2

Philosophical Context

The first question asked, more often than not, when embarking upon a new development project is, "what language will be used for implementation?"

This is very unfortunate. Whatever the answer, it is almost always made for the wrong reasons. Even worse, it is made for reasons that are never articulated and therefore never subject to reasoned judgment.

Common reasons (not necessarily expressed out loud) for adopting a programming language include: (in no particular order)

- Loyalty, "we are a Microsoft shop, we use Visual Basic (or, today, C#)."
- Bandwagon, "everyone is doing Java."
- Economics, "Java programmers are a dime-a-dozen and completely interchangeable – we lose one we can find a replacement easily."

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

- Culture, "you can't do telecom / real-time / embedded applications in anything except C++."
- Resume, "all the job ads ask for C# experience, so I had better get some exposure."
- Inertia, "I wrote my first program in COBOL, you can do anything you want in COBOL, so COBOL is the right language for this project. And, it makes it easier for me to manage."

Given that all programming languages are ultimately equivalent ("anything you can do, I can do also"), are there any legitimate reasons for selecting one language over another? I would suggest that there are two reasons, one major and one minor.

The major reason for choosing an implementation language is the degree to which the philosophical assumptions, and the expressed intentions of the language designers for their language, are consistent with the way the development team thinks and the problems they are thinking about. If you think in formulas then FORTRAN is your language. If reports and accounts dominate your thoughts, COBOL can't be beat. Think like a machine, Assembler or C will allow you to directly express your thoughts. And if you think like an object? I will defer an answer for now.

davidwest 7/30/2003 12:10 PM
Deleted: An important

davidwest 7/30/2003 12:10 PM
Deleted: criterion

davidwest 7/30/2003 12:10 PM
Deleted: : are the

davidwest 7/30/2003 1:40 PM
Deleted: ?

davidwest 7/30/2003 4:27 PM
Deleted: If the answer is yes, then the language is likely to be more naturally expressive and there will be less need to contort language expressions so that they implement the design ideas.

The minor reason: if, and only if, performance mandates cannot be satisfied with effective design, it may be appropriate to select a language that provides more direct access to and control of hardware.

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

Developers, especially programmers, may be quite surprised at the assertion that philosophy plays the most critical role in language selection. They may be surprised that philosophy played any part in the design of programming languages in the first place. Observation of the debates about language that have occurred the past thirty years, and especially the debates about which languages are or are not object oriented, makes it clear that something other than syntax and benchmark results accounts for the fervor and acrimony exhibited by the debaters. This point can be illustrated with a short foray into the history of three languages of particular interest to object thinkers: Simula, C++ and Smalltalk.

davidwest 7/30/2003 4:33 PM
Deleted: Nevertheless, it is philosophy that explains the acrimony of debates over language and, almost always, the ultimate selection of a language by project teams or organizations.

davidwest 7/30/2003 4:35 PM
Deleted: .

davidwest 7/30/2003 4:37 PM
Deleted: [1]

Philosophy Made Manifest – Dueling Languages

All of the quotations regarding Simula, Smalltalk, and C++ languages in this section are taken from the respective chapters in *ACM History of Programming Languages*, ACM Press, 1978, Richard Wexelblat (ed.) or *History of Programming Languages II*, Addison-Wesley,1996, Thomas J. Bergin Jr. (ed.).

Computers – actually the engineering behind the construction of various hardware devices – and mathematics are the binary stars around which the world of computer science has revolved for more than fifty years. Programming once was a matter of re-wiring. It was a decade (circa 1960) before machine level problems were solved to an extent that developers could turn their attention to larger issues of program design. Another decade (crica 1970) elapsed before efforts to formalize (structured programming, structured design, structured analysis) the development of applications and systems were

davidwest 7/30/2003 4:38 PM
Deleted: we

davidwest 7/30/2003 4:39 PM
Deleted: our

davidwest 7/30/2003 4:39 PM
Deleted: we worked on

davidwest 7/30/2003 4:40 PM
Deleted: analysis

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

widely deployed. Significant attention is now being paid to the "soft" issues of usability, human-machine interaction, and even culture.

davidwest 7/30/2003 4:40 PM
Deleted: employing computers and programs

Despite these advances, whenever a new idea is introduced in computer science it receives little attention until it has been made concrete and explicit – frequently as a new programming language. Debates about ideas are then transformed into debates of the relative merits of the artifact languages. Arguments about design, analysis, method and process inevitably follow, but those discussions are more often grounded in the programming language artifacts than in the original ideas. In this process, the original ideas become secondary if they are not lost entirely.

This has certainly been true in the case of the "object idea" and the programming languages that lay claim to that idea. The most vituperative debates in the object community tended to center on questions of programming language. The overt focus of those debates was on programming language features and technical benchmarks. But it was the emotions induced by covert (not consciously hidden, merely assumed and unspoken) philosophical positions that accounted for the intensity and the hostility evident in those debates.

Behind the Quotes

Alan Kay, Kristen Nygaard, and Bjarne Stroustrup

Each of these individuals is known for designing and promoting a particular programming language (which we will be discussing shortly).

davidwest 7/30/2003 4:41 PM
Deleted: Each individual is also known for many other contributions to the world of software development and computer science.

