SOFTWARE ENGINEERING FOUNDATIONS

A Paradigm for Understanding Software Design Methods

Software Ontology

Kent D. Palmer, Ph.D.
Software Engineering Technologist
PO Box 4402 Garden Grove CA 92642
palmer@netcom.com

Software ontology concerns the ultimate reality of the software process and product. The study of software ontology forms the necessary basis for understanding software engineering[1] and particularly software engineering methodologies. Software ontology is a branch of metaphysics directed at a particular class of entities that have a different foundation in Being that is very rare. This same field of study has been called Grammatology, the science of traces, by J. Derrida[2]. Being is not just an empty concept, but covers all ways things have of making themselves present within our world. These particular kinds of entities called "software" have a different way of making themselves present in showing and hiding relations than we are normally used to in

1. See Ghezzi, C., Jazayeri, M., & Mandrioli, D., Fundamentals of Software Engineering (Englewood Cliffs NJ: Prentice Hall, 1991)
2. See Of Grammatology (Baltimore: Johns Hopkins U.P., 1973); See also Ulmer, G.L., Applied Grammatology (Baltimore: Johns Hopkins U.P. 1985); Gasche, R., The Tain of the Mirror (Cambridge MA: Harvard U.P., 1986)

our interactions with technological systems. Understanding these different ways of presencing peculiar to software must be the starting point for any truly deep paradigm for understanding software design methods, because these methodologies are ways of attempting to show software entities. Treating software as if it were merely another instance of the kinds of things we normally encounter within the world, will certainly lead to anomalies and conundrums which can only be avoided by attempting to understand the strange nature of software from a philosophical perspective.

Metaphysics gets its name from the type of philosophical questions it asks that are beyond the realm of questions answerable by physics, or any science that addresses a particular kind of thing. It normally is composed of epistemological questions about how we know what we know and ontological questions that concern the reality of the things we know about. The deepest level of questioning concerning the universe which is not religious, i.e. theological, is ontological. Ontology deals with the most general concept we have about the nature of things which is their Being. Being

concerns the showing of things within our world. Not all things show themselves in the same way within the world. This is a discovery of modern ontology which we will use to attempt to understand the nature of software. If we wish to have a really deep foundation for our scientific paradigms, it must be ontological. Since we are not taught to think at this level of generality, it is very difficult to get a picture of the differences in the way things manifest. However, unless we understand these differences, we are likely to commit a category mistake in our reasoning about software. The following attempts to clarify these issues to prevent this mistake in our classification of software entities that might cloud our ability to understand the general phenomena of software.

Software is a relatively new kind of object that we might chance to discover when we look around us within the world. The strange thing is that most people never see software. They see automated machinery or occasionally computers, but rarely are the essential instructions that make computers function ever seen except by software engineers. Normally, software engineers are so busy trying to build

software that they do not question whether its ontological foundations are any different from any other kind of object. In fact, normally we live without questioning the reality of anything within our lifeworld[3]. It has been said:

Philosophy may be ignored but not escaped; and those who most ignore least escape.[4]

As software engineers, it is our failures in building software systems that cause us, at moments of desperation, to wax philosophical. Because engineers normally do not have philosophy as a part of their training, these moments of reflection on the ultimate truth of Murphy's laws are experienced as dark nights of the soul where existential absurdity becomes palpable. Camus used the myth of Sisyphus[5], who eternally pushed a boulder up a mountain only to watch it roll down again, and again as the image for the absurdity of life. Murphy's law--If anything can go wrong it will--is only an addendum to the basic message of the myth of Sisyphus that states that

3. This term was introduced by Husserl in Crisis of European Sciences and Transcendental Phenomenology (Evanston: Northwestern U. P., 1970) and developed by Alfred Schutz in his two volumes on The Structures of the Life-world (Evanston IL: Northwestern U.P., Vol. 1, 1973; Vol. 2 with Thomas Luckmann, 1989). It refers to the world as experienced from a phenomenological perspective.

4. David Hawkins; p419 [Klir, G.; Architecture of Systems Problem Solving: (NY: Plenum Pres, 1985)]

5. See The Myth of Sisyphus (London: Hamilton, 1955)

anything can start the boulder rolling back down the hill at anytime so be expecting it. For the software engineer and the user of computer software this image of the boulder is transformed into the disaster of spaghetti code that is totally incomprehensible, or the tragedy of the system going down, or the nightmare of losing a disk. Software engineering itself is a response to these existential problems as they relate to the software portion of the technical system. Software engineering takes as its premise that if we concentrate on the way we build software and attempt to introduce rigor into the development process, it will be possible to build higher quality systems and avoid the terrors of existential absurdity for both ourselves and the users of the software we build. Unfortunately, there appears to be a hitch. Software systems are not yielding readily to traditional pragmatic engineering disciplines. If they had, software engineering would have never separated from hardware engineering. The addition of the word "engineering" and recent modifications to standard software production practice has not been enough to subdue the monster (seen by some as a werewolf[6]). Superficial changes in the discipline of programming and its

management have only served to emphasize the enormity of the problem we face as software engineers. As we search for ways to apply engineering discipline to software production, we are faced more and more with the question, "What is software anyway?" and "How can such a simple thing as a few commands to a stupid processor cause so many problems?"

Normally, software is treated as a product by software engineers. However, some researchers have realized that it is an entity in the world which can be treated as the object of scientific inquiry. Software science is also a fledgling discipline. Software science treats software texts, compiled code, and running software systems as objects of study. Applying the scientific method to these objects, the software scientist attempts to understand the characteristics of the software. Halstead[7], who coined the term attempted to discover complexity metrics. However, anyone who treats software from an objective point of view as a subject of study may be classed as a software scientist. For instance, software scientists may be interested in measuring and

6. See Brooks "No Silver Bullet: Essence and Accidents of Software Engineering" in Information Processing '86, H.-T. Kugler, editor (North Holland: Elsevier Sci. Pub. 1986) and in IEEE Computer, April 1987, pp 10-19.

7. Elements of Software Science (North Holland, Elsevier, 1977)

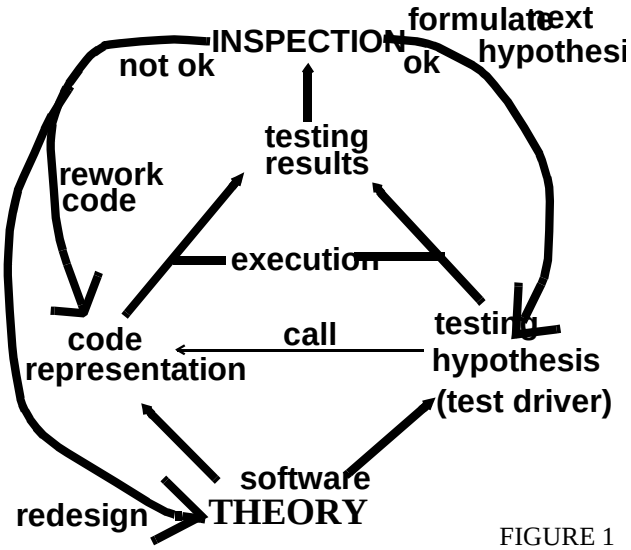


FIGURE 1

modeling performance of computer systems and networks. Software engineers produce the product which is used by the end user. The software scientist observes the product, the production process, or the usage of the product and attempts to analyze and measure the resulting system. Software engineering and software science are complementary disciplines which at this time in practice merge into an ambiguous whole. However, in the future we must learn to separate the engineering perspective from the scientific perspective as has been done in other disciplines. We need to be able to switch back and forth between these perspectives in order to hone our software practices much the way mechanical and chemical engineers know and use physical and

prajna {Buddhism}, wisdom *source of meaning*	**traceless** *trackless* *lostness*	**difference^5** *emptiness* *unthinkable*	*a decernment of a decision on a dircrimination*
touch touching {Merleau-Ponty} *inscription in the flesh {Deluze}* *chiasm {Merleau-Ponty}* *reversibility*	**no trace** *palimpsest* *labyrinth*	**difference^4** *Being^4*	*a decision on a discrimination of a demarcation* *interference between traces* *what appears after cancellation of traces*
arche-writing ~~*origin*~~, ~~*essence*~~, ~~*beginning*~~ *{erasure}* *hinge* *spacing*	**trace** *discreteness* *differance* *reserve*	**difference^3** *Being^3*	*a discrimination of a demarcation* *the indenture of the line in the paper beyond the r* *Grammatology, the science of traces*
writing/speech *origin, essence, beginning* *transformation across discontinuities*	**sign** *system & structure* *text production* *noesis/noema* *mark, code*	**difference^2** *Being^2*	*a distinguished distinction or a demarcation* *a difference that makes a difference {G. Bateson}* *relevance, significance, varities, kinds*
book, paragraph, sentence etc. *discourse, rhetoric, presentation*	**form** *morphe* *letter, picture, word*	**difference^1** *Being^1*	*a distinction, threshold, variation of the variation* *outlines of things, laws of form {G. Spencer-Brow* *duality, transcendence, domination*
colors, tints, hues, tones, styles *mood*	**content** *hyle*	**difference^0** *beings, Being^0*	*variation, heterogeniety, undistinguished, indeter* *indistinct, indecisive, undecernable, indiscrimina*
cleared, cleaned, leveled *prepared surface* *genocid, ethnic cleansing*	**tablet** *the writing surface* *the clean slate* *blank sheet* *hewn*	**anti-difference** *no difference* *no-thing* *indifference* *identity*	*plenum of pure difference {homogeniety = hetero* *suppression of variety* *differences that make no difference* *nihilistic landscape produced*
partially ordered *arranged, fitted together*	**texture** *rough hewn* *garden*	**sameness** *"things," social entities* *the They, das Mann*	*selected variety, minimal modification of what is* *excluded middle, prior to identity* *all things embedded in the social fabric*
natural variety *unordered, unarranged*	**resources** *natural complexes* *unhewn materials*	**alterity** *otherness* *alien*	*unsuppressed discovered unadulterated variety* *deep ecology*

TABLE 1

chemical scientific knowledge. Because our software engineering and software scientific disciplines are so new, we are hard put to answer just what knowledge software science has to offer the software engineer, or vice versa.

