some 33 billion source lines of code (SLOC) of new or modified software are produced yearly, giving approximately one trillion SLOC produced since the late 1940s (when high-order programming languages began to gain traction).

Collectively, these artefacts have changed the way we live, and it is at once surprising (that relatively so few people have laboured to produce these artefacts) and humbling (that these artefacts have woven themselves into the interstitial spaces of the world).

Software-intensive systems are created through an astonishingly labour-intensive activity. Most software-intensive systems of value go through the incremental and iterative cycle of development, deployment, operation and evolution, each phase of which transforms the system’s architecture. In that regard, the process of building software has certain parallels to the process of building things. Small software-intensive systems are like dog-houses: they do not require any blueprints and are largely disposable. Modest to large software-intensive systems are like houses or skyscrapers: they entail more cost and risk and therefore best practices demand more rigorous blueprints, better tools, significant testing, a risk-driven process and accountability. Ultra-large, long-lived software-intensive systems exhibit obduracy, and thus are closer to the problem of the organic growth of a city and the attendant activities of urban renewal [7].

We can and do construct software-intensive systems whose behaviour surprises us. For example, there have been automatic theorem-proving systems that have yielded novel proofs that no human had ever thought to derive; projects such as Massachusetts Institute of Technology’s (MIT) Kismet and Honda’s Asimov offer robots that are astonishingly engaging from a social perspective; even IBM’s Watson is able to make human-like leaps of ‘intuition’ when attending to the nuances and puns one finds in playing the game Jeopardy! Nonetheless, every software-intensive system—no matter how complex or how surprising its emergent behaviour—is built by humans; there is always an outside force that brought these artefacts into being. Software-intensive systems do not spring up whole from the primordial soup of bits. 8 Furthermore, every

6This is so even if one considers self-modifying programs or programs that create other programs. Ultimately, there is a genealogy that traces every program back to some software-intensive system that was created by some human at some point in time.

7It considers the number of software professionals worldwide, the percentage of those who actually cut code and the approximate number of source lines of code (SLOC) per person per year. The numbers I offer are likely very conservative and thus low.

8Robert Wilensky once noted ‘we’ve all heard that a million monkeys banging on a million typewriters will eventually reproduce the entire
software-intensive system is simply bits at the bottom; there is no magic, there is no *Deus ex Machina* necessary to explain the operation of a software-intensive system. If there is anything we may call magic, it arises only because the modular, hierarchical layers of abstraction formed upon those bits give us the illusion of simplicity (and which we name as magical because their mind and their matter are obscure). It may be that considerable complexity obscures the meaning or the operation of a software-intensive system, but ultimately, such Turing complete systems are also decidable.

It is important to note that this apparent complexity is imposed upon a software-intensive system from the outside, from the top-down. We may have a vision for a software-intensive system which we morph into more precise statements of requirements which in turn are made manifest in strings in our programming languages, which in turn are transformed (usually by other software-intensive systems) into some instance of the equivalence class we call software, as, for example, holes, or physical configurations, or charges. At each level in this chain from top to bottom, there is nothing that cannot be reduced to something simpler, something Turing complete.\(^9\)

Nonetheless, as another lyrical aside, there is a curious inverse relationship between the concerns of theoretical physics and those of software-intensive systems. In physics, one considers the fierce complexity of the universe and works to tease apart each string and make transparent the simple models from which all things are made. In the engineering of software-intensive systems, we take very simple things (Turing complete machines are wonderfully simple mathematical constructs), manufactured huge, tangled, dripping hairballs containing millions of lines of code and then unleash them into the wild, expecting that they will fade into the shadows, the best ones ending up completely invisible.

6. COMPLEXITY

A deeper examination of complexity is in order [8]. One may use classical system measures to name the complexity of the hardware components of computing thingsies, but measuring the complexity of its software components is less well codified.

As Brooks [9] has so eloquently noted, software-intensive systems embody an essential complexity, a complexity that is baked in to the very domain and is hence inescapable. Furthermore, software-intensive systems are discrete, and that brings along with at least four fundamental challenges that contribute to their complexity.