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

Alan Kay designed, and Dan Ingalls implemented, the Smalltalk programming language while working at the Xerox sponsored Palo Alto Research Center (PARC), during the period when PARC was a premier center of exciting and important innovation in the field of computing . After leaving PARC, they both worked at Apple Computer where they reinvented Smalltalk and called it Squeak. Squeak underwent further development when both men moved to Walt Disney Corporation. Today, they are independent developers focused on educational applications of the Squeak language and its multimedia capabilities.[2]

Dr. Kay (he received his Ph.D. in computer science from the University of Utah) developed the idea of a "Dynabook" – a laptop, network enabled, computer with the Smalltalk environment – while at PARC. Arguably, the Pad Computer version of Microsoft's Windows operating system would be – if coupled with Smalltalk / Squeak – a pretty close approximation of his ideas.

Kay's biggest contribution was changing the way industry and users think of computers. Before Kay, computers were impersonal tyrants that required you to speak their language and limited their interaction to text-based communications. Kay personalized the machine expanded the means of interaction to include graphics, point-and-click, and multimedia using a language designed for human communication. The PC on your desktop reflects more of Kay's ideas than most people realize.

davidwest 7/30/2003 4:57 PM
Deleted: s rather than machines

Kristen Nygaard developed, with Ole Johan-Dahl, the Simula programming language - introducing the concepts upon which all later object-oriented programming languages are built: Objects, classes, inheritance, virtual quantities and multi-threaded program execution. He was also a social

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

activist greatly concerned with the use of information technology and spent a good part of his career concerned with the social impact of computer technology. His work became the foundation of what today is called "the Scandinavian School in System Development", closely linked to the field of Participatory Design – design by and on behalf of real users. XP and its practice of having an on-site customer is reflective participatory design.

davidwest 7/30/2003 4:58 PM
Deleted: .

He has received the Norbert Weiner prize from Computer Professionals for Social Responsibility (CPSR) for responsibility in social and professional work; the Rosing Prize, awarded by the Norwegian Data Association for exceptional professional achievements; the John von Neumann Medal by IEEE (Institute of Electrical and Electronic Engineers); the A. M. Turing Award by the ACM (Association of Computing Machinery) for 2001; and, in August 2000 he was made Commander of the Order of Saint Olav by the King of Norway.

Bjarne Stroustrup designed and implemented C++ while at A.T. & T. where problems of high speed switching systems for large scale networks provided the context for this thinking about programming and the values that should be incorporated into a programming language. He pioneered the use of object-oriented and generic programming techniques in application areas where efficiency is a premium, e.g. switching systems, simulation, graphics, embedded systems, and scientific computation. *The C++ Programming Language* (Addison-Wesley, 1985, 1991, 1997, and "special" edition in 2000) has been translated into 14 languages.

davidwest 7/30/2003 5:01 PM
Deleted: A later book, In addition to his five books, Stroustrup has published more than sixty academic and more popular papers

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

Dr. Stroustrup (Ph. D. computer science from Cambridge University) is an ACM Fellow and has received that society's Grace Murray Hopper award for his work in C++. He is an ATT Bell Labs Research Fellow and an ATT Fellow and currently holds the position of Professor at Texas A&M University.

Until the appearance of Java and C#, Smalltalk and C++ were the prime contenders for the hearts and minds of object developers. The intensity of argument between advocates of each language is legendary. Both sides tended to see Java as an interloper and tended to criticize those aspects of Java that reflect their more traditional nemesis. Disagreement between Smalltalkers and C++ers gains added interest from the fact that both claim to be the direct heirs of another, older, language, Simula.

Simula

> *Simula did not start as a programming language, and during all of its development stages, reasoning outside traditional programming played an important part in its design.*
>
> *From the very outset Simula was regarded as a* ***system description language****... [emphasis Nygaard's]*

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

Simula 67, however, was as a general purpose programming language. Was the original purpose abandoned or did Simula have a "dual personality?" The latter option is correct. This will be important when we examine what Smalltalk and C++ chose to borrow from Simula.

The concept of Simula began with an analysis of operations research and the kind of complex systems being modeled and analyzed in that domain. The first goal was to develop a "useful and consistent set of concepts" for modeling the "structure and interaction" of elements in complex systems. The initial objectives for the language were:

> *1. The language should be built around a general mathematical structure with few basic concepts. This structure should furnish the operations research worker with a standardized approach in his description so that he can easily define and describe the various components of the system in terms of these concepts.*
>
> *2. It should be unifying, pointing out similarities and differences between various kinds of network systems.*
>
> *3. It should be directing, and force the operations research worker to consider all aspects of the network.*
>
> *4. It should be general and allow the description of very wide classes of network systems and other systems which may be analyzed by simulation, and should for this purpose contain as a general algebraic dynamic language, as for example ALGOL and FORTRAN.*

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

> *5. It should be easy to read and to print, and should be suitable for communication between scientist's studying networks*
>
> *6. It should be problem oriented and not computer oriented, even if this implies an appreciable increase in the amount of work which has to be done by the computer.*

Perhaps the most dramatic of these objectives was number six which is at odds with typical objectives for programming languages: efficiency, speed of execution, smallest possible executable "footprint," and more intuitive and useful representations for computer primitives (memory addresses, op-codes, etc.). Subsequent statements of object for Simula significantly modified this original goal.

One reason for the change was market driven, "the success of SIMULA would, regardless of our insistence on the importance of problem orientation, to a large extent depend on its compile and run-time efficiency as a programming language." (pp. 447) The other reason for the change was as a perceived lack of conflict between problem orientation and computer orientation. Simula's developers discovered that, "good system description capabilities seem to result in as a more simple and logical implementation," (pp.447) thereby reducing the load on the computer's capabilities.