Software science and engineering hope to draw upon the tradition of science and engineering that has developed in relation to other more advanced disciplines. It naturally looks to those more mature disciplines for guidance in how to treat its products and objects. Unfortunately, it seems that a problem arises at this point. There doesn't seem to be any direct mapping of techniques or approaches from these more developed science and engineering disciplines into the software disciplines. In fact, as we look for help, we often find that these other more mature disciplines, despite their successes, are themselves in trouble. They are themselves undergoing profound changes and are undergoing critical examinations of their own foundations. Thus, we do not discover a horde of certain knowledge and practices which our fledgling science can rely on unquestioningly. Instead, we find that we are forced ultimately to invent much of our new software disciplines. So we end up asking ourselves, "What is engineering and does it apply to software?" and "What is science and how does it study software?" As we ask these questions, we find that our understanding of science and engineering is altered fundamentally. Ultimately this causes

us to ask, “What is software, anyway, that it causes us to rethink science and engineering?” We know from experience that software is strange stuff. It is for us what quick silver was for the ancients (***liquid metal?***). It is a marginal object of some kind. More primitive societies always treated marginal objects as taboo[8]. This is why the werewolf comes readily to mind as a mythical image. Marginal objects always require special initiation, purification, and ritual observances. In our culture this means special higher education like the Software Engineering Institute’s software engineering masters curriculum, special training on the job in producing software systems, and special methodological approaches which are very different from normal programming in the small. Marginal objects are hard to categorize, difficult to handle, impossible to pin down, and seem to be crazy combinations of normal things thrown together in an outlandish fashion. Every culture treats these objects with care using special precautions. The question is, “Why does software deserve this status?” and “Is there any way to turn software into just another normal thing that can be manipulated without

8. See Mary Douglas, Purity & Danger (London: Routledge, Kegan, Paul, 1960)

special precautions?" This brings us back to the question of the nature of software.

The nature of software is the domain of software ontology[9]. This discipline is more basic that either software engineering or software science. It is the root of both of these disciplines. Software ontology could be defined as the interface between software science and software engineering. Software science seeks to discover the characteristics of software objects. Software engineering seeks to discover ways of making software products which may be considered as objects by the software scientists or tools by the end user. Between the making and the observing of software, somewhere lies the essence of the software itself. The observers and the producers of software deal with this essence of software every day. The essence of software provides the ultimate constraints on what may be observed about software or what can be created in a software system. Yet it is difficult for us to see what exactly the essence of software is. We are caught up in all the accidental characteristics of particular software

9. Others are also becoming aware of the importance of ontology in different ways. See also "An Ontological Model of an Information System" Wano, Y. & Weber, R. in IEEE Transactions on Software Engineering; Vol. 16 #11; November 1990. "A Rigouous Model of Object Reference, Identity, and Existence" Kent, W. in Journal of Object-Oriented Programming; June 1991; pp.28-36

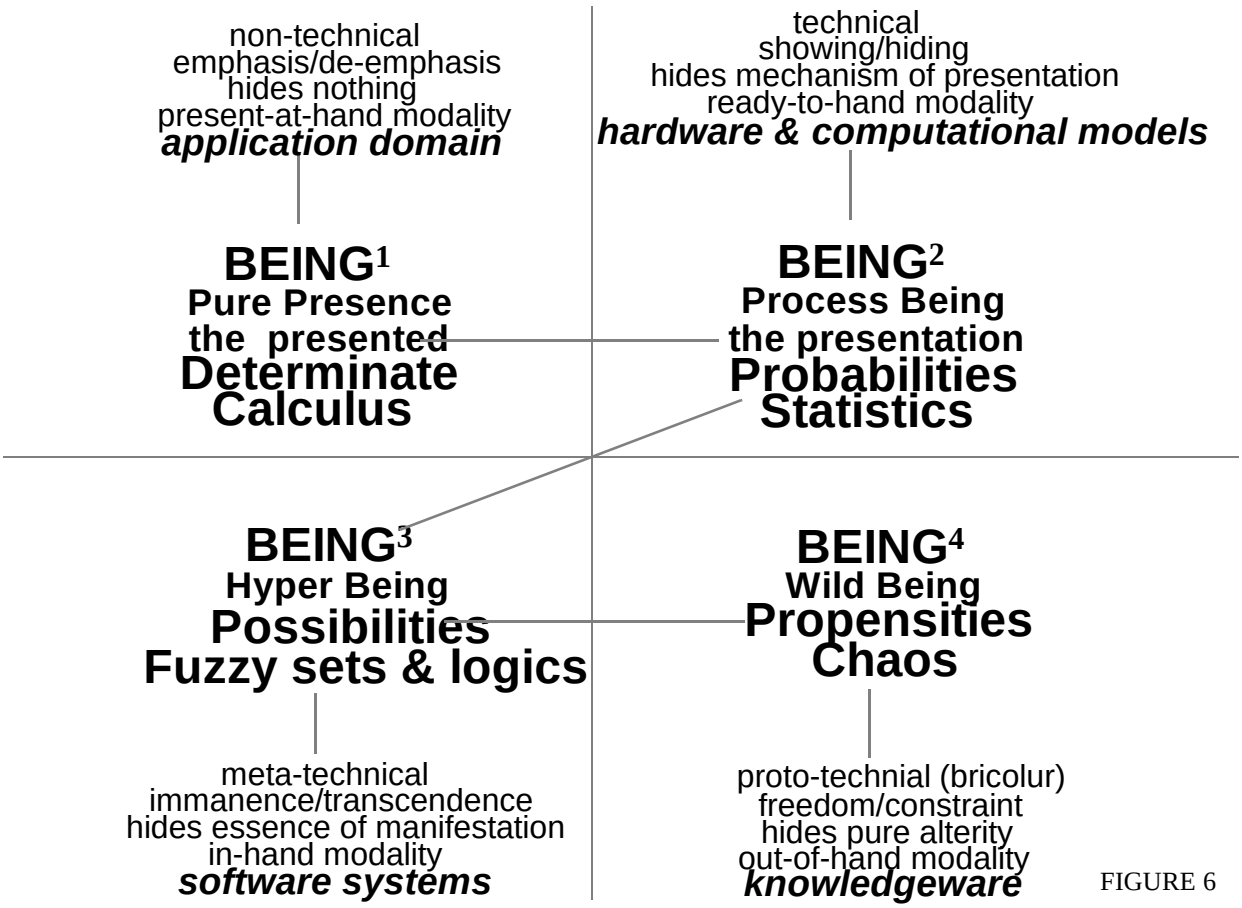


FIGURE 6

systems. There is no ideal pure software system, but many different particular systems in different languages, running on different machines, doing various applications. Attempting to see the essence of software and thus define an ultimate statement of what software itself is, cannot be done either inductively or deductively. We recognize software when we see it, but we cannot deduce all its forms from axioms nor see it clearly through all the diversity of software systems. We recognize that what we call software is

different from other things, but cannot pin down exactly what the difference is. Thus, it is necessary to start with the discipline of ontology itself. What can ontology tell us about any “thing”.

It turns out that in this century many advances have been made in the discipline of ontology. These are not as well known as the advances in physics and other disciplines. This is because it appears to be a highly esoteric field that doesn’t seem to apply to anything because it applies to everything. What can be said with assurance about everything would appear to be very little. In fact, until the turn of the twentieth century ontology was little more than a recounting of the Western view of reality as defined by Kant in the 18th Century. Around the turn of the century, though, things began to get interesting because it was discovered that the received view of reality was more complex than first imagined. Since then the history of ontology has been a progressive dismantling of the Western worldview which has coincided with the discovery that the physical world was very different than expected as well. The development of physics and the dismantling of the Western worldview, along with the

systematic research into the other worldviews as a result of global colonization of the third world by the West, has led us to an interesting existential situation. It turns out that our worldview, as expressed by philosophers, has been discovered to be groundless. This means that there is no ultimate philosophically secure basis for our view of the world that we have foisted upon everyone else via colonization. The groundlessness of our worldview is expressed within ontology as what will be described as the fragmentation of the concept of Being. On the other hand, other worldviews discovered in the process of colonization are not only more sophisticated, but agree in significant ways with the puzzling physical nature of the universe discovered by modern physicists. We have used our worldview efficiently to dominate the world and discover the nature of the physical universe, while at the same time we have discovered that this worldview has no claim to any ultimate truth. We could, perhaps, live with our worldview which confers power over nature and other nations regardless of doubts of its claims to truth if it were not for the dangerous side effects which indicates a fundamental imbalance. This imbalance in our worldview,

because of its groundlessness, causes us to destroy the environment upon which we ourselves are dependent. The list of palpable side-effects of the imbalance in our Western worldview are too many to catalogue here, and others have been more eloquent in bringing attention to these areas of concern[10]. The point is that the combination of the side-effects of the unbalanced and groundless worldview, with its inability to comprehend physical reality, throws everyone who thinks about the world in which we live today into a tizzy. The power we exercise over our environment and each other through the knowledges derived from our scientific and engineering disciplines is awesome, yet is offset by our powerlessness to solve basic environmental and social problems. In philosophy these core issues concerning the adequacy of our worldview are dealt with under the rubric of the relation between man and technology. Man creates technology through a skillful combination of science and engineering. Looking at technological systems produced by men, we gain immediate insight into our own core. It is our product just as the nests are the products of birds and the webs are the products of spiders. By observing

10. See Morris Berman Coming To Our Senses (NY: Simon & Schuster, 1989)

technology, we gain insight into the nature of the Western worldview in both its positive and negative aspects. We see one of the possibilities of man taken to its ultimate conclusions. Through this we learn more about the nature of man and things.