First, discrete systems may exhibit non-continuous behaviour. In an analogue world—the world in which we live—the laws of physics are continuous;\(^10\) there are no discontinuities. Thus, as one throws an apple into the air and watches it fall to the ground, classical physics can predict its parabolic path; we would not be surprised to watch its graceful arc. However, were we to throw a digital apple in a virtual world, no such laws of physics need intrude\(^11\) and thus it would be possible—and it sometimes happens—to see said virtual apple zig and zag or stop in mid air, only to rapidly shoot off in some random direction. Now, it may be that this is not the behaviour we desired or that we thought we programmed, but the discrete nature of the software-intensive system that gives rise to that virtual apple does not rule out such behaviour.

Second, as the size of a software-intensive system grows, so does its state space. Indeed, there is often a combinatorial explosion of states that rises in such systems. A program with one variable has a simple state space; a program with millions of variables has an enormous state space. Indeed, the state space of a given problem may be so large that any solution is intractable: there may be an algorithm that yields an answer, but it would require more computational elements than there are particles in the known universe or more seconds than there are in the life of the known universe to compute. As it turns out, it is not difficult to reach the point of intractability. In computer science, this is the domain of complexity theory, which has given rise to a class of problems called non-deterministic polynomial (NP) time. The travelling salesman problem is the classic example of an NP-hard problem: given a set of cities with measured paths among them, determine a route that is less than some specified distance. For a small number of cities and paths, a solution is possible, but as the state space grows, the solution quickly becomes intractable.

In that light, complexity [10] may be measured by consideration of the number of possible states that a problem might embody [11]. By that measure, at many levels of abstraction, a boulder is arguably simple, recognizing that any state changes it may undergo are few in number and spread across geological time. In contrast, even the most modest software-intensive system will embody an enormous number of possible states, and it is only by abstracting away a myriad of details that we may have any degree of confidence in the correctness of that system. Now, lest the formalists berate me, remember that I am also talking about the state of software-intensive systems in the real world, wherein the discrete, digital world collides with the harsh reality of a very continuous, very analogue real world. The fact that we may have to attend to a combinatorial explosion of possible states in the digital world is nothing compared with the context of possible states in the real world.

Speaking of computing in the real world, consider this third element: software-intensive systems may be corrupted by unexpected external events. In this regard,
we may think that our software-intensive systems are closed systems whose behaviour is entirely predictable, but, alas, the ugliness of the real world sometimes intrudes. A cosmic ray may flip a bit in our processor or memory; a hardware device might fail. While these are elements that any reasonable systems engineering process must take into consideration, especially where human lives are involved, the reality is that there is no pragmatic means of defending against all such gremlins in the system. Furthermore, building fault-tolerant mechanisms introduces further complexity into a system, such that even more points of failure spring into being and the state space is further expanded.

Finally, there is one other pragmatic challenge that limits our ability to attend to complexity: we lack the mathematical tools and intellectual capacity to model and reason about the behaviour of large discrete systems. Ultimately, there is little one can do to reduce the essential complexity of software-intensive systems; the best we can do is hope to manage that complexity. The primary mechanism we as humans attend to complexity is by abstracting; we see this reflected in our programming languages, our methodologies, our platforms. It is fair to say that the entire history of software engineering is one of the raising levels of abstraction. By abstracting, we are able to focus on the essence of a problem and platter over the implementation of that problem. Again, the notion of Turing completeness comes in play: each layer of abstraction is built upon another, and at the very bottom, we have a Turing complete machine. As such, as long as we maintain semantic integrity, each level is decidable.

It is these layers of abstraction, coupled with the notion of information hiding, that give rise to the organized complexity of a well-structured software-intensive system. Each such layer may be recursively layered (meaning that the modular hierarchy of such a system flows from the top of the system down to its non-divisible bits) and each such layer may embody its own decidable complexity (by building upon the semantics of lower levels, whose implementation is hidden from the levels above).

7. MEASURING COMPLEXITY

While we speak of a given system being more or less complex than another, there are—shockingly—few meaningful measures of complexity for large software-intensive systems. Classically, complexity may be measured in two ways: computational complexity and Kolmogorov complexity.

Computational complexity, in general, attends to the time and space needed to carry out a particular algorithm. Generally, one may name the best case, worst case and average times for execution, and this offers a measure of complexity. In sorting, for example, a bubble sort will run (at best) in \( n \) time (where \( n \) is the number of items to be sorted) and at worst \( n^2 \) time. On the other hand, the average case for the quicksort is on the order of \( n \log n \) (and the worst case is on the order of \( n^2 \)). A bubble sort therefore has a higher computational complexity than a quicksort, for it requires more resources.