The focus on problem description, and the resulting simplification of the implementation, promoted by Simula's developers is paralleled in a paper by David Parnas, written in 1972[3]. Parnas'

davidwest 7/30/2003 5:02 PM
Deleted: written nine year s later

davidwest 7/30/2003 5:04 PM
Deleted: ,

davidwest 7/30/2003 5:04 PM
Deleted: called "On Decomposition."

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

paper examined two conceptual abstractions for decomposing complex systems for the purposes of developing software. One was top-down functional decomposition, the approach gaining widespread acceptance under the label "structured design." Functional decomposition is based on an attempt to model the performance of the computer and software and to translate the requirements of the domain problem into those computer-based constructs.

Parnas offered an alternative approach called "design decision hiding," modeling and decomposing the problem or the problem domain without consideration of how the component parts of that domain or problem would be implemented. He was able to show that his alternative led to simpler, easier to read, easier to maintain, and more composable software modules than functional decomposition. Unfortunately his advice was essentially ignored as the juggernaut of structured development came to dominate, (at least officially), the manner in which software was conceived and implemented.

Decomposition into sub-units is necessary before developers can understand, model, and build software components. Both Parnas and the SIMULA team point to an important principle: if decomposition is based on a "natural" partitioning of the domain the resultant models and software components will be significantly simpler to implement and will, almost as a side effect, promote other objectives like operational efficiency and communicational elegance. If, instead, decomposition is based on "artificial," i.e. computer derived, abstractions like memory structures, operations, or functions (as a package of operations) the opposite results will accrue.

davidwest 7/30/2003 5:08 PM
Deleted: Both Parnas and the SIMULA team point to an important principle.

davidwest 7/30/2003 5:08 PM
Deleted: we

davidwest 7/30/2003 5:09 PM
Deleted: I

davidwest 7/30/2003 5:09 PM
Deleted: that

davidwest 7/30/2003 5:09 PM
Deleted: as

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

A corollary of this principle is that design trumps direct expressiveness in a computing language. Performance, the minor and restricted reason for selecting a programming language noted above is as much or more a matter of proper design as it is of direct manipulation of hardware and virtual machine components. Object thinking leads to better designs that reduce the demand placed on the machine so raw efficiency and speed are far less critical than presumed by most developers.

davidwest 7/30/2003 5:10 PM
Deleted: Remember the minor reason

davidwest 7/30/2003 5:12 PM
Deleted: Nygaard's and Parnas' observations are consistent with that assertion.

As the SIMULA team expanded their understanding of the simulation problem domain, and the kinds of systems to be simulated, they identified needs for more generalized and inter-related components. These components had to be implemented and implementation required adding sophistication to the language, resulting in the ideas of objects, classes of objects, data and implementation hiding, virtual procedures, and inheritance.

The legacy of Simula is twofold. First, and most important from the perspective of object thinking, it provided an orientation (a philosophy) of giving primary importance to understanding and modeling the problem domain. This philosophy led the quest for an elegant and powerful language that would allow direct mapping of components in a domain to the modules employed in the computer. Second, a number of original concepts, with appropriate vocabulary (object, class, inheritance), and some important implementation tricks like abstract data types and compiler generated structures were invented or advanced as the language developed.

What use did the inheritors of SIMULA make of this legacy?

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

C++

Bjarne Stroustroup was motivated by the desire to create "a better C." The C programming language is noted for its power and conformity to machine architecture; which assures that C programs are maximally efficient in terms of machine resources. This same power, however, made its misuse almost inevitable. Bugs were easy to create and difficult to track down. Too many C programmers lacked the discipline necessary to properly utilize the language. In Simula, Stroustroup saw a model for introducing discipline to the C language.

> *C++ was designed to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming. ... While a modest amount of innovation did emerge over the years, efficiency and flexibility have been maintained without compromise.*

Stroustroup was concerned with creating a "suitable tool" for projects such as the writing of, "a significant simulator, an operating system, and similar systems programming tasks." His focus was on the machine - systems level programming - and on the program. Even though he found Simula to be an excellent tool for describing systems and directly mapping application concepts into language constructs, he seemed to be more concerned with performance features of Simula than its descriptive capabilities.

Simula's class based type system was a huge plus but its run time performance was "hopeless:"

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

> *The poor runtime characteristics were a function of the language and its implementation ... The overhead problems were fundamental to Simula and could not be remedied. The cost arose from several language features and their interactions: run-time type checking, guaranteed initialization of variables, concurrency support, and garbage collection ...*

Simula was conceived to make it easier to describe natural systems and simulate them in software even if that meant the "computer had to do more work." The inefficiencies noted by Stroustroup were indeed intrinsic to the language and the paradigm created by Simula. Stroustroup essentially rejected the Simula philosophy because his problem domain was the computer itself and performance was the primary goal.

> *C with Classes [precursor to C++] was explicitly designed to allow better organization of programs; 'computation' was considered a problem solved by C. I was very concerned that improved program structure was not achieved at the expense of run-time overhead compared to C. The explicit aim was to match C in terms of run-time, code compactness, and data compactness. To wit: someone once demonstrated a three percent systematic decrease in overall run-time efficiency compared with C. This was considered unacceptable and the overhead was promptly removed.*

C++ maintained the goal of adding program structure without a loss of performance. The constant measure of the language was the machine, either the physical computer platform or the virtual

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

machine - the program. This constant focus on machine performance limited what could be borrowed from Simula - most importantly Simula's goal of being a general systems description language.