Technology and the interlocking technological system which is supported by software systems plays an important role in our relation with our environment and each other. Software plays a unique role in mediating this relationship. Software may be thought of as a transitional object[11] mediating the relationship between man and technology. In a way, software is a meta-technology which allows us to control technological systems which are themselves controlling physical, chemical or social processes. This meta-technology allows us to adapt technological systems to new circumstances without re-tooling. It eases our interaction with the technological systems and gives us more control and more information concerning controlled processes. Thus, in observing technology, we must pay special attention to the meta-technology which software represents. There is a symbiosis

11. This phrase is sometimes used in psychology for objects that make other things accessible. See Winnicott, D.W., Playing and Reality (London: Travistock, 1971).

occurring between men and technology which software allows to be flexible, adaptable, accessible, and semi-intelligent. We are beginning to see the world in terms of software systems. By decoding the genetic code, it has become possible to reprogram the genes in biological machines called organisms. Simultaneously, we are destroying the gene-pool and thus radically limiting the materials with which geneticists might work. We are beginning to see the relation between mind and body in terms of the relation between software and hardware, and thus have developed neuro-linguistic programming to alter behavior of individuals. Software is providing us with new tools for organizing information about the world. Different software tools run on the same computer providing different applications. These tools are extending the range of our understanding of the world. At the same time we are learning to see the world itself in terms of a software metaphor. The field of Cognitive Psychology is an example of a discipline that primarily sees the mind/brain as an information processor. This strange development, accelerated by the advent of personal computers and workstations, brings the question of the nature of software to the

fore for everyone, but especially for software engineers. We make the stuff and don't even think to wonder what it is.

◆ ◆ ◆

This series of essays[12] will focus on the central concern of software engineering. That concern is the definition of the correct methods for designing software. Other aspects of software engineering rigor appear to be straightforward, even if they are expensive, difficult to implement, or hard to define, measure and control. An excellent introduction to these other aspects will be found in Software Perspectives[13] by Peter Freeman. This book offers a wide perspective on the nature of the software engineering discipline into which many of the other textbooks on software engineering practices need to be fit. This broad perspective offers a structural model that focuses on External Organization, Objectives, Policies, People, Procedures, Information, Tools, Software Systems, and Supporting Technology. The perspective on software engineering offered by Peter Freeman will be

12. This essay is the first in a series on Software Engineering Foundations. The second essay concerns Software Systems Meta-methodology which goes into depth concerning the structure of software engineering methods and its links to General Systems Theory. The third essay presents the outlines of the Integrated Software Engineering Methodology in terms of the formulation of a design language for specifying real-time software designs.

13. (Menlo Park CA: Addison-Wesley Pub. Co., 1987)

assumed as the context for the rest of the discussion in this essay. The functional model consisting of the activities of Analysis, Design, Construction, Delivery, Support, and Management presented by Freeman is very basic and needs to be augmented with a fully articulated process model such as IEEE P1074 Standard for Software Lifecycle Processes[14] or an equally relevant process model like that produced by Brian Dickinson in his book Developing Quality Systems[15]. To these process models must be added a lifecycle model[16] such as the waterfall model exemplified by the military software standard DOD-STD-2167A[17], or the spiral model of Barry Boehm[18], or the stepwise refinement model defined by Vaclav Rajlich[19], among others, in order to get a complete picture of the complexity of software engineering regimen. These books set the stage for a discussion of the software engineering methods. Methods are particular implementations of crucial software engineering processes. Non-trivial processes are defined by procedures.

14. IEEE publication.
15. (McGraw-Hill Software Engineering Series, 1989)
16. See my paper "The Future of Software Process" in which an autopoietic life-cycle is developed as an alternative to the others mentioned. Autopoiesis means self-organizing.
17. The new version of this, so called 2167b, is officially MIL-STD-SDD which now allows spiral life-cycle models to be used..
18. The Software Productivity Consortium of Herndon VA has taken these ideas first presented by Dr. Boehm and have developed them.
19. He has developed stepwise refinement into a formal life-cycle with the same rigor of the waterfall.

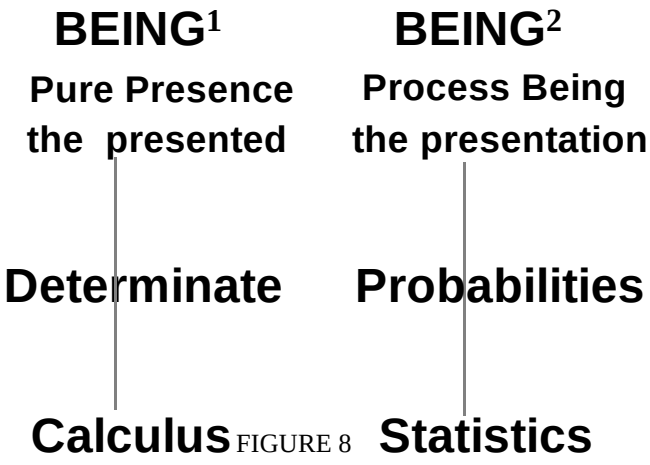


FIGURE 8

Procedures have, according to Freeman, two dimensions answering the questions, “How the procedure works” and “what the procedure produces.” Procedures may work as Methods, Techniques, Guidelines, or Heuristics. Procedures may produce Technical, Quality Assessment, or Management outputs. Freeman defines a method as “a mostly complete set of rules for arriving at a desired result; in extreme cases, it may be an algorithm.” He defines the technical product of a procedure as “the primary result of the procedure is intended to be a technical workproduct such as a design or a program.” The technical methods of software engineering are crucial because it is through them that the primary transformations of the software system are performed. The transformation of the software system, from

requirements to specifications, to architectural design, to detailed design, to code, to a tested and integrated deliverable product are each difficult to achieve steps. Methods attempt to make these transformations rational, rigorous, and achievable, if not easy and efficient. Of these transformations the most difficult steps are from requirements to specifications to architectural design. These are called the upstream transformations as distinguished from the others called the downstream transformations. This paper will concentrate upon these upstream transformations. They are still ill defined, and no known published method completely covers these transformations. Examples of published methods are those of Hatley-Pirbhai[20], Ward-Mellor[21], Project Technology (Shaler-Mellor)[22], etc. All of these methods fit Freeman's basic definition of a technical method. Xaiping Song has compared many of these methodologies and has developed a framework for understanding methodologies[23].

A software method is a means of abstracting

20. Hatley, D.J. & Pirbhai, I.A., Strategies for Real-time Systems Development (Dorset House Pub. Co., 1986)
21. Ward, P.T. & Mellor, S.J., Structured Development for Real-time Systems (Englewood Cliffs NJ: Prentice Hall, 1985)
22. Shaler, S. & Mellor, S.J., Object-oriented Systems Analysis (Englewood Cliffs NJ: Prentice Hall, 1988)
23. "A Framework for Classifying Parts of Software Design Methodologies" Xiaping Song & Osterweil, L.J. in Proceedings of the 2nd Irvine Software Symposium, ISS'92; pp49-68; Selby, R.W. editor; University of California, Irvine CA; March 6, 1992.

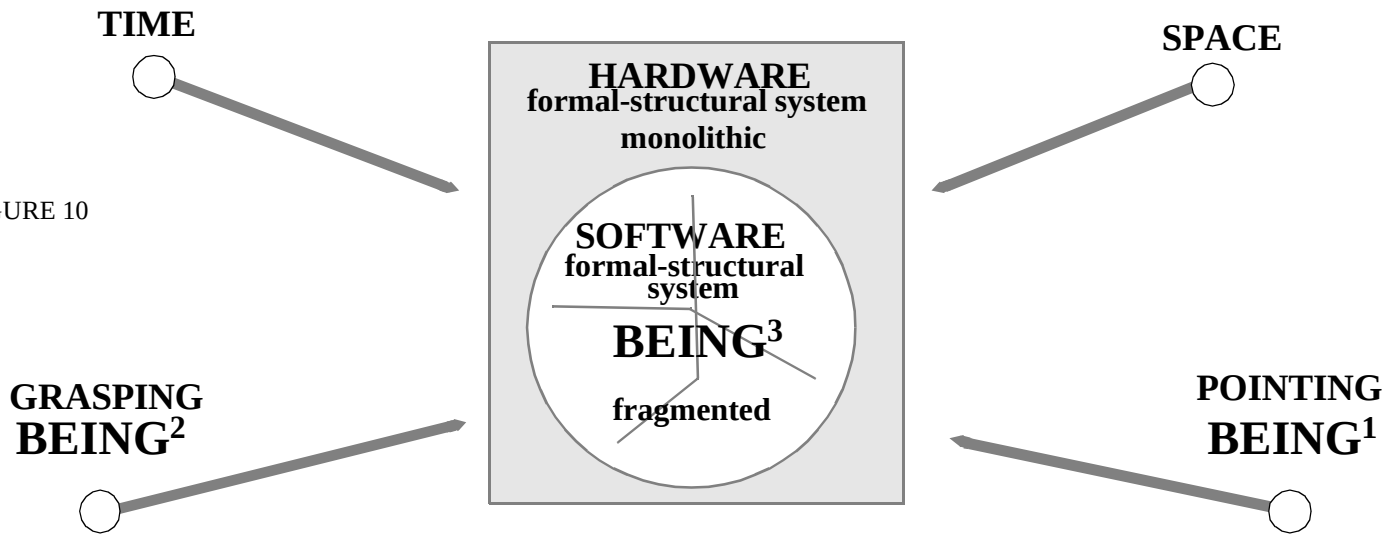
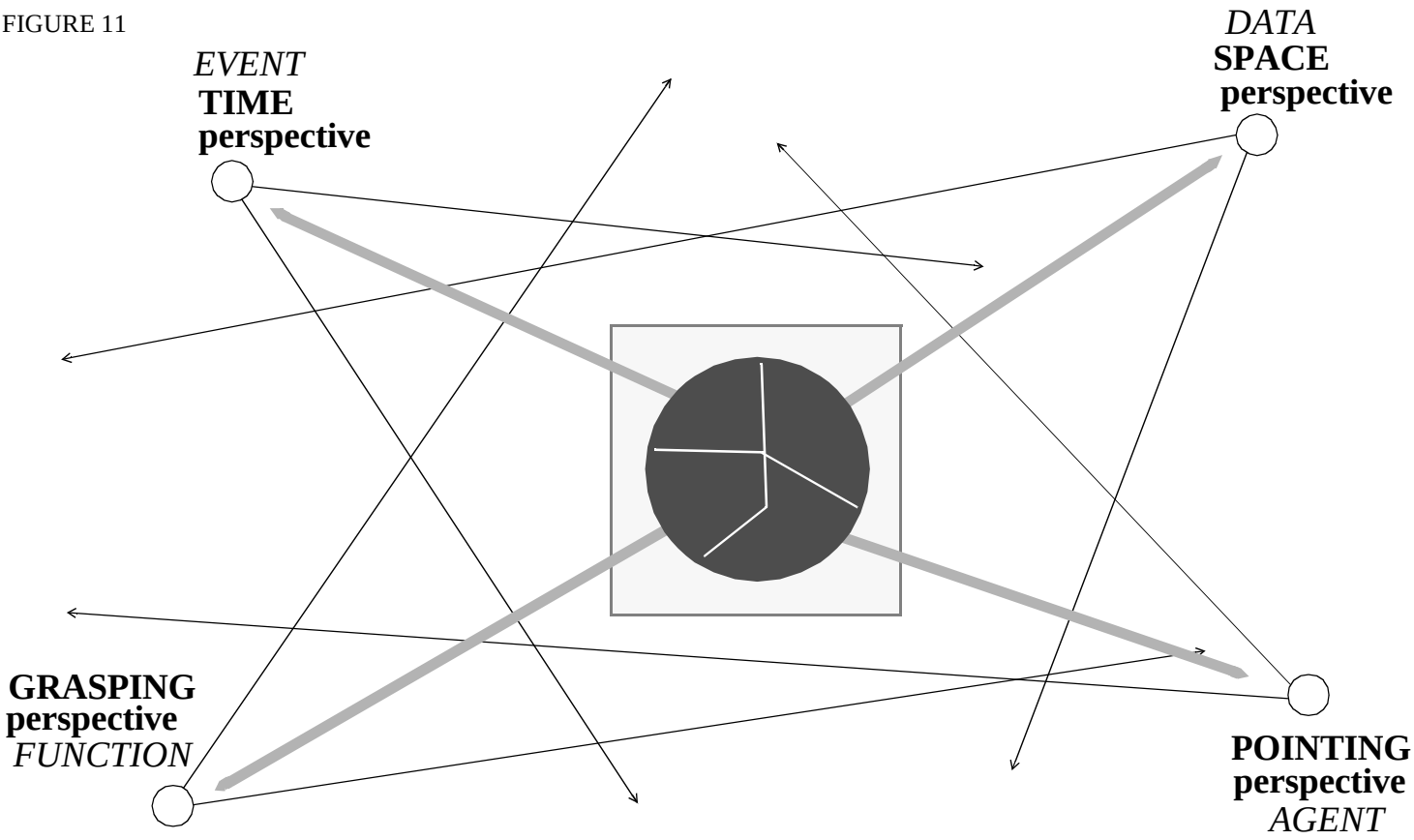