While these sorts of complexity measures are useful from an operational sense, the very production of a software-intensive system introduces its own nature of complexity, and therein Kolmogorov complexity comes into play. Given an problem involving a simple input–output mapping (meaning, for every given input, there is a desired output that is a function of that input), it is reasonable to say that a million line program that solves the problem is more complex than a 10 line program. Kolmogorov complexity is thus more of a descriptive complexity—assuming that we are comparing implementations for the same problem space—wherein it is a measure of the size of the artefacts required to carry out such a mapping. If we think of a software-intensive system as just a string—and the Turing machine is in effect a reduction of any such system into such a string—then we consider the minimum string necessary to carry out this mapping. An implementation that requires a string of length \( x \) would be deemed simpler than an implementation that requires a string greater than length \( x \).

While theoretically interesting and relevant, calculating the computational complexity or the Kolmogorov complexity of anything but the simplest algorithm is impractical. How does one reduce the corpus of Facebook to a single string in order to compare its complexity against, for example, Adobe Photoshop? Elsewhere [12], I have posited that software architectural complexity may be measured in terms of mass,15 the enumeration of things, the enumeration of connections and the presence of patterns across views. By mass, I mean simply the number of lines of code in a solution. While this offers a gross level characterization of a system, it does suffer from a number of problems: what is a line of code? How does a line of code in one language compare with a line of code in another? Are there not multiple ways of varying efficiency—even within the same programming language—of expressing a solution using different lines of code? In short, measuring the complexity of a system by measuring its SLOC is a somewhat useful but easily misleading approach.

As we will see, however, measuring things, connections and patterns is a more promising approach, but we need to explain the large-scale structure of software-intensive systems first (and will do so shortly).

As it turns out, practical software-intensive systems are even messier than one might dream theoretically. Indeed, there are several triggers of complexity that flow from the nature of their implementation and the nature of their interaction with the real world.

All but the simplest software-intensive systems embody a large number of moving parts, each with significant degrees of freedom and a significant number

---

12This is the root cause of what Brooks means by essential complexity: it is essential in that we as humans have clear limits as to our ability to wrap our minds around complexity.

13Facebook consists of approximately 1 million lines of code, mostly written in PHP.

14Adobe Photoshop consists of over 10 million lines of code, mostly written in C and C++.

15As calculated in SLOC, though recognizing that software has no weight.
of potential interactions among those parts. While we strive to design programming languages and machines that make it difficult for a programmer to do violence to the software, reality is that any memory location in a system may, in theory, be visible to a process. This reality is what makes it possible for various species of computer viruses to wreak havoc among normally well-behaved systems. Thus, even if we find a problem of state space explosion in the problem space, then complexity is further triggered by the increased state space introduced by the elements of the solution.

Continuing, because software-intensive systems are built by humans, and because such economically interesting systems tend to evolve in fits and starts over time, it is difficult to develop a system that is perfectly textured. The lack of regularity within a system is yet another trigger of self-made complexity. For example, when simulating a weather system, calculating the forces of wind on a flexible structure such as a building, or simulating the explosion of a nuclear device, there is an embarrassing degree of parallelism in the real world that can be mapped directly to physical parallelism in the computing space. Finite element analysis, which underlies each of these three examples, involves the nearly independent calculation of millions upon millions of partial differential equations. While full of hairy mathematics, there is considerable regularity in such systems. In contrast, there are many classes of problems for which there is no such natural parallelism and hence no opportunity for such regularity. For example, the processing of airline seat reservations, the scheduling of parcels or people across multiple destinations, and the analysis of speech are all examples that involve tightly coupled data and algorithms that make it difficult to map perfectly to multiple processors; difficult, because there is far less semantic independence among processing elements.\textsuperscript{17}

The presence of non-holonomic constraints also introduces complexity. Basically, a non-holonomic system is one in which history plays a role: the journey that such a system takes impacts its current state. Non-holonomic systems are common in the real world; robotic motion control and path planning are common examples in the space of software-intensive systems.\textsuperscript{18}