Although the claim is made that C++ is a general purpose programming language, that assertion should be modified. C++ is a general-purpose language for describing and efficiently implementing programs that model software implementation constructs (e.g. control structures, data structures, algorithms), virtual machines, or hardware elements.

The focus on the machine is C++'s greatest strength and its greatest weakness. Stroustroup explicitly rejected the philosophy and values behind Simula and merely borrowed some of its implementation tricks, thereby created a language that inhibits the direct expression of application designs in any domain except that of the computer itself. The "simple and logical implementations" observed by Nygaard and advocated by Parnas of non-computer problem domain designs cannot be expressed in C++ without some degree of compromise with those principles upon which the language is predicated.

Smalltalk

> *Philosophically, Smalltalk's objects have much in common with the monads of Leibniz and the notions of 20th century physics and biology. Its way of making objects is quite Platonic in that some of them act as idealizations of concepts - Ideas - from which manifestations can be created. That the Ideas are themselves manifestations (of the Idea-Idea) and that the Idea-Idea is a-kind-of Manifestation-Idea - which is a kind-of-*

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

> *itself, so that the system is completely self-describing - would have been appreciated by Plato as an extremely practical joke.*

Kay describes Smalltalk as a "crystallization of style" language, one that is an expression of, "the insight that everything we can describe can be represented by a single kind of behavioral building block ... " - in essence an object. From the outset, Kay is characterizing Smalltalk as deriving from the same goals as motivated Simula - a desire to have a simple and expressive language for describing and representing (simulating) naturally occurring complex systems.

> *Object-oriented design is a successful attempt to qualitatively improve the efficiency of modeling the ever more complex dynamic systems and user relationships made possible by the silicon explosion.*

Note the absence of any reference to computer or program efficiency or organization. When Kay first encountered Simula and its objects and object manipulation constructs he experience a kind of "epiphany" as, in Kay's mind, ideas from mathematics, philosophy, and biology came together.

> *Bob Barton had said ... 'The basic principle of recursive design is to make the parts have the same power as the whole.' For the first time I though of the whole as the entire computer and wondered why anyone would want to divide it up into weaker things called data structures and procedures. Why not divide it up into little computers ...I recalled the monads of Leibniz, the 'dividing nature at its joints' discourse of Plato, and other attempts to parse complexity. ... It is not too much of an exaggeration to say that most of my ideas from then on took their roots from Simula - but*

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

> *not as an attempt to improve it. It was the promise of an entirely new way to structure computations that struck my fancy.*

Another major stream of influences that shaped the development of Smalltalk was education and cognitive theories - ideas about how people (often children) think or can be encouraged to develop thinking skills. The computer, for Kay, promised a potential vehicle for supporting and promoting thinking, the foundation for an alternative advocated by Marvin Minsky.

> *It was clear that education and learning had to be rethought in the light of 20th century cognitive psychology and how good thinkers really think. Computing enters as a new representation system with new and useful metaphors for dealing with complexity, especially of systems.*

Although Kay's account of the origins of Smalltalk addresses issues of machine efficiency, (almost always in the context of making performance conform to human user expectations for dialog), and compactness, (an over-riding goal was to create a Dynabook or at least a notebook computer), they are overwhelmed by other goals. Expressiveness in describing complex systems, support for education and dialogic interaction between children and machines, and even a search for beauty in programming languages are important examples.

> *One part of the perceived beauty of mathematics has to do with a wondrous synergy between parsimony, generality, enlightenment, and finesse. ... When we turn to the various languages for specifying computations we find many to be general and few to be parsimonious. For example, we can define universal machine languages in just a few*

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

> *instructions that can specify anything that can be computed. But most of those we would not call beautiful, in part because the amount and kind of code that has to be written to do anything interesting is so contrived and turgid. ...*
>
> *A fertilized egg that can transform itself into the myriad of specializations needed to make a complex organism has parsimony, generality, enlightenment, and finesse - in short, beauty ... Nature is wonderful at both elegance and practicality - the cell membrane is partly there to allow evolutionary kludges to do their necessary work and still be able to act as components by presenting a uniform interface to the world.*
>
> *- Kay 19xx*

Alan Kay clearly was not interested so much on what went on inside the machine as how the existence of the machine redefined the act of communication between person and machine. He saw the personal computer as a potentially liberating and creative device, but its potential impact was inhibited by the mode of communication. He saw that a better **language** not *programming language* was required.

There are two lessons and one assertion to be derived from our brief historical retrospective:

- Lesson: the essential differences among programming languages are those that reflect philosophical ideals and values. Those values and ideas, in turn, determine the degree to which a language naturally and simple expresses design concepts without resorting to "contrived and turgid" code.
- Lesson: if you **think** about design using an implementation language – as programmers and especially Extreme Programmers are wont to do – your **designing** will be enhanced or severely restricted by that language.

davidwest 7/30/2003 5:14 PM
Deleted: true

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

- Assertion: If your development project involves modeling, designing, and solving problems in the domain defined by the boundaries of the computer itself (e.g., operating systems, device drivers, network infrastructure) you will best be served by languages like C++, C#, and Java. If, like the vast majority of software developers, you are interested in modeling, designing, and solving problems in an application space you will be far better served to use languages like Smalltalk, Lisp, Fortran, COBOL, and Visual Basic. (With Smalltalk and Visual Basic being generally applicable to a wider variety of application domains.)