FIGURE 10

certain aspects of a software system from all the other aspects of that system in order to make the design of the whole system tractable. Currently, most software is produced with little reliance on methods. Production of software without the use of methods is called by practitioners “hacking”. Hacking consists of programming a software product directly, using the software language alone as the representation of the system under construction. Software engineering is distinguished from direct programming by the mandatory implementation of methods for transforming abstracted representations of the system under construction. Direct programming of systems may be suitable for small, non-production systems which can be

FIGURE 11



understood by one person or a small team. Even the usefulness of direct programming on small systems is questionable to the advocates of the use of software methods. Direct programming, though, is widespread and is still being taught to computer science students as the exemplary means of arriving at a working program. Thus, software methods are only learned in production environments where direct programming has been demonstrated to break down. Software methods demand that direct programming not occur until late in the

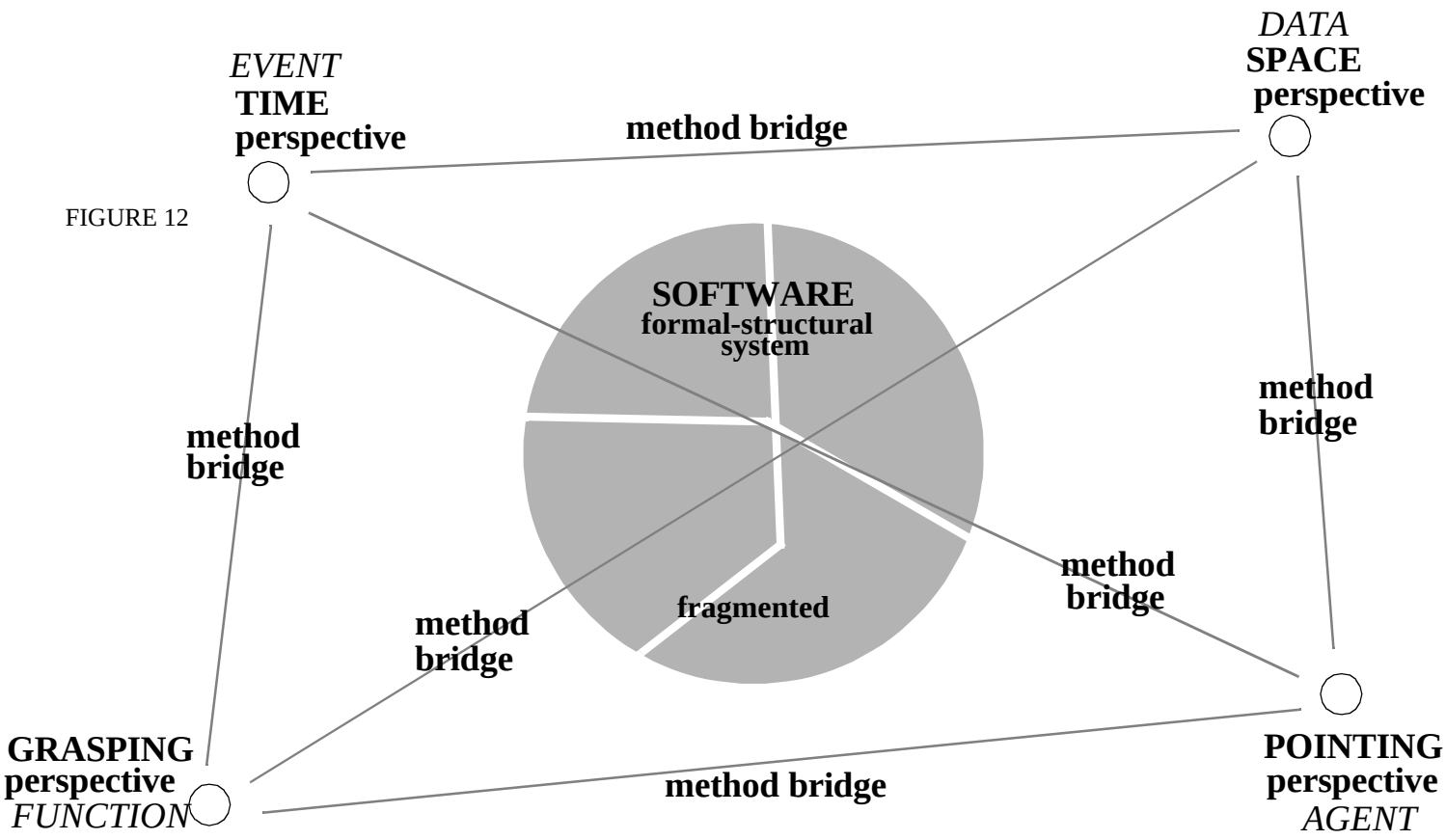


FIGURE 12

lifecycle of the product except on a trial or experimental basis in the case of prototypes. Software methods abstract the basic characteristics of the software system so that problems may be solved within the abstracted form, which is simpler to repair and modify than the programmed representation of the system in a programming language.

Both software engineering processes, which define the work context in which methods operate, and methods for transforming the

software system must evolve to solve the problems faced by software engineering. Watts Humphries of the Software Engineering Institute in his book Managing the Software Process[24] has explained a basic framework for the evolution of software engineering practice into a rigorous discipline. This evolution is seen as a judicious mixture of software science and software engineering. Product and process measurement go hand in hand with control and optimization of the software development process. Assuming that this evolutionarily path is correct, it is based on the assumption that adequate methods will be created which will make software design of large, complex real-time software systems tractable. At this point that is a questionable assumption. There is no doubt that methods will be introduced. There are already a plethora. The question is whether they will be effective in dealing with the basic difficulties of software production. The basic difficulties, which have been described by many practitioners, all lead us back to a questioning of the nature of software itself. The methods which will ultimately be effective in dealing with these problems will directly address the basic nature of software entities.

24. (NY: Addison-Wesley, 1989)

The failure of existing methods to fully address the entire spectrum of software characteristics in a systematic way has led to the current confusion of practitioners concerning the usefulness of particular methods and the efficacy of the use of methods in general. Software engineering methods are in a very primitive, nascent, stage of their development. We are all so happy just to have something to guide us in our work, that it is very difficult to think critically about the methods as they exist now. All the current criticism seems to be of a partisan nature directed against the methods which do not catch our fancy and are balanced by unabashed praise for the methods that inexplicably appeal to us. It is significant that these uncritical attitudes are referred to by practitioners as “religious wars”. Each of us has our favorite methods from among the many that are being proposed. However, the ability to be critical of methods in general has not dawned yet.

Informed criticism is not partisan. It is applied uniformly to everything regardless of personal preferences. This type of “constructive criticism” is Cartesian and is properly scientific. When we are finally able to be

properly critical concerning software engineering methods, we will begin to ask different kinds of questions about methods. These questions will not focus so much on methods themselves and the comparisons between individual techniques proposed by different authorities. Instead, we will begin to focus on the field within which a method is proposed and its functioning within that field. What is meant here can be explained simply with reference to Gestalt psychology. A Gestalt is a figure/ground relationship. Psychologists have shown that there is a dynamic relation between figures and their backgrounds. At the moment each method is a figure. We look from figure to figure, comparing the methods without recognizing that these figures stand out on a common ground. They are all a function of a common field. At the moment that field is invisible to us. All we can see are the figures of individual methods. In order to see the common field from which they arise, it will be necessary to develop meta-methods that explicitly render that background visible. When one attempts to develop these meta-methods, called paradigms in philosophy of science, then the study of software methodologies will begin to enter the

first phases of scientific discipline.

