RC25032 (W1008-013) August 5, 2010
Computer Science

IBM Research Report

On the Ontological Nature of Software-Intensive Systems

Grady Booch
IBM Research Division
Boulder, CO
gbooch@us.ibm.com

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

On The Ontological Nature of Software-Intensive Systems

Grady Booch
IBM Fellow
February 2010

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 analyzers 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'd learn much about macroeconomics.

The computer metaphor is 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 artifacts ever created.

First, I use the term *systems* rather than programs. In my world, all economically-interesting computing thingies[2] are very much systems – they have a unique identity, they have discernible boundaries, they are composed of reducible constituent parts, they exhibit certain behaviors 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 artifacts 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 animates that hardware and

[1] Dr. George Ellis, letter to author, June 21, 2009.
[2] 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.

draws it to action.

Second, I use the terms *artifact* and *created* to acknowledge that humans directly or indirectly (though other computing thingies) make a computing thingy. Developing hardware is fully an engineering activity, today involving the creation of digital circuitry containing hundreds of millions of transistors, each transistor forged from a hundred or so atoms. Similarly, developing software is an incredibly labor-intensive activity that works on the cusp of engineering and art, of mathematical formalisms and creative writing; it's 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[3] 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 1940's (when high order programming languages began to gain traction). Collectively, these artifacts have changed the way we live, and it is at once surprising (that relatively so few people have labored to produce these artifacts) and humbling (that these artifacts have woven themselves into the interstitial spaces of the world).

Third, there is the matter of *complexity*[4]. One may use classical system measures to name the complexity of the hardware components of computing thingies, but measuring the complexity of its software components is less well codified. Elsewhere[5] I have posited that software architectural complexity may be measured in terms of mass[6], the enumeration of things, the enumeration of connections, and the presence of patterns across views. Complexity[7] may also be measured by consideration of the number of possible states that a system might embody[8]. By that measure, at many levels of abstraction, a boulder is arguably simple, recognizing that any state changes it may undergo are few in nature 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'm 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 analog real world. The fact that we may have to attend to a combinatorial explosion of possible states in the digital world is nothing compared to the context of possible states in the real world. Fred

---

[3] Which considers the number of software professionals world-wide, the percentage of those who actually cut code, and the approximate number of source lines of code per person year. The numbers I offer are likely very conservative and thus low.

[4] See also Simon, H. "The Architecture of Complexity." *Proceedings of the American Philosphical Society,* vol. 106 (6), 1962.

[5] Booch, G. "Measuring Architectural Complexity." *IEEE Software*, vol. 25 (4), 2008.

[6] As calculated in SLOC, though recognizing that software has no weight.

[7] Booch, G. "The Defenestration of Superfluous Architectural Accoutrements." *IEEE Software*, vol. 26 (4), 2009.

[8] Sessions, R. *Simple Architectures for Complex Enterprises*, Microsoft Press, 2008, p. 58.

Brooks[9] speaks of this as an element of the essential (and inescapable) complexity of software-intensive systems.

A software-intensive system may be considered Turing complete; it manifests no irreducible complexity; it is potentially time-invertible[10]; it may even be introspective, reflective and self-healing.

If we determine something to be computable (say, for example, the brain[11]), then theoretically it could be manifest as a software-intensive system[12]. 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 behavior. 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.

Software-intensive systems do not spring up whole from the primordial soup of bits. Rather, they are created through an astonishingly labor-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 doghouses: they don't 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[13].

As an 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), manufacture huge, tangled, dripping hairballs containing of millions of lines of

---

[9] Brooks, F. *The Mythical Man-Month*, Addison-Wesley, 1995.
[10] Landauer, R. "Irreversibility and Heat Generation in the Computing Process," *IBM Journal of Research and Development*, vol. 5, pp. 183-191, 1961.
[11] One of the open considerations of the Church-Turing thesis.
[12] Today, no software-intensive system can be said pass the Turing test, although we continue to advance tantalizing close.
[13] Hommels, A., *Unbuilding Cities*. MIT Press, 2005.

code, then unleash them into the wild, expecting that they will fade into the shadows, the best ones ending up completely invisible.

A naïve 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.

But this is only a very small part of the story.

My expertise is in software-intensive systems, so I will table for this discussion the already reasonably understood ontological nature of hardware, and instead focus on software thingies. 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.

For much of the history of computing[14], data and processes – nouns and verbs – were treated as relatively distinct. However, as the essential complexity of software-intensive systems grew, the then contemporary algorithmic-oriented languages (such as Fortran) and methods (such as structured analysis and design techniques) grew tired, and were over time replaced by object-oriented ones (such as Java and the notation and processes associated with the Unified Modeling Language[15]). 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 identify, state, and behavior, and thus represents the fusion of data and action. This also is a wonderfully recursive concept: objects 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.

---

[14] For that matter, one may find a discussion of a similar duality in the work of the Greek philosopher Lucippus and his student Democritus.

[15] http://www.uml.org

First, there are communicating sequential processes[16]. 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 multicore chip to server farms to the entire computing cloud[17]. Be it just a few processes and threads or many, such concurrency introduces tremendous complexity associated with critical regions, synchronization, race conditions, and livelock or deadlock.

Second, there are the issues of patterns[18]. A pattern denotes a common solution to a common problem, and most often is manifest as a named society of objects that collaborate in certain ways. Design patterns in particular are part of the texture – be it accidental or intentional – of every well-tempered software-intensive system[19] as well as of beautiful physical structures[20].

Finally, there are issues of crosscutting concerns, commonly called aspects. In richly textured software-intensive systems, one will typically find scattering (one aspect being manifest in multiple places) and tangling (a given software thingy contributing to the manifestation of more than one aspect). This is an area of considerable challenging research.

There are a multitude of consequences of the presence of this classification of software-intensive system elements, but two in particular are useful in this context. First, most such systems have no top. One may identify certain places of interest, multidimensional cusps in the fabric of that system, but there is often no real top, no starting place. Second, 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[21].

Software-intensive systems allow us to form new worlds[22] and yet, in any universe, such systems must still constrained by the laws of that universe[23]. 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[24]; a computation must preserve the second law of thermodynamics) to the theoretical (for example, the problem of

---

[16] Hoare, C. "Communicating Sequential Proceses." *Communications of the ACM,* vol. 21 (8), 1978.
[17] *The Datacenter as a Computer: An introduction to the Design of Warehouse-Scale Machines.* Google, 2010.
[18] http://www.hillside.net
[19] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*, Addison-Wesley, 1994.
[20] Alexander, C. *The Timeless Way of Building*, Oxford University Press, 1979.
[21] *Recommended Practice for Architectural Description of Software-Intensive Systems*, ISO/IEC Standard 42010:2007.
[22] http://secondlife.com
[23] We believe.
[24] True or false depending on one's stand regarding apparent faster-than-light communication.

NP-complete class complexity) 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).

One parting observation.

Take all the squishy bits of a human, or 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 3,500 million symbols, a healthy 3.5 gigabytes 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[25] 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 gigiflops, 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. Again, by comparison, Merkel estimates that the human brain – a very parallel computer indeed - has the ability to process only about 10^{13} to 10^{16} instructions per second.

Why, then, is there such a gulf between what we know of computability and what it is to live as a sentient being.[26]?

[25] http://www.merkle.com

[26] The metaphor of *Flatland* by Edwin Abbot 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.