Languages are not the only implementation decision that affects design in a similar way. The decision to employ a relational database is an implementation decision with the same kind of implications as selecting a language. Consider the following example.

A customer in the real world may use many different addresses for different reasons. A domain reflective object model might have the partial diagrammatic representation in figure 2.1 (a). Address is a collection – a recurring field – and therefore, according to the dictates of relational database design[4], cannot be an attribute of Customer. In fact we have to create two entities with a relationship between them as shown in figure 2.1 (b).

F02xx01a

Figure 2.1a

Domain reflective Customer Model (partial)

F02xx01b

Figure 2.1b

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

Relational Customer Model (partial)

In the store I might ask a customer, "where would you your purchase delivered?" The customer would think a bit and give me back an address. But in the relational example I cannot ask the customer this question because the customer does not know the answer. Instead, I have to ask the customer for his customer number and then ask the collection of addresses in the Address relation, "which of you belongs to customer 'custNo' and also has the type value 'delivery'?" The implementation code gets even more "contrived and turgid" when I attempt to account for the fact that the customer may want to use different delivery addresses at different times or for different situations.

How does the human customer select an appropriate address. We do not know, and in one sense we do not care because the method is inside her encapsulation barrier, but we might imagine a simple mechanism so that we can model it. We will presume that the customer has an "address selection rule" that contains variables for time of year, state of special circumstances, and similar

Analogous design problems occur when you use strongly typed languages (the real world is pretty fuzzy when it comes to classification) or other constructs that effectively represent the computer but not the application domain.

Programming languages are concrete manifestations of a set of values, ideas, and goals; themselves but parts of, or reflections of, a more inclusive world-view or philosophical context. Given that most of computing and software development emerged from the western world[5], a common world-view or philosophical context might be assumed.

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

This is clearly not the case.

Formalism and Hermeneutics

For philosophers the eighteenth century is considered the "Age of Enlightenment," or the "Age of Reason." Other descriptive labels for the era include the "Age of Science," or the beginning of the "Age of the Machine." The Universe was considered a kind of complicated mechanism that operated according to discoverable laws. Once understood these laws could be manipulated to bend the world to Man's desire. Physicists, chemists, and engineers daily demonstrated the validity of this worldview with consistently more clever and powerful devices.

Writers such as Descartes, Hobbes, and Leibniz provided the philosophical ground that explained the success of science and extended the mechanical metaphor to include human thought. For these thinkers the Universe was comprised of a set of basic elements. Literally, the Periodic Table of Elements in the case of chemistry, properties such as mass for the physicists, and mental tokens in the case of human thought. These basic elements could be combined and transformed according to some finite set of unambiguous rules: the "laws of nature" in the case of the physical sciences, or classical logic in the case of human thought.

Descartes, Hobbes, and Leibniz are likely to be familiar to most readers as rationalist philosophers. Perhaps less well known is the fact that all three were convinced that human thought could be simulated by a machine – a foundation idea behind classical artificial intelligence research in the 1970's. All three built or attempted to build mechanical thinking / calculating devices. Leibniz was so entranced with binary arithmetic (he invented aspects of binary logic) that it influenced his

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

theology, "the void is zero and God is one and from then all things are derived," and prompted an intense interest in the Chinese *I Ching*, (Book of Changes), which has a binary foundation.

This tradition of thought, this worldview or paradigm, has been labeled, "formalism." Other names with various nuances of meaning include "rationalism," "determinism," and "mechanism." Central to this paradigm are notions of centralized control, hierarchy, predictability, and provability (as in math or logic). If one could discover the tokens and the manipulation rules that governed the Universe you could specify a syntax that would capture all possible semantics. You could even build a machine capable of human thought by embodying that syntax in its construction. It is not a coincidence that all three philosophers, (Descartes, Leibniz, and Hobbes), attempted to construct some form of thinking machine.

As science continued to advance, other philosophers and theoreticians refined the formalist tradition. Russell and Whitehead are stellar examples. In the world of computing, Babbage, Turing, and von Neumann assured that computer science evolved in conformance with formalist worldviews. In some ways, the ultimate example of formalism in computer science is classical Artificial Intelligence as seen in the work of Newell, Simon, and Minsky.

Formalist philosophy has shaped western industrial culture so extensively that even cultural values reflect that philosophy. For example, *scientific* is good, *rational* is good, and being *objective* is good. In both metaphor ("our team is functioning like a well oiled machine") and ideals (*scientific* management, computer *science*, software *engineering*) western culture-at-large ubiquitously expresses the value system derived from formalist philosophy.

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

Computer science is clearly a formalist endeavor. Its roots are mathematics and electrical engineering. Its foundation concepts include data (the tokens), data structures (combination rules), and algorithms (transformation and manipulation rules). Behind everything else is the foundation of discrete math and the predicate calculus. Structured programming, structured analysis and design, information modeling, and relational database theory are all prime examples of formalist thinking. These are the things that we teach in every computer science curriculum.

As a formalist the computer scientist expects order and logic. The 'goodness" of a program is directly proportional to degree to which it can be formally described and formally manipulated. Proof – as in mathematical or logical proof – of correctness for a piece of software is an ultimate objective. All that is bad in software arises from deviations from formal descriptions using precisely defined tokens and syntactic rules. "Art" has no place in a program – in fact there is no such thing as art. Art is nothing more than formalism that has yet to be discovered and explicated.