In this paper I will propose a paradigm for understanding the field of software methods. Paradigms, like the methods themselves, are tentative and exploratory at this point. If we do not begin exploring, however faltering our first steps, we will not be able to understand what software methods are really all about. We will be trapped in the partisan wars between methods and authorities without any critical apparatus to decide issues and judge methodological theories. The general theory of software engineering methods has not been developed yet, and this is a first attempt in this direction. Such a theory asks questions like the following:

o What is a software engineering method?

o What is the relation between software engineering methods?

o Is there a minimal necessary set of software engineering methods?

o Is there criteria for judging the quality and effectiveness of software engineering methods?

o How do software engineering methods relate to the foundations of computer science?

o How should software engineering methods be taught?

o How do software engineering methods relate to the definition of the discipline of software engineering?

There will be many different kinds of answers to these and other related questions. The answers will form what will become the paradigms of software engineering in the future. A few of these paradigms will achieve dominance because of their demonstrated value and will vie with each other for supremacy. Occasionally, fundamental changes to these paradigms or their relations will occur, and these will be called the revolutions in software engineering. We are now in the pre-Copernican phase of the development of our discipline. We are longing for the first revolution in which our new discipline will begin to look at itself and its assumptions about the world critically.

◆ ◆ ◆

I will begin by outlining what, I believe, has the makings of a fruitful approach to the construction of a general paradigm for understanding the field of software engineering methods. This paradigm has a more general applicability to the understanding of structural systems[25], but software systems are a specific case of great interest that will be used here as a concrete example. Practitioners will become, as time passes, more sophisticated and begin to take account the relation between software engineering methodology and other disciplines. This is crucial for the development of our methodologies, but it demands that we become familiar with other disciplines in order to apply the advances made there to our own problems. Thus, the tutorial nature of what follows should be taken as an invitation to the reader to explore other disciplines, and to see the connections between our discipline and many others like philosophy of science, ontology and physical science. However, it is not necessary for the reader to have any deep knowledge of these other disciplines in order to understand the paradigm proposed here.

The departure for this paradigm definition is

25. See Salthe, S.N., Evolving Hierarchical Systems (NY: Columbia U.P. 1985); Wilden, A., System and Structure (London: Travistock, 1972)

Peter Naur's[26] idea that the object produced by engineering is a "theory". The theory is not code, nor a design, nor documentation, nor any artifact of the software engineering process. Artifacts are secondary products which are spin-offs from the essential work of building a software theory. Naur declares that the software engineering product is not completely captured by any of the artifacts produced in the software engineering process. Naur cites the inability of new programmers on a project to fully understand how a product works without tutoring by more experienced team members. No amount of documentation seems to be sufficient to transfer a good understanding of the software system. In fact, documentation can actually obscure the software product it is meant to elucidate. The relation between the software product and its documentation is not as clear cut as the writers of military standards might have hoped. In fact, the relation between the software theory, that is the real product of software engineering, and the representations of that theory dictated by software engineering methods is enigmatic and peculiarly complex. It is the recognition of this strange relation that makes software methods intrinsically

26. See Naur, P.; "Programming as Theory Building"; in Microprocessing and Microprogramming; Volume 15 (1985), pp. 253-216 (North Holland Pub. Co.)

interesting, and necessitates the development of software engineering paradigms which explain and help the practitioner to navigate through what is quickly becoming a methodological morass.

The recognition that software engineering does not produce just artifacts, but primarily theoretical structures, is extremely significant. First of all, it immediately allows us to connect our discipline to the core issues of Western science in a way that few engineering disciplines have been connected. All disciplines which function within the our Western philosophical and scientific tradition strive to be scientific. However, few disciplines are as “inherently scientific” as software engineering appears to be. In fact, the fundamental problems in philosophy of science are perfectly exemplified in software engineering in a way which is unique. Science produces a theory of nature. In physical science, which is the preeminent scientific discipline, there is a direct relation between these theories and observations. The closer this relation between theory and observation, the more a discipline comes to exemplifying the inner structure of science. Scientists classically

produce theories of how nature works. From these theories, they develop hypotheses which are then tested. The results of these tests cause changes in the theory which lead to new questions and new theories. As we have seen in the history of physics, this leads us to very interesting views of the world that can be totally counter to our intuition and received authoritative explanations. Different sciences have different traditional relations between theory and observation which cause them to be graded in terms of how “scientific” the discipline appears. Some disciplines are purely formal, like mathematics and logic, with little room for experimentation. Others, like biology, are mostly classifacatory and observational. Still others, like social sciences, have a very difficult time connecting theories to observations. In fact, it seems that as more and more complex dynamical systems are studied, it is more and more difficult to construct adequate theories and to test them in ways that are convincing.

Software engineering is a very interesting case. In software engineering we construct a theory, which like all theories can be described in various ways, but it cannot be completely

captured by its representations. From this theory, one representation is constructed called the “code” that runs on a computing machine. We have hypotheses, which are tested concerning how this representation should function. These hypotheses are either borne out or refuted by the tests. Thus, in software engineering there is a close, direct relation between the production of the code representation of the theory and the hypotheses of how this representation should function which is held up against actual performance in test. Testing causes us to alter our code representation of the theory, to a greater or lesser degree, in order to bring the hypothesis into sync with the results of testing. Sometimes we are forced to change the theory upon which the code representation is based. This is a fundamental change called a redesign. This close, immediate relation between software theory and test results, which is so important in software engineering, is in some ways even more direct than the same connection that appears in physics. We are not dealing with cosmic questions of how the universe works, but the practical question of how a piece of software performs its intended function. But the recognition that the piece of software is a

good or bad representation of a theory, allows us to see here a more general phenomena than might be expected.

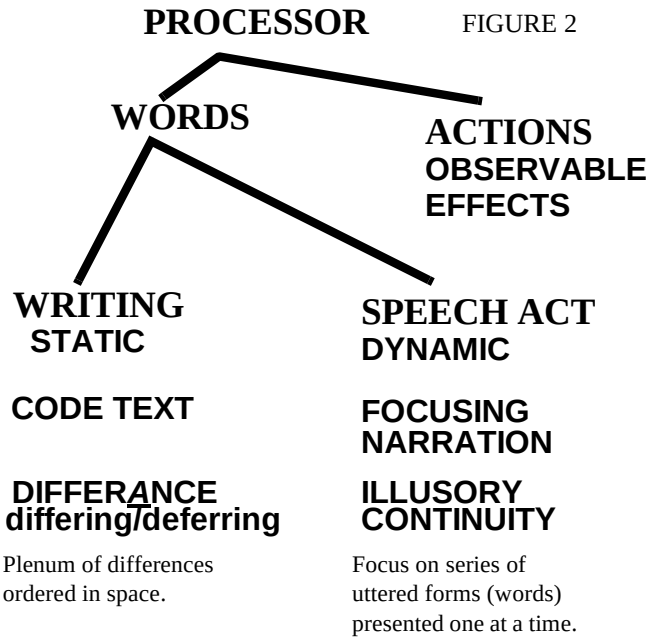


FIGURE 2

As Persig notes in Zen & the Art of Motorcycle Maintenance[27], all machines are embodied theories. Machines have been developed into a high art by our civilization. We have mechanical machines, electronic machines, and are even developing light-based machines. However, software machines have a special and unusual place in the scheme of things. Our appreciation of the peculiarity of software machines, as opposed to all other types of hardware machines, is crucial to the founding

27. (NY: Bantam, 1974)

of the software engineering discipline. Software machines are written. They cannot function without their hardware computational platforms, whether mechanical, electronic or light based. When software is written, it is a static representation which, when executed, becomes dynamic and does things in the world. The relation between code and test is mediated by the relation between the static representation and its dynamic realization executed on hardware. The transition between the written representation and the execution of instruction is a fundamental transformation. We never really know what will happen when we cross that boundary. We create the code representation which we hope is fully understood. Then we test that representation within the dynamic environment. What happens is always slightly different than what was expected. Closing the gap between the code representation and its dynamic effects is a crucial phase of software engineering. The written nature of software allows us to, relatively easily, change it and to narrow the gap between hypothesis and test results. Once a change is made and the software works, it does not change back so that it no longer works unless there is a random coding error.

Perfected aspects of software remain perfect until the world changes or we want something else. Software, once it works, has minuscule reproduction costs in contrast to hardware production costs. Software does not suffer from the effects of entropy, except in terms of its relation to its media. These properties are very different from normal machines which deteriorate through the action of entropy, and have material costs associated with production, which mixes different kinds of materials together in very precise ways to achieve the desired results.

The fact that software is written is a key to understanding why it functions as it does. It has recently been realized by Western philosophers that writing is inherently different from speaking. Almost all our models of reality are based upon vision and speech. However, writing has always served as the shadowy bearer of our civilization. It has always been assumed that it worked the same way as speech, but was merely frozen in a different form. The French philosopher Derrida has recently pointed out that, in fact, writing functions very differently from speech, and our society has constantly denied this

difference. Writing is what makes continuous civilization possible, yet it has, from the time of Plato, been denied and denigrated in relation to the preeminent place of speech. Writing lacks the inherent continuity of speech. Writing, in fact, falls apart into separate units almost immediately upon being written down, and has an opacity which speech lacks. Writing is hard to produce, and has a more formal dimension because it will be scrutinized over and over. Speech vanishes almost as it is spoken (unless it is tape recorded). But the key difference between speech and writing is what Derrida calls "differ***a*** nce". Writing is constantly differing/deferring. Writing refers constantly beyond itself, and differs from itself while deferring the resolution of meaning. Derrida illustrates this with the example of a book which has a preface. The meaning of the book may be changed entirely by the addition of a preface written after the book is completed. Thus, writing in its spatial organization allows the juxtaposition of passages written at different times in any order. The arrangement of these passages can completely alter the meaning of what was written earlier, but presented later, or vice-versa. This is unlike speech which has an inherent relation between

speech acts and temporal ordering. Speech acts always appear in the order of performance. In writing, meaning is deferred until the whole has been seen. In speech, meaning is more immediate and localized to the utterance. Also, writing is a heterogeneous plenum of differences. Any text appears as a string of signs whose differences are the crucial feature. Speech, on the other hand, creates an illusory continuity which is its strongest feature. Speech organizes the stream of differences and causes the hearer to apprehend one speech performance at a time in a series determined by the speaker, not determined by the listener. Speech is a focusing narration that creates a series of figure/ground relationships in a specific order. In speech the speaker is in control instead of the listener. A book, on the other hand, can be opened anywhere. There is no dynamic focusing mechanism, other than the reader's own attention and sense of what is relevant, that forces the reader to apprehend the presented figures in a particular order. There is, instead, just the suggested order linear text itself.