Although not strictly non-holonomic, many interesting software-intensive systems are stateful, meaning that a thread of control may behave as a function not only of its inputs but also of its current state.\textsuperscript{19} A very simple example is that of a website: visiting a site once might generate a cookie on the client machine, such that revisiting the site offers up a subtly different page. By contrast, functional programming (FP) eschews state. The essence of FP is the creation of computations that evaluate mathematical functions that are stateless and immutable. Pragmatically, the class of problems that are best made manifest by FP is small, for it is easy to do hard things in FP but very hard to do many easy things.\textsuperscript{20}

8. ON SIMPLICITY

We have examined what complexity is; what does a simple software-intensive system look like? Building on the observations of Herbert Simon, my experience would posit that simple, quality software-intensive systems exhibit the following characteristics.

First, they are filled with crisp abstractions. By abstraction, I mean simply a simplification of reality. A software object that denotes a customer is not the customer itself; it is an abstraction. In software-intensive systems, one builds abstractions in layers, the bottom most layer being closest to the machine and the physical world, the highest being closest to the metaphor in the problems space. By crisp, I mean that such abstractions have clear boundaries and are conceptually nearly independent from others.\textsuperscript{21} In the history of computing, we have seen a shift from algorithmically oriented abstractions to object-oriented ones. Algorithmically oriented abstractions are a reflection of early programming languages (such as \textsc{Fortran} and \textsc{COBOL}) that were focused on the domain of mathematical and symbolic manipulations; object-oriented abstractions are a reflection of more contemporary languages (such as \textsc{C++} and \textsc{Java}) that are focused on more abstract manipulations. In algorithmic design, abstractions focus on functions, with data existing external/global to that function. In object-oriented design, abstractions combine data with the operations that are appropriate do that data.

Second, there is a clear separation of concerns among these abstractions. This separation may be vertical through the system, yielding the layers of abstractions described earlier. A major force that drives such layering is that first proposed by Parnas, namely information hiding. In such a strategy, the interface of an abstraction is exposed to peers and layers above, whereas the implementation of that interface is hidden. In this manner, there is clear separation between structure and behaviour and their manifestation. This separation may also be horizontal. At a given layer of abstraction, the state of other objects may impact a given thread of control, but these states would be considered inputs.

\textsuperscript{16}This is the problem of legacy software. From the moment a line of code is written and becomes a part of a software-intensive system, it becomes legacy code. Small systems can be thrown away; large systems have economic and intellectual inertia that resist such abandonment, and so additional complexity is introduced by the need to attend to the transformation of legacy. There are interesting parallels in the evolution of software-intensive systems and biological systems worth consideration, but this margin is too small to contain them.

\textsuperscript{18}In many such systems, it is possible to offer the illusion of single threads of control, while hiding the messy details of parallel threads of execution below the surface; concurrency adds considerable complexity to the task of programming, and the average program is not an expert in multi-processing. In many such systems, this illusion may be easily maintained, for loose or lazy synchronization is possible. However, as we see the end of the frequency wars, wherein computational power was largely a matter of increasing processor clock speed, there has been a shift to multi-core processors. Such intimate concurrence is hard to abstract away, and so more developers are having to become skilled in concurrent algorithms.

\textsuperscript{19}Recording the history of a system is a state-changing operation, typically manifest by changing the values of variables hidden from the outside.

\textsuperscript{20}This is the problem of legacy software. From the moment a line of code is written and becomes a part of a software-intensive system, it becomes legacy code. Small systems can be thrown away; large systems have economic and intellectual inertia that resist such abandonment, and so additional complexity is introduced by the need to attend to the transformation of legacy. There are interesting parallels in the evolution of software-intensive systems and biological systems worth consideration, but this margin is too small to contain them.

\textsuperscript{21}In many such systems, it is possible to offer the illusion of single threads of control, while hiding the messy details of parallel threads of execution below the surface; concurrency adds considerable complexity to the task of programming, and the average program is not an expert in multi-processing. In many such systems, this illusion may be easily maintained, for loose or lazy synchronization is possible. However, as we see the end of the frequency wars, wherein computational power was largely a matter of increasing processor clock speed, there has been a shift to multi-core processors. Such intimate concurrence is hard to abstract away, and so more developers are having to become skilled in concurrent algorithms.
semantically distinct concepts will be manifest as conceptually or physically distinct abstractions. This separation is induced by the strategy of building crisp abstractions.