Countering the juggernaut of formalism is a minority worldview of equal historical standing even though it does not share equal awareness or popularity. Variously known as "hermeneutics," "constructivism," "interpretationalism," and most recently "postmodernism" this tradition has consistently challenged almost everything advanced by the formalists.

Hermeneutics, strictly speaking is the study of interpretation, originally the interpretation of texts. The term is used in religious studies where the meaning of sacred texts, written in archaic

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

languages and linguistic forms, must be interpreted to a contemporary audience. Husserl, Heidegger, Gadamer, Dilthey, and Vygotsky are among the best known advocates of hermeneutic philosophy.

Hermeneutics (*er men u tiks*) is derived from the name of the Greek god Hermes – the messenger or god of communication. It is a difficult name and does not flow easily off the tongue like "formalism." Unfortunately there is no comfortable alternative term to use. Most of the philosophers most closely associated with this school of thought – excepting Heidegger – are probably unknown to most readers. Unfortunately, there is not space to fully explicate the ideas of these individuals in this book. It is strongly suggested, however, that your education – and your education as a software developer in particular – will not be complete without a reasonably thorough understanding of their ideas.

davidwest
Comment [1]: This is a placeholder for properly formatted phonetic spelling of the word. I am sure the Word can do this, but did not have time to discover how.

The ideas of the hermeneutic philosophers are frequently illustrated with examples from linguistics, but hermeneutic principles are not limited to that domain. For example, words (the tokens of thought according to formalists) do not have clear and unambiguous meaning. The meaning (semantics) of a word is negotiated, determined by those using it, at the time of its use. Semantics are ephemeral and emergent from the process of communication. Therefore, the possibility that any formal syntax, no matter how comprehensive or sophisticated, is capable of capturing the semantics of the natural world is denied.

The hermeneutic conception of the natural world claims a fundamental non-determinism. The world is deemed to be chaotic - at minimum. More often they assert that the world is self-organizing, adaptive, and evolutionary with emergent properties. Our understanding of the world, and hence the nature of systems we build to interact with that world, are characterized by multiple perspectives and

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

constant change in interpretation. Contemporary exemplars of this paradigm are Gell-Mann, Kauffman, Langton, Holland, Prigogine, Wolfram, Maturana, and Varela.

> Murray Gell-Mann, Stuart Kauffman, Christopher Langton, and John Holland are closely associated with the Santa Fe Institute, the study of complexity and of artificial life. This new discipline challenges many of the formalist assumptions underpinning classical science – suggesting that significant portions of the real world must be understood using an alternative paradigm based on self-organization, emergent properties, and non-determinism. Ilya Prigogine is a Nobel prize winning physicist (as is Gell-Mann) whose work laid many of the foundations for the study of emergent and chaotic physical systems. Steven Wolfram, developer of *Mathematica* and expert in cellular-automata has recently published *A New Kind of Science*, that suggests all we know can be best explained in terms of cellular-automata and emergent systems. Humberto Maturana and Francisco Varela are proponents of a "New Biology" consistent with complex systems theory and collaborators with Terry Winograd of a hermeneutic theory of design strongly influenced by the philosophy of Heidegger.

As exotic and peripheral as these ideas may seem, they have been at the heart of several debates in the field of computer science. One of the best examples is found in the area of artificial intelligence. The formalists represented by Newell and Simon arguing with the Dreyfus brothers and others representing hermeneutic positions.

> Allen Newell and Herbert Simon are among the leading advocates of traditional artificial intelligence – the theory that both humans and machines are instances of "physical symbol systems." Both humans and machines "thought" by manipulating tokens in a formal way (Descartes *redux*) and therefore it was perfectly possible for a digital computer to "think" as well as (actually better than) a human being. Hubert L. Dreyfus working with his brother was one of the most vocal and visible critics of traditional AI. *What Computers Can't Do* and *What Computers Still Can't Do*,

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

written by Hubert, present arguments based on the work of Husserl and Heidegger against the formalist understanding of cognition.

Another example centers on the claim for 'emergent" properties in neural networks. Emergence is a hermeneutic concept inconsistent with the formalist idea of a rule-governed world. Arguments about emergence were heated. The stronger the claim for emergence by neural network advocates, the greater the opposition from formalists. Current work in cellular automata, genetic algorithms, neural networks, and complexity theory clearly reflect hermeneutic ideas[6].

Marvin Minsky is another leading advocate of traditional AI. He was vehemently against the idea of emergent properties in systems – a view which seemed to soften in later years. His book, *Society of Mind*, attempted to use object-oriented programming ideas to develop a theory of cognition that could rely in interactions of highly modularized components without the need for emergent phenomenon.

The hermeneutic philosopher sees a world that is unpredictable, biological and emergent rather than mechanical and deterministic. Mathematics and logic do not capture some human-independent truth about the world. Instead they reflect the particularistic worldview of a specific group of human proponents. Software development is not a "scientific" nor an "engineering" task. It is an act of "reality construction" that is political and artistic.

Formalism and hermeneutics contest each other's basic premises – the core assumptions made about the nature of the universe and the place of humanity within that universe. Challenges to basic assumptions are frequently challenges to core values as well. Fundamental assumptions, and values, are

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

seldom examined. Like articles of faith they are blindly defended and arguments at this level come to resemble "religious warfare."

As noted previously, Western culture in general is largely formalist (using the labels "rationalist" and "scientific" rather than formalist) in its orientation. Anything challenging this position is viewed with suspicion and antagonism. It is for this reason that the conflict between hermeneutic and formalist worldviews frames the debate about an object paradigm.