This linearization of the text is exactly what is challenged by grammatology, that is the

science of traces. In the Phaedrus[28] Socrates challenges the speech of Lysias on the grounds that it has no beginning, that each line may appear in any order. The example of the plaque commemorating the death of Midas that also appears in the dialogue is precisely the same, a text without inherent order. When we produce writings, we attempt to impose form on them using book, section, chapter, paragraph, sentence in order to subjugate the text to our will and make it accessible. However, inherently each line of text is independent and can be moved to any place within the full range of the collection of the text itself. This ability of the text to become a pure plenum of difference without externally imposed form is the basis for the major difference between speech, which naturally organizes itself into gestalts which linguistics studies and thus differentiates itself from text that must be artificially organized by the writer. Text lends itself much more to a multidimensional writing and reading which today is known as hypertext. There is no beginning and no end--merely a labyrinth of differences. Derrida's grammatology is aimed at understanding these differences; and, in fact, it is possible to locate

28. Eros: The Bittersweet; Anne Carson (Princeton NJ: Princeton U.P., 1986, pp124-146)

Grammatology in a hierarchy of differences that parallels what will be called the meta-levels of Being.

Table 1 attempts to locate differance in a series of stages that follows the articulation of the meta-levels of Being[29] by describing different meta-levels of difference. What is clear from the table is that difference level zero does not occur in a vacuum. There are levels of articulation that form the context for that lowest level of difference that produces the context within which that possibility arises. The ultimate context is pure alterity from which all man's production takes its resources. There is a natural variety, which we discover ourselves situated within, which is unordered and not arranged by us. That primal condition is full of unsuppressed, discovered, unadulterated natural variety. The human being may begin by rearranging and ordering this natural variety without suppressing it. This brings about a rough hewn world much like the hunter gatherers must have lived within before the advent of the earliest agrarian civilizations. In this environment it is the sameness of things and not their differences

29. This ontological model will be fully developed later in the paper.

that is most important. Things are not yet abstractable and perceptible as beings; there is, instead, a myriad of textures which itself is all the Same in the sense of belonging together. By selecting some of the variety, and ordering and arranging it, the unhewn becomes rough hewn. It is difficult for us to imagine this world which is neither alien nor shot through with differences. It is the archaic protoworld of ambiguity, indeterminateness, and flux in which everything is swimming in the soup of everything else merged and blending together. The word “thing” in Old English was a social event. The things within the sameness are part of the social fabric. That fabric ties the archaic proto-world together by a mutual recognition. What is important about the texture of this sameness is that it must be suppressed for difference to arise.

Difference arises out of identity. Without identity difference cannot appear. Before identity, which is the destruction of the sameness by making things too much the same, there was an apprehension of the world that did not know differences but only variations within the same. Alterity and anti-difference become known in one moment. Alterity, that other, like

Grendel and his mother in the Beowulf saga, which may not be made the Same, which can never be identical, arrises at the same time as the identical, that which suppresses all differences and thereby makes difference visible as difference, rather than as variations in the sameness. Anti-difference created the surface, stage or tablet on which differences may be seen. It produces a plenum of pure difference which is indistinguishable from absolute homogeneity. This tablet becomes the writing surface. It is a cleared, cleaned, leveled surface where all the variations of sameness have been converted into differences that make no difference, a nihilistic landscape which suppresses all original and genuine variety. It is important to realize that without identity, difference would not arise. Difference is a counterpoint to identity; it is the counterbalancing overcompensation to the "clear cutting" that destroys the sameness, producing indifference and an indefinite extent of no-thing out of which beings might appear as abstractions of things. Upon the platform of anti-difference the meta-levels of difference are erected. Sameness has been completely suppressed, along with alterity. Alterity appears reflected in the shattered spider web of

differences of self from itself. But true alterity has been destroyed along with sameness. Anti-difference erects the enclosure in which difference may appear against the background of a totalitarian imposed identity.

Software has exactly this kind of basis in anti-difference as well. In software there is a pure plenum of bits which are all identical except for location. For software everything beyond this pure plenum of indifferent differences is an alterity. Software can only exist where the plenum of pure difference, based on identity, has been constituted. The sameness of the human world is reduced to this tablet of arrayed memory locations. To software the sameness and the alterity are identical. The hardware within which the array of memory locations exist, and the processor, are already a structural system which has been precisely produced to create the plane of anti-difference which exists as identical cpu cycles and identical memory locations. The hardware has abstracted the entire field of differences, reducing all similarities and variations to a minimum and defining everything outside the hardware as completely alien to the system. Within the system the samenesses that would prevent the

discreteness of the differences between the identical space and time places have been isolated and reduced with the same vigor as the alien-ness of dust particles that might blow in from the outside environment beyond the boundaries of the system. Due to these rigid boundaries, internally and externally, the computer system hardware becomes a very fragile kind of a system which must function perfectly for software to work within it.

Upon the pure plenum of identical differences, a whole series of differences in a cascade of meta-levels is formed. The first level is the pure content, which in the case of software is the momentarily magnetized or energized bit. It is these on-bits that form the content which is formed into words or numbers. This formation takes place through the creation of codes. The codes are at a higher meta-level than the forms because it is through the codes that the forms are transformed by the changing of the contents of the forms. So within the field of pure difference there is a distinction between off and on that is the minimal possible variety upon which the whole edifice of software is built. When the bits are randomly set, all on or all off, then the status of the bits are pure content.

When the bits are set to particular patterns, this is the production of a distinction which allows forms to appear. These forms appear as particular patterns of on and off bits which are described and transformed according to a code. The code is at the meta level of the sign that allows a difference that makes a difference to be seen. We see forms directly on the screens of the computer. They are what the software presents to the human being in the form of a presentation. But behind these forms mediating between the pure content of ones and zeros, is the signs and codes that are actually manipulated by the processor as directed by the software. We have described three meta-levels of difference: the variation between on and off which is level zero difference; the distinction, or thresholds, which are based on the variation of the variations that produce the forms of letters on the screen that the user sees which is level one difference; and the differences that make a difference[30] that produces varieties and kinds as codes through the articulation of the semiotic level of the software which is level two difference.

There is another level of difference beyond the

30. See Gregory Bateson, Steps to an Ecology of the Mind (London: Palidin, 1973)

sign, upon which it is based, called the trace. Derrida speaks of this level using the terms discreteness, differance and reserve. Within the software system all the codes, as they are animated, produce a myriad of discontinuities which are dynamically changing. What the codes represent are discrete patterns with limits. Those limits, like the limits to an iterating loop, interact with each other in inexplicable ways to produce a myriad of effects within the software system. The articulation of all those interactions of limits creates a pattern of non-manifestation which stands in reserve, i.e. is held back and never is seen. Derrida calls this manifesting of what is absent (as absence) arche-writing, as opposed to the active writing or speaking which appears on the surface of the system. It is seen in stress testing as the limits of the software system are discovered. Sometimes a system will just stop at a certain point because some constant has been hardcoded, or some resource or design limitation has been reached. Working through the system which is only there to present things is a non-manifestation which patterns the whole. The traces of this non-manifestation, the unconscious of the software system, the limits placed upon the system by the designer

often because he did not think of the particular combination of events that occurred during the design, are constantly being manifested as an emphasized absence, underlined and made "present" by its very lack.

One good way to envisage the trace is to think of the impression on a piece of paper below the one being written upon which can be rubbed with color to see the traces within the tablet of prior writing. Traces are a relation between the substance of the tablet and the act of writing. We can see the traces as the deepest deep-structure of the system, where the natural breaks occur for carving up the problem. It is the frontier of structuralism where the deep-structures deal with non-presence as well as understanding the patterns of presence. The traces are the anomalies in the structures. Traces are the inner relation between the off/on-ness and the physical substrate. What particular bits are off or on are arbitrary within limits. It is within the play that is built into the software system that makes it flexible, that these limitations arise through the actualization of particular unforeseen patterns. Signs operate on probabilities, whereas traces invoke possibilities within the combinatorial

labyrinths of the ultimately untestable software system. For any software system of any size testing all possible conditions is impossible given the limits of computational time. When the limits are breached, catastrophes occur. Thus the trace is the inner relation between the bit and its state which is a trace of the unconscious of the software system itself founded on its nature as writing. As writing, every statement is independent and can be executed in any order so that the phenomena of delocalization as the spread of design elements across lines of code occurs. At the level of design this cumulated delocalization appears as the essence of manifestation which concretely is the part of the design that never appears. That non-appearing non-computable untestable aspect of every design leaves its trace within manifestation.

Derrida speaks of the hinge[31] which is at once both join and break. They are an absence that is paradoxically always already present as a lack. They are perturbations and holes in the field that holds the inscription or representation. For software theories they are the impossibility of seeing the whole of the

31. La Bresure; See Of Grammatology pages 65-73.

formal-structural system at one time. There are holes, or warpages, in the topology of the field within which software design methods operate. These holes, or warps, can only be seen obliquely by their effects on the representations of software theories. We see the traces by their distorting, decentering, and displacing effects. Within software the hinge is that aspect of the software which allows it to fold back on itself without breaking in order to continue running. Many design elements of software have this property. For instance, memory management systems allow the computer to keep reusing memory instead of meeting the limit of system memory. Sometimes there may be what are called memory leaks within the software system that cause it to meet those limits that the memory management system is suppose to avoid. Here the correct return of memory to the pool for reuse is a folding of the software system back on itself so it can avoid a hardware limit. The fold in the software produces a break, but allows it to join across that break in order to continue to create the illusory continuity which would shatter if the limit were encountered and could not be avoided.