Third, there is a balanced distribution of responsibilities among these abstractions. By responsibility, I mean the behaviour associated with a given abstraction; by balanced, I mean that the behaviour of a system is crisply divided among different abstractions, so that no one abstraction does more/is significantly weightier than others. For example, in user interfaces, it is a best practice to have three distinct abstractions, one representing the thing being modelled, another representing the projection of a view of that thing and the third responsible for controlling the flow of information and the flow of control among a model and its views.

Fourth, such systems are filled with patterns. The example above (of user interfaces) is an example of such a pattern, called model–view–controller. Naively, one may view a software-intensive system as just a bundle of bits in layers of abstraction. However, in practice, the most well-structured systems are made simple by the presence of these kinds of patterns, which name repeatable structures that are woven through a system. Implementing such patterns often involved scattering and tangling. With scattering, a pattern is not manifest as a single abstraction, but rather involves a society of abstractions that work collaboratively to carry out that pattern. With tangling, a given abstraction may participate in many patterns, and thus if we look at its interface (or its implementation), we may trace some elements that correspond directly to a given pattern but that most often exist to support multiple patterns simultaneously.

Weaving patterns into a complex software-intensive system is a significant element of conceptual simplicity that transcends the physical complexity because patterns represent element of reuse at a relatively high level of abstraction.

9. THE LARGE-SCALE STRUCTURE OF SOFTWARE-INTENSIVE SYSTEMS

A software-intensive system may be considered Turing complete; it manifests no irreducible complexity; it is potentially time-invertible (assuming that the information entropy of the system is not increased owing to external factors, such as by deletion or the introduction of a random variable) [13]; it may even be introspective, reflective and self-healing.

If we determine something to be computable (say, for example, the brain), then theoretically it could be manifest as a software-intensive system. The fact that a software-intensive system has no irreducible complexity leads to the implication that even if we execute something as (formally) unpredictable as a neural net or a genetic algorithm on a Turing complete platform, no Deus ex machina is needed to explain its behaviour. Logical reversibility means that a software-intensive system may, functionally, ignore the arrow of time. Reflection means that a software-intensive system may aware of itself, through meta-agents (themselves being software-intensive) that observe the system, and thus permit the system to ‘know thyself’ and change and evolve.

A naive view of computing often focuses on the circuits from which hardware is formed, structured at growing levels of hierarchical abstractions (and their connections) from transistors to gates to circuits to chips to subsystems and so on. A slightly more advanced yet equally simplistic view of computing focuses on the expressions of the digital domain from which software rises, also found structured in layers of abstraction.

My expertise is in software-intensive systems; so I will defer a discussion of the parallel considerations of hardware: ontological nature of computing hardware is already well understood [14].

At the lowest level of abstraction in software—concerning myself first to data—we find bits, which may be chunked into words, and those into primitive data structures, then those into composite ones and so on. When we project a suitable symbolic meaning on them—their semantics—then we begin to approach the realm of information. Now, while data forms the nouns of our systems, we have also the verbs: expressions formed into statements leading to control structures composing functions leading to components and then on to subsystems and finally systems themselves.

As we introduced earlier, for much of the history of computing, data and processes—nouns and verbs—were treated as relatively distinct. However, as the essential complexity of software-intensive systems grew, the the contemporary algorithmic-oriented languages (such as FORTRAN) and methods (such as structured analysis and design techniques) grew tired, and were replaced over time by object-oriented ones (such as Java and the notation and processes associated with the unified modelling language, http://www.uml.org). Most contemporary software-intensive systems are object-oriented in some nature, and so at the lowest level of abstraction we have, not surprisingly, the object. An object has identity, state and behaviour, and thus represents the fusion of data and action. This is also a wonderfully recursive concept: objects

22There is a vibrant sub community of software engineering regarding the language of patterns, as found at http://www.hillside.net and as manifest in many books such as Gamma’s Design Patterns. This work was first inspired by the ideas of the architectural Christopher Alexander in The Timeless Way of Building (and was first promoted by Kent Beck and Ward Cunningham, the latter who invented the first Wiki from which Wikipedia emerged). The software patterns community has devised hundreds of design patterns, which serve as mechanisms for naming the structure and behaviour of societies of abstractions, patterns such as Singleton, Abstract Factory, Adapter, Facade, Observer and Command.