XP is the most recent example of a series of attempts to apply hermeneutic, human-centric, and aformal ideas to software development. Some antecedents include the "two cultures" identified by Robert L. Glass[7] and the associated debates over the role of creativity in software development; the conflicts between "fuzzies" and "neats" in AI; and the classic debates between devotees of Smalltalk and those of C++.

Behind the Quotes

Robert L. Glass

A prolific writer and chronicler of ideas in software development as well as a leading practitioner, Robert L. Glass appears frequently in publications ranging from *ACM Communications* to his own newsletter, *The Software Practitioner*. His extensive experience in the real world and the world of academe make his insights into the conflict between "how it is done" and "how theorists think it is done" invaluable for everyone involved in any kind of software development. One of his main themes is that practice requires a great deal more creativity and aformalism than computer scientists, software

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

engineers, and academicians are willing to acknowledge. His latest work, *Facts and Fallacies of Software Engineering*, Addison-Wesley, 2003, presents in a concise and highly readable fashion many of the critiques of formalist approaches to software development that are presented here as foundation positions for object thinking and extreme programming.

More recently, Michael McCormick notes that:[8]

> *What XP uncovered (again) is an ancient, sociological San Andreas Fault that runs under the software community - programming versus software engineering (a.k.a. the scruffy hackers versus the tweedy computer scientists). XP is only the latest eruption between opposing continents.*

XP is the latest assertion of the view that people matter. XP is the latest challenger to the dominant (and hostile) computing and software engineering culture. XP is the latest attempt to assert that developers can do the highest quality work using purely aformal methods and tools. And XP is the latest victim of the opprobrium of the formalists and the latest to be told that its approach is only suitable for dealing with small non-critical problems.

To the extent that objects (our present focus) are seen as an expression of a hermeneutic point of view they have been characterized as anti-rationalist, or at best non-rationalist, challenges to the prevailing philosophy. It is my assertion that objects are (or are perceived to be) a reflection of

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

hermeneutic philosophy. A way to test this assertion is to compare objects to other ideas about software development that are clearly hermeneutic – i.e. postmodernism.

Postmodern Critiques

> *There is a controversy smoldering in the computer science world at the intersection of two important topics: formal methods and heuristics. The controversy, though it may sound esoteric and theoretic, is actually at the heart of our understanding of the future practice of software engineering.*
>
> *What is meant by formal methods? Techniques, based on a mathematical foundation, which provide for systematic approaches to problem solution. What is meant by heuristics? Techniques that involve trial-and-error approaches to problem solution.*
>
> *It should be noted that careful reading will disclose a potential middle ground. Formal methods, perhaps, are appropriate for solving mechanistic and well understood problems or parts of problems; heuristics are necessary for more complicated and creative ones.*

Robert L. Glass probably would not identify himself as a postmodernist but he does provide a good transition to a consideration of postmodern philosophy and computer science. The preceding quotes are consistent with the contrast between formalist and hermeneutic philosophy discussed in the

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

previous section. The third quote additionally suggests the superiority of heuristics over formalism when we need to address large-scale complicated (and possibly complex) systems that involve human beings.

Glass's arguments in favor of heuristics and creativity in software design mirror the hermeneutic arguments against formalism. Formal approaches simply will not work beyond a certain scale. Formalism works "close to the computer," is highly questionable at the level of an application, and fails at the level of complete systems and architectures.

The work of Terry Winograd provides complementary parallel to Glass while making a more direct link to hermeneutic philosophy. Early in his career he was a strong advocate for classical AI and was a strong formalist. Exposure to the ideas of Humberto Maturana and Francisco Varela along with the philosophical works of Martin Heidegger prompted his reconsideration of AI's formalist tenets.

Winograd has turned his attention from formal modeling of the world to the issue of designing software to be used in the real world. He characterizes design as: a conscious act, human centric, conversational or dialogic in nature (between artifacts like software and hardware and human beings who are users of same), creative, communicational, with social consequences, and done as a social activity.

One of the influences on Winograd's thinking was the work of Humberto Maturana and Francisco Varela regarding the evolution of autopoietic (self-organizing) biological and cognitive systems. The structural-coupling mechanism used by cells to establish cooperative complex structures,

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

and eventually all the familiar forms of flora and fauna, provided one bridge to the social nature of computer systems and their development.

Another influence was the work of Martin Heidegger – a hermeneutic philosopher. Writing with Fernando Flores, Winograd explored the work of Heidegger and its implications for computer system design.

One of the most important implications was the denial of "intrinsic truth or meaning" in any artifact – whether it was a computer, a piece of software, or a simple statement in a natural language. This claim is also central to the school of thought that has been labeled, "postmodern." It is also one of the core claims of all the hermeneutic philosophers. Because of this implication, the design of computer systems must, for Winograd and other postmodernists, be refocused on the use of software and hardware as communication devices for a particular group of people at a particular point in time.

While Winograd and Flores concentrated on Heidegger's notion of "breakdown" and issues of communication, Christiane Floyd and her co-authors extended the discussion to include other facets of postmodern philosophy.[9] The role of politics and power relationships in both the imposition of a software artifact on a community of users and in the group dynamics of those charged with the creation of the software artifact in the first place are central concerns of Floyd and her colleagues. In a similar vein, Richard Coyne[10] addresses the issue of design in a postmodern age.