Derrida also talks of spacing[32]. Spacing

temporalizes space and spatializes time. Spacing occurs in software in the memory management system; for instance, in the obtaining of a chunk of memory for a certain time. Spacings are the internal juggling of the limits within the software system which allows it to avoid those limits and still manifest what is suppose to be seen. Software continually works around its own limits as well as those provided by the hardware environment. The active work of spacing allows it to recognize those limits and to produce realms of activity within those limits which continually fold back on itself through the hinges to allow the processing to continue. With iterative loops the software exists as wheels within wheels which continually check themselves so as not to get off track. This constant checking of itself produces the spacing by which the software gives itself room to maneuver and to continue executing. Ultimately Derrida calls this continual spacing of itself Differance, the work of differing and deferring, because in the spacing the software is continually avoiding built-in limits. It is the interaction of these limits which cause unintended results that are different from what should occur. The

32. See Of Grammatology; pages 68-69.

appearance of those unintended results differing from the requirements and differed until the right conditions manifest so that eventually failure occurs and the hidden aspects of the design, always there but never manifested, become suddenly known. This difference between what should have occurred and what did occur in the failure is a differance, that is a third order meta-difference.

Derrida writes under erasure in order to show that the actual action of the traces within writing. When we erase what was written before is usually seen through what is overwritten. There is a discordance between what was there before and what is there now. Even if we cannot see what was erased the knowledge that erasure has occurred calls the text into question. Heidegger was the first to use this tactic to move from the conceptual realm of Process Being to the realm of Hyper Being. Derrida took over this ruse as a means of showing the insubstantial nature of the text and the subject that produces the text. For Derrida the text has a special nature where the rules for what is inside and outside are violated. He indicates this with the statement “The inside **is** the outside.[33]" The crossed-out is refers to

the fact that between speech and writing it is difficult to say what is inside and what is outside. Naively we think along with the linguists that thought is covered by speech which is in turn covered by writing. But as Derrida shows we can also think of speech as a kind of writing so that the normal relations are inverted. Writing under erasure is a way to make the special point that in the arena of the trace the natural relations between things are likely to be distorted and some would say perverted. For instance we normally think of software as coming after and as a supplement to the hardware. But in reference to the genetic code we see that in relation to our organisms software comes first before we even dream of the possibility of hardware. Software is one of those dangerous supplements that Derrida speaks of that displaces what they are added to so that the relations between supplement and original become paradoxical. Software itself in its memory management as well as many other aspects continually uses the possibility of writing under erasure as a positive feature. It reuses memory by writing over what was there before as a general feature of its functioning. In certain kinds of software systems, like

33. Of Grammatology, page 44.

genetic algorithms, this feature of erasure is used as a positive part of the system. In genetic programming[34], which uses genetic algorithms to synthesize and evolve computer programs, the pieces of memory which operate under erasure are programs that are evolving through the combination of other programs. Programs of the last generation are broken apart and their pieces swapped then compared against some fitness measure to see if the program has gotten any better at doing the job it is being evolved for. Here the software ~~**is**~~ writing itself. We can see a system like this as erasing each older generation and building the new generation out of the parts of what has been broken up and reconstituted. Erasure is an effect that is normally not used in a positive way in the building of software. But because software can write, read, and erase its own memory location this is definitely a positive feature of software itself.

Above the third order meta-difference there are two higher levels which will be briefly introduced. The next level up is the palimpsest where all the traces are accumulated. Different traces cancel each other out so that a chiasmic

34. See Koza, J.R.; Genetic Programming (Cambridge MA: MIT Press; 1992)

effect of reversibility is created. This chiasm is referred to by Merleau-Ponty as "touch touching[35]" and by Deleuze and Guattari as the inscription of the flesh by the savage social mega-machine[36]. Here one set of limits might be replaced by another set of limits, and the software system might move from one regime to another. When this occurs, the software is oscillating between chaotically-induced states. This occurs quite often in networks. Here the interferences between limits and spacings is so complex that the system becomes opaque. That opaque-ness is the prelude to transparency. The transparency is the ultimate emptiness of all things. It is due to that emptiness that things may have meaning. In software this is the pure discontinuity between all the bits, whether on or off. If each bit is considered independently, the system vanishes. It is only because we consider the bits to be related to each other that we see a software system at all. Software ultimately disperses as a cloud of electrical or magnetic pulses. But before we see that dispersion, there is the moment when software attains to utter alien-ness or blends to utter sameness. At that moment it is

35. See The Visible and the Invisible; op. cit.
36. See Anti-Oedipus; op. cit.

incomprehensible without yet having hit the ultimate limit of impossibility. These higher meta-levels of difference are not important for defining software itself, but only the entities that appear within software. The level of no trace where opacity appears is important for artificial life and intelligence and defines the essence of these kinds of computer programs. The level of emptiness is important for defining the phenomena of virtual reality and cyberspace, another incarnation of what the Japanese Buddhists called the “floating world”.

Derrida speaks in several places in Of Grammatology of cybernetics and the relation of the gramme to the programme. He seems to be aware of the possibility of software as a kind of entity that embodies differeance, that is structured as a series of folds or hinges in which spacing occurs. He does not pursue the point[37]. However, it is clear that the programme is dynamic, and the gramme is static, and it is the dynamic gramme that more closely relates to his concept of the trace as a dynamic meta^3-difference. So software, as the informing of information, is the real subject of grammatology, the science of traces. Traces

37. See Of Grammatology pages 9-10 and 81-87.

are the bit patterns which transform other bit patterns, but beyond that they are the internalized limitations of the hardware and software which never manifest but guide the manifestation of what is presented by the software system. The **pro**-gramme is the active trace that leaves a trace in memory and the phosphorescent dots of the display screen. The active trace traces itself, folding through itself and keeping distances from itself as it avoids its own limits. Whenever we trace the execution of code within a debugger, we are pursuing this active trace which quickly becomes lost in the incomprehensibility through its interaction with other traces. The entire system of multiple interacting agents across a plenum of distributed computing nodes is just the palimpsest which holds all these non-linear interfering traces. This morass is opaque to human understanding to the extent it is operating fast enough to produce an illusory continuity for us. But beyond this opacity, it is ultimately, like everything else--empty because the traces reduced to a non-related array of bits can be seen to not be a system at all. Seeing the traces in the context of multiple overlapping traces and the tracelessness of pure dispersion, allows us to keep software within context as the

meta-level where the traces arise but have not yet cancelled themselves out and disappeared.

This diversion into the esoteric philosophy of Derrida may not be so far afield as it may first seem. Speech is a metaphor for the illusory continuity created by the fast execution of instructions by a micro-processor. In drama a predefined text is spoken in an order determined by the playwright. The written text is executed by the actors in the theater. Thus, the written text from the earliest times was executed on human processors. The writing of the text allowed the playwright to see the text as a whole before it was performed. He could change it to achieve greater effect on the audience. Performance allows the attributes of speech and writing to be combined to heighten the effect of both. At a minimum, the lone reader is always the performer of the text he has chosen to read. Unread books are totally lifeless. It is the introduction of this early form of software, to the dynamic environment of the human processor, that causes interesting things to happen that do not happen in a totally oral culture.

The work of Derrida[38] is crucial for

understanding the problems confronting software engineering, for which, as Brooks says, there does not seem to be any silver bullet. The monstrosity of software is the presentation as lack, absence or arbitrariness of its nature as trace. Software programs are animated writing. The hardware produces an illusory continuity by executing commands at lightening speed. The illusory continuity experienced in the film theater or on TV now becomes easily modifiable, so that the interaction between the program and the user becomes possible. Yet, the nature of the writing which holds the static representation is very different from the dynamic "speech acts" of the processor executing operations. The gap between static writing and dynamic "speech acts" is a transformation fundamental to the software engineering discipline. Thus, it is important for us to understand the differences, now widely recognized, between speech and writing. These differences in static and dynamic representations, and their interaction, cause many of the problems recognized as crucial in philosophy of science.

38. Jacques Derrida, Writing and Difference (Chicago: Chicago U.P., 1978); Margins of Philosophy (Chicago: Chicago U.P., 1972); Speech and Phenomena (Evanston IL: Northwestern U.P., 1973)

For instance, until Karl Popper[39] everyone thought the relation between theory, hypothesis, and observation was straightforward. Popper then showed that theories could never be proved, but only disproved. This caused a major crisis within the scientific community. It meant that science could never ultimately know the truth which it had set out to know. It meant that all undisproven theories were possibly valid. It also made more stringent what could be called a proper theory. Theories had to be falsifiable in order to be considered scientific. All parts of theories that were not falsifiable were not theory but suddenly philosophy. This made a lot of physical science suddenly disappear into another discipline. Popper still believed, however, that it was possible to have a logic of scientific discovery which depended on systematic disproving and disqualification of theories. This assumption of Karl Popper was challenged by I. Laktos[40] and P. Feyerabend[41], who showed that all methods were only backtracking. How a particular theory was arrived at was arbitrary. Good theories were never achieved historically by applying

39. Logic of Scientific Discovery (NY: Harper & Row)
40. Proofs and Refutations (Cambridge U.P., 1976)
41. Against Method (London: New Library Books - Atlantic Highlands Humanities Press, 1975)

methods. Systematic application of methods, almost always, resulted in bad theories. Methods were only there to help others get to the same view of things the discoverer had achieved by unknown or idiosyncratic means. Thus, science not only could not know what the truth was, but did not know how to get where it was going in a rigorous and well-disciplined way.