23One of the open considerations of the Church–Turing thesis.

24Today, no software-intensive system can be said pass the Turing test, although we continue to advance tantalizing close.

25To be clear, this is not so for the general case, but only for the specific case where sufficient metainformation is preserved so as to be able to unwind/recalculate the chain of events that led to all state-changing operations upon the system.

26This is true of so-called adaptive systems and of systems for which machine learning is an element. For example, most modern routers are able to change their behaviour based on the real-time analysis of traffic patterns; similarly, IBM’s Watson is able to learn during the conduct of a game, by ‘listening’ to the answers of its competitors and adjusting its understanding of context and lexical analysis.

27For that matter, one may find a discussion of a similar duality in the work of the Greek philosopher Lucippus and his student Democritus.
may be composed of objects may be composed of objects
and so on. No object is an island, however, and thus
there are rich relationships that abound: associations,
whole/part structures and inheritance, to name the three
most important.

And yet, there is more than just these objects and
relationships to be found in a well-tempered software-
intensive system. Three in particular come to mind as
particularly germane.

First, there are communicating sequential processes
[15]. The most basic software-intensive systems are
fully sequential, having a single line of control that
weaves through it; all others tend to have multiple
lines of control, which are the warp and the woof in
the fabric of the data and processing structures of
such a system. This sort of parallelism may be manifest
at many levels, ranging from individual software agents
to the multi-core chip to server farms to the entire com-
puting cloud.28 Be it just a few processes and threads
or many, such concurrency introduces tremendous com-
plexity associated with critical regions, synchronization,
race conditions, and livelock or deadlock.

Second, as we explained earlier relative to simplicity,
there are the issues of patterns (http://www.hillside.
.net). A pattern denotes a common solution to a common
problem, and most often is manifested as a named society
of objects that collaborate in certain ways. For example,
the Iterator pattern offers a mechanism for traversing
the parts of any complex object independent of the rep-
resentation of that object. Design patterns in particular
are part of the texture—be it accidental or intentional—
of every well-tempered software-intensive system [16] as
well as of beautiful physical structures [17].

Finally, there are issues of crosscutting concerns,
commonly called aspects. Aspects are akin to patterns,
in that aspects may be manifest as a pattern; aspects
are subtly different than just patterns, for they rep-
resent the abstraction of structure or behaviour as
seen from the perspective of specific sets of concerns.

There are a multitude of consequences of the presence
of this ontology of software-intensive system elements,
but three in particular are useful in this context.

First, most such systems have no top. One may ident-
ify certain places of interest, multi-dimensional cusps
in the fabric of that system, but there is often no real top,
no starting place. Even to the user of a personal comput-
er, the notion of a top is an illusion that arises from
the relationship of that stakeholder (the user) and the
computer (the object that appears to have a top). The
user may see the on/off switch, a command line or a
start menu as the ‘top’ of the system, but that is only
a conceptual convenience useful to the human; the
software-intensive system itself sees any of these inputs
as just one message that flows into the system from
some port, a port that is distinctly not at any ‘top’ that
is so named by the computing system.

Second, interesting software-intensive systems are
still input—output mappings, but not in the pragmatic
sense as we might envision from a theoretical point of
view. In the real world, any interesting software-intensive

system $S$ may ingest information $I$ from multiple sources
simultaneously or in some partial ordering of time,
metabolize $I$ and then yield $I'$, along with a potential
state change in $S$. While this at one level of abstraction
is an input—output mapping, it is a bit messier than
that: $I$ may be semantically and physically very large,
the state space of $S$ may be enormous to the point of
intractability, and the very exportation of $I'$ may
change the state of the real world. In other words, we
can make some interesting theoretical conjectures
about the behaviour of a software-intensive system, but
such conjectures are significant abstractions of what
goes on in the system itself.