The importance of all this work – beginning with Glass's concerns about creativity and including the postmodernists concerns with computer system design – is a clear extension of hermeneutic

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

arguments against formalism. It is also the context in which objects as a software development metaphor were coined. The first expression of "object thinking" by Alan Kay and the researchers at Xerox PARC, were concerned, like the postmodernists, far more with people and communication issues than they were technical computer and formalist issues.

Self consciously or not, the object community was concerned – in the 1960s and 1970s with the exact same issues raised by the hermeneuticists in the 19th and 20th centuries and the postmodernists of the 1990s. To an object advocate, objects are valuable because they facilitate user-computer interaction and communication among members of development teams. Objects enhance the "art" of software development but not necessarily the "engineering."

Adherents to formalist ideas, including computer scientists and software engineers, dismissed objects as irrelevant. When objects technology looked like it might make serious inroads into real world development the formalists attacked in the same manner that they attack other critiques of formalist approaches – for example, the creativity discussed by Robert Glass.

Simultaneous with the attack on basic philosophy, the traditionalists began to lay claim to the "form" of objects by equating them with the "black box module." They were also quick to adopt less threatening innovations like abstract data types as logical extensions of traditional software engineering theory. As a consequence the object technology that became widely adopted was the formalist recasting of object ideas rather than the "pure" object paradigm itself.

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

A similar phenomenon is evident in the "patterns movement" where many of the core philosophical ideas of Christopher Alexander (original inspiration for the attention paid to patterns) are being dismissed or co-opted by traditional software developers who like the form of patterns but are uncomfortable with the more esoteric ideas.

Rejecting Mysticism

Before he wrote, *A Pattern Language*, Alexander published, *The Timeless Way of Building*. The former book is cited by everyone in the patterns movement as an inspiration for their own efforts, but the latter is seldom mentioned. Of course, it is in the latter book that Alexander's mysticism is most evident. Consider the following:

> *A building or town will only be alive to the extent that it is governed by the **Timeless Way.** To seek the **Timeless Way** we must first know the **Quality Without A Name.** To reach the **Quality Without A Name** we must then build a living pattern language as a **Gate**. Once we have built the **Gate**, we can pass through it to the practice of the **Timeless Way**. An yet the **Timeless Way** is not complete, and will not fully generate the **Quality Without A Name**, until we leave the **Gate** behind.*
>
> *Christopher Alexander*

The quoted statements are actually chapter headers from *The Timeless Way of Building*. They read far more like the Taoist (and later Zen) story of the Boy and the Bull which is an allegory of the process of obtaining enlightenment.

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

It is not surprising that computer scientists and software engineers, entrenched in a formalist culture, find little of value in this aspect of Alexander's work. You have to wonder, however, how deeply they understand Alexander's ideas about patterns if they dismiss what are clearly, for Alexander, fundamental philosophical presuppositions. Is it not possible (likely) that they are reading into Alexander's pattern ideas their own philosophical biases? And, if so, how valuable was Alexander's contribution after all?

This example exposes a bias of the author – you cannot claim to understand something, in this case object thinking and extreme programming unless you are able to understand the form, the substance, and the presuppositions that support form and substance.

Deciding to be an object thinker or an extreme programmer is a decision to set oneself in opposition, in very important ways, to mainstream software development thought and practice. It is a decision likely to result in becoming a member of a minority, using "niche" languages to solve problems in "non-critical" and "small scale" problem domains. Unless … we can use our knowledge of the history and philosophy just discussed to effectively wage a true software development revolution.

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

The arena of software development is filled with the discarded or dilapidated remains of "revolutionary" languages, methods, tools, and techniques. It is probably safe to say that any real revolution will come about only as a result of majority adoption of new ways of thinking, of alternative world-views and associated philosophical values, i.e., from the establishment of an object culture.

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

[2] The reader is encouraged to explore the material available at www.squeak.org, both to see the past, present an future of Squeak and to see additional information on the common philosophical roots behind Squeak and object thinking.

[3] Parnas, D.L.. "On the criteria to be used in decomposing systems into modules," Communications of the ACM, 15(12): 1053-8, December 1972.

[4] As taught in every data modeling text book, except some advanced texts on post-relational database design, of which I am aware.

[5] This is an historical observation, not an expression of ethnocentrism or an attempt to claim computing for Europe and the US. As will be seen in the rest of the discussion this is more of an indictment of the history of computing than a boast. Clearly the roots of computing and many of the most important contributions to our understanding of computing come from many different places and cultures. The argument will be made in the next few pages that most of computing, however, is firmly grounded in a philosophical tradition that arose in Europe and formed the foundation of "the Age of Reason."

[6] Do not confuse *formalism* with the use of formal tools like mathematics that are employed in the cited fields of study.

[7] Glass, Robert L. *Software Creativity*. Englewood Cliffs, NJ: Prentice Hall. 1995.

[8] McCormick, Michael. Programming Extremism. *Communications of the ACM* 44(6), June 2001, 109-110.

Unknown
Field Code Changed

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003

david west 10/16/2016 9:26 AM
Deleted: 12:08 PM

[9] Floyd, C., H. Zullighoven, R. Budde, and R. Keil-Slawik (eds.). *Software Development and Reality Construction*. Springer-Verlag. 1992. Also, Dittrich, Yvonne, Christiane Floyd and Ralf Klischewski, *Social Thinking, Software Practice*. MIT Press. 2002.

[10] Coyne, Richard. *Software Development in a Postmodern Age*.

david west 10/16/2016 9:26 AM
Deleted: 7/30/2003
david west 10/16/2016 9:26 AM
Deleted: 12:08 PM