Third, to understand, to reason about, to visualize
any such system that lies above a certain level of
complexity, one must use multiple points of view.29

The importance of views is a relatively recent yet
very significant advancement in our understanding of
software-intensive systems from the perspective of its
architecture. As defined by International Standards
Organization (ISO) 42010, ‘architecture is the funda-
mental conception of a system and its environment
embodied in elements, their relationship to each other
and to the environment, and principles guiding system
design and evolution’. In that sense, as 42 010 descri-
ies it, architecture serves as a naming of the essence
of a system and serves as a blueprint, as literature,
as a language of expression, as a set of significant
design decisions.

As the standard goes on to note, every system has an
architecture,30 and every architecture may be rep-
resented by one or more descriptions.31 Furthermore,
every such description is best formed from different
views, each view representing a set of concerns from
the perspective of a given set of stakeholders.

The ISO standard does not specify what views should
be used to describe a software-intensive system, but my
experience is that Philippe Kruchten’s 4 + 1 View
Model [18] provides a necessary and sufficient set of
views for virtually all classes of software-intensive sys-
tems. Kruchten’s views include (http://en.wikipedia.
org/wiki/4%2B1_Architectural_View_Model):

- use case view (highlighting sequences of interactions
  between the system and its context);
- logical view (concerned with end user functionality);
- process view (focused on the needs of system inte-
gration and the characteristics of performance,
scalability and throughput);
- implementation view (concerned with the needs
  of programmers and the issue of configuration
  management); and

29Recommended Practice for Architectural Description of Software-

30In my experience, every system has an architecture; most such
architectures are accidental, born of the hundreds of thousands of
decisions made by its developers, while a few architectures are
intentional. For any interesting software-intensive system, an
architecture will evolve over time, sometimes collapsing, sometimes
transforming.

31The fact that there may be multiple possible implementations at
each level of abstraction is, as Ellis has noted, ‘a sign that top-down
causation is in effect’.

[28]The Datacenter as a Computer: an introduction to the Design of

Interface Focus
— deployment view (focused on the concerns of systems engineering, with attention paid to system topology, communication and provisioning).

In my experience (http://www.handbookofsoftwarearchitecture.com/index.jsp?page=Main), each such view is best described by enumerating the key abstractions and the key mechanisms present, using the language of patterns to express all crosscutting abstractions. At the system level, my experience also shows that a system’s architecture can be partially described by an enumeration of the significant design decisions that may cross view boundaries. While all architecture is design, not all design is architecture, and so here we may have the naming of the most significant decision, each of which represents some load-bearing wall in the design of the system as a whole.

10. THE LIMITS OF SOFTWARE-INTENSIVE SYSTEMS

Software-intensive systems allow us to form new worlds (http://secondlife.com) and yet, in any universe, such systems must still be constrained by the laws of that universe. Indeed, while the human imagination may be unlimited, there are limits to software-intensive systems, from the fundamental (information may not be transmitted faster than the speed of light; a computation must preserve the second law of thermodynamics) to the theoretical (for example, the problem of NP-complete class complexity) to the semiotic (as Terrence Deacon observes, a symbolic system must be sufficiently rich so as to permit the creation of complex sentences, but also as Godel notes, any such complex system will give rise to sentences that are true but that cannot be proved so in that symbolic system) to the organic (how does one best architect a system? How does one best architect the organization that architects that system?) to the social (economic, moral and ethical limits are often forces that weigh upon such systems).

11. THE QUESTIONS

What we hope to have achieved is to make it clear that what Hawking might have meant by computer is a fiercely complex concept. While by no means judging his semantics—we have had no conversations with him on the matter—to say that anything is ‘just’ a computer glosses over the elegance of this metaphor.

So, in light of that metaphor, let us attend to the questions we introduced at the start of this discourse.

Is the self emergent, or is the self a product of external causes?

Strictly from the perspective of the computer metaphor, there can be no emergence save for the initiation by some external cause. Even if we consider systems of systems, we must always accept that such systems were constructed by humans and thus their ‘laws of physics’ in which they were conceived and operate were ultimately external causes.

There do exist self-organizing software-intensive systems. Research into swarms of robots has yielded some astonishing, emergent behaviour. Genetic algorithms and neural nets have been applied to attend to problems that are otherwise intractable. And yet, from perspective of a computing system, we may be surprised by the behaviour of a system, we may name some behaviour as emergent because we did not program it directly, but nonetheless there is no irreducible complexity in a software-intensive system that makes the operation of a Turing complete system undecidable.

As to the nature of the meaning of self in the context of a software-intensive system, it does beg the question of the semantics of self. For our purposes, let us suggest that self is a property of reflection, in which an entity—within the system of the entity itself—possesses a state that represents itself. A further analysis of this definition is far beyond the scope of this present discourse, but I shall posit that, under that definition, yes, a software-intensive system may embody a sense of self. Introspective (and self-healing) systems are not uncommon, and as Minsky [19] has described, the illusion of self may simply be an emergent property rising from a society of nearly independent agents. While such a conclusion may be surprising, it is, in my framing of the issue, simply a tangled loop, as Hofstadter [20] describes it.

Can any significantly complex system exhibit interesting behaviour without some element of externally applied top-down causes?

(Hofstadter 1999, pp. 709–710)

Again, from the strict perspective of software-intensive systems, the answer is yes—and no. Yes, in the sense that, at the limit, no software-intensive system is truly self-starting; no, in the sense that one must question what top-down means. If we interpret the semantics of top-down to mean simply external causes, then again we must answer yes; if we truly require there to be a top, then we must abstain, for we cannot offer any meaningful semantics to the concept of top within a software-intensive system except insofar as one layer may build upon a lower one.

Indeed, is there even such a thing as a closed system?

From a theoretical sense, perhaps yes; we can posit such a system for the sake of understanding and reasoning about said system. From a pragmatic sense, we can also assume that our software-intensive system is closed: it exists in a particular well-defined context and it behaves thusly. However, in both cases, we know that these are just abstractions necessary for us

32Where significant is measured by the cost of change. Software is a most fungible medium, and even though software has no mass, it does have weight, so to speak, and ‘moving’ software has very real costs.

33We believe.

34True or false depending on one’s stand regarding apparent faster-than-light communication.

35It would be a mistake to extend this argument to support the domain of intelligent design.

36Therein, however, lie some wickedly fascinating issues. In particular, there are some problems—the halting program is the classic one—that can be stated but yet cannot be decided by a Turing machine. Godel’s theorem comes in play here, as we have an instance of a sufficiently complex system—i.e. software-intensive systems—wherein truths may be stated yet not provable within the system itself.

37This presumes that there are no cycles in the layers of abstraction of a system, an assumption that, as noted by Simon, appears to be true for all classes of complex systems.
to wrap our human minds around a vastly more complex reality. Software-intensive systems are never truly closed: they consume energy; they are impacted by forces outside the context that we may imagine exist. Indeed, the real world has a nasty habit of intruding on our illusion of closed and context.

12. ON THE COMPUTABILITY OF THE MIND

Take all the squishy bits of a human, a cat or a plant. Physically and chemically and biologically you will find astonishing similarities at a given level of abstraction, and yet there are fundamental, subtle differences at the bottom. Now, the current Wikipedia, in all of its languages, consists of over 3500 million symbols, a healthy 3.5 Gb of information. As late as the 1960s, the collective capacity of every computer in the world together fell far short of a single gigabyte; today, I can carry far more information around in my hand, on my phone. By comparison, Ralph Merkel (http://www.merkle.com) has estimated that the storage capacity of the human brain is a strikingly modest value, numbering perhaps only a few hundred megabytes of information. Similarly, the Power7 chip, which lies at the heart of IBM’s line of mainframe computers, can process over 264 gigaflops, or approximately $10^{30}$ floating-point operations per second. Much of this performance can be attributed to astonishingly fast circuitry; processing multiple streams of data in parallel contributes to this high performance as well.

Why, then, is there such a gulf between what we know of computability and what it is to live as a sentient being?38

REFERENCES

1 Lakoff, G. 1987 Women, fire, and dangerous things. Chicago, IL: University of Chicago Press.
18 Kruchten, P. 1995 Architectural blueprints: the 4 + 1 view model of software architecture. IEEE Softw. 12, 42–50. (doi:10.1109/52.469759)

38 The metaphor of Flatland by Edwin Abbott also comes to mind. As an agent outside of a software-intensive system, I have the power of life and death (as long as I have my hand on the power switch), I can see from beginning to the end (for I see the whole and am its context), I create and control its actions (a system may offer the illusion of freedom, but a deterministic system is, well, deterministic). Were a software object sentient, I would appear as a god to it. But of course that I know I am not.