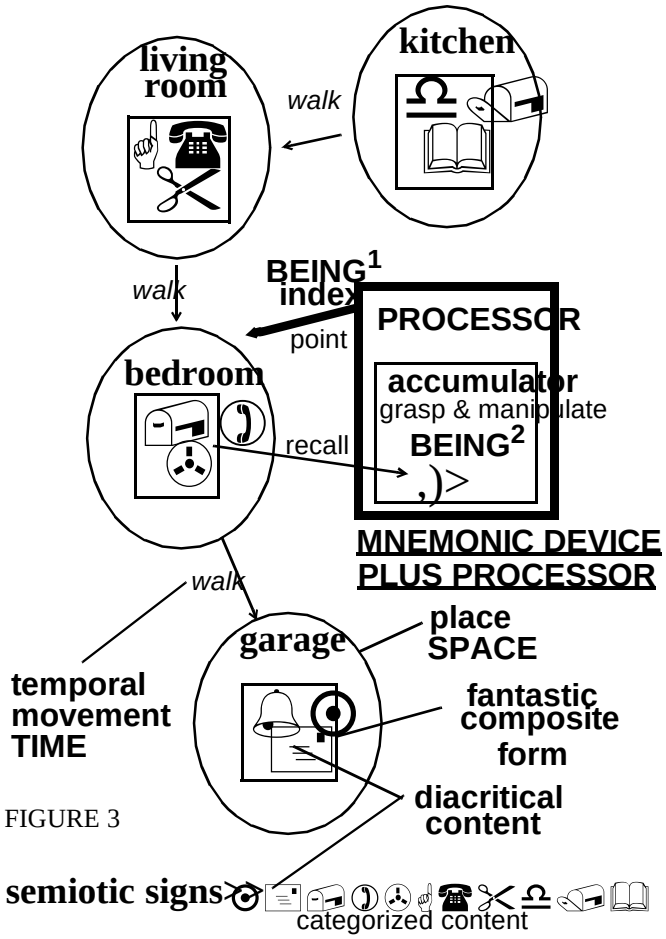


FIGURE 3

The crowning blow to the classical view of science was Kuhn's concept that science was based on paradigms--meta-theories--which periodically undergo radical transformations. Why these transformations occur, and when they will occur, is only understandable by hindsight. The most significant study of these radical transformations is G.H. Mead's The Philosophy of the Present[42] in which they are called emergent events[43]. What we experience is a build-up of accumulated anomalies unexplained by current theories. Elaborate work-arounds are constructed by normal science to explain and avoid these anomalies. Then at some point, a new paradigm coalesces in the mind of one or more individuals who are working at the cutting edge within their discipline. This paradigm shift radically alters our understanding of the field within which all the theories operate. A scientific revolution causes the whole world from the point of view of the discipline in question to change. These revolutions always accent the relation between observations and theories and cause that relation to change fundamentally. This is because our observations lead us to embrace

42. (La Salle IL: Open Court Pub. Co., 1959)
43. See the Ph. D. dissertation of the author titled The Structure Of Theoretical Systems In Relation To Emergence, London School of Economics, University of London, United Kingdom, 1982.

theories which we would not arrive at otherwise. Not only did science not know how to get where it was going systematically, it could at any moment have the rug pulled out from under it. Shockingly we discover that normal science works completely different than the aspect of science that allows progress to be made.

Software engineering theory has many of these same features exhibited by scientific theories. Software defects, which cause observed behavior to differ from hypothesized behavior, cannot be proven to not exist in a given piece of software. Mathematically, NP-completeness theorems and computational limitations provide upper limits to what is achievable in terms of proof of correctness. Proving the verification and validation of a program in relation to requirements, and many other aspects of software engineering, suffer from the same problems. Software methods function similarly to scientific methods. They are good as explanatory tools; but good designs do not come from the slavish use of methods. In fact, the use of methods can be a positive hindrance to the solution of a highly constrained, "wicked"[44] problem. We must remember that

good designers rely on experience, and rules of thumb derived from experience, to develop a software theory. How they arrive at the theory is an aspect of their craft which is not under the auspices of method. Method is an auxiliary tool that provides a language, or set of languages, to understand and communicate the problem and potential solutions to others.

Finally, although it has not been generally recognized yet, it is clear that software theories, and their associated methodical representations, are based on underlying sets of assumptions which may be called paradigms. The standard computational paradigm is that developed by von Neuman and defined by Turing. Examples of nonstandard paradigms are cellular automata, neural nets, massively parallel computers with actors, expert systems. These computational paradigms cause the software theorist to have radically different approaches to designing and coding software. Within the standard software engineering arena, as opposed to computer science where most of these alternative computational paradigms live, there are also "software centered" methodological paradigms

44. Simon, H. A.; "The Structure of Ill-structured Problems" in Artificial Intelligence; Volume 4 (1973), pp. 181-201

that are developing concerning how a network of von Neuman machines should be programmed. These “software centered” software engineering paradigms are still, at this stage, mostly unconscious, i.e. unarticulated. Although software engineering is evolving rapidly with the addition of networks, processors are still thought of individually and programmed as isolated units. The software centered paradigms are based on those structured programming constructs developed in computer science. The first major indication that other software paradigms are possible is the rise of object-oriented programming to challenge functional programming. We are now dealing with the trade-offs and attempting to understand the differences between theses two approaches. However, this change is probably a small tremor compared to the paradigm shifts to come.

These strange aspects of software engineering which appear so similar to scientific theorizing are intimately connected to software’s relation to writing. It is only possible for us to have theories because of written culture. Written culture allows us to formulate hypotheses and write them down for comparison to

observations. Thus, our ideas about the world are represented in a static picture that is inspected after the experiment has been set up and run. Theories exist in a special space made possible within our culture by writing. Theories are, in fact, a special case of the deferred meaning of the text. The written observations appended to the theoretical text and the test descriptions give the theoretical text a different meaning. In software these relations between writing and theorizing are accentuated even more than in standard physical science. The pure difference of the software text is exhibited in its intrinsic unreadableness. The text is a string of commands to a processor that is not present. One must defer finding out what the significance of the text is until it is executed. In writing software texts, we are confronted by the utter alienness of writing in a way that prose texts never achieve. This utter alienness and difficulty of understanding flows from the fact that the meaning of the text is the actions performed upon its commands. These actions are mainly invisible. We read prose for pleasure. Code representations are never read for pleasure. Pleasure only comes when the code is executing properly. The limited visible

outcome of the performed text is an extreme form of delayed meaning. The action controlled by the program executing on an automata is a special kind of speech. The transformation across the gap from static to dynamic, from writing to speech, is also a transformation from our programming words to the actions of the slave-other. Our programming words have no intrinsic meaning as the meaning of prose texts have for us. Our programming words are in a language of action whose meaning is only apparent once the radical barrier between static and dynamic has been passed.

We do not think about these underlying structures within which we work every day. But it is precisely these rigid structures which have never been confronted by human beings before, that we must understand if we are to comprehend what software engineering is really about. The fundamental concepts of how formal and structural systems work have been knit together in a unique configuration in the computing environment. These concepts have direct links to some sophisticated philosophical ideas recently developed within our Western scientific tradition. For instance,

the preeminent philosopher of our time is Martin Heidegger. His work Being and Time[45]is probably the greatest work in philosophy since Kant's Critique of Pure Reason[46]. Heidegger presents in that book a fundamental reorientation of Western philosophy from that given it by Kant. From one point of view this book written in the 1920s could be seen as the fundamental statement of a philosophical basis for computer science[47]. Heidegger's work has as a central focus the relation between man and technology. In computer science the expression of this relation is extremely intense. Therefore, it is interesting to note that in this book the major concepts developed are isomorphic with the basic model of computer hardware structures. Looked at from the perspective of computer science, definite mappings between computational structures and philosophical concepts may be seen. Space is equivalent to memory. Time is represented by CPU cycles. And the key thing that makes the CPU work is the difference between accumulators and index registers. This difference between types of CPU registers appears, strangely enough, also in another form

45. (NY: Harper & Row, 1962)
46. (NY: St. Martin's Press, 1965; Macmillan & Co. 1929)
47. Winograd, T. & Flores, F., Understanding Computers & Cognition (NY: Addison Wesley, 1986)

as a fundamental distinction in Heidegger's thought. Heidegger tells us that Kant thought there was only one kind of Being, i.e. Pure Presence. He distinguishes between Kant's kind of Being and a new kind he describes as mixed with Time, dynamic, instead of static. Being is the most general philosophical category signifying the ontological status of "remaining present". As software engineers, we should be interested in this most abstract of all categories, because the ontological status of our product is questionable, i.e. it is merely a theory. We have serious problems presenting these theories in our representations. The science of Being, i.e. Ontology, helps to explain why we are having these problems that other branches of science have run into before us, though perhaps not in such extreme forms.

"Being" is whatever is verifiably present. Each entity in existence has Being as its most basic attribute. Heidegger makes the point that Being, as an idea is not contained in any individual entity but is a field that embraces all entities. This concept he calls ontological difference[48]--it is the difference between the being of the entity itself and the field of Being

48. Vail, M.; Heidegger and Ontological Difference (London: Penn. State U. P., 1972)

in general shared by all entities. The human being is a special kind of entity that projects Being on other things in existence. At least Indo-European cultures traditionally project Being as the basis of the world. It is unclear whether other human cultures do this. This process of projecting Being is called "Dasein" which means "being-there" interpreted as being-in-the-world. Projecting Being on things is in Heidegger's view intimately connected with the creation of our world by ourselves. Technology plays a large part in this process of world building which is one aspect of the social construction of reality[49]. Heidegger points out that there are at least two kinds of Being, instead of only the one, the static kind, traditionally identified since Parmenides[50]. The traditional static sort of Being will be identified with a superscript as "Being^{1}". The other sort of Being, "Being^{2}", he claims, is hidden but ever present. It is seen when the traditional category of Being^{1} is seen in relation to the category of Time. Being^{1} mixed with time produces Being^{2}. A.N. Whitehead's Process Philosophy[51] explores this same realm of ideas. Whenever what is important is not

49. See Peter Berger & T. Luckmann; The Social Construction of Reality (NY: Double Day, Anchor, 1967)
50. This distinction between the philosophies of the two Pre-Socratics is a generalization for illustration purposes.
51. Process and Reality (NY: The Free Press, Macmillan Pub. Co., 1978)

what remains present but the flux of existence itself, then Being^2 must be considered. This is Heraclitus' perspective on existence. The human being relates to existence either from the perspective of Heraclitus or that of Parmenides. These represent age old modalities for humans to relate to things. Until recently, since things have sped up in our civilization, we were mostly interested in what remained the same. Now, however, the flux of existence itself is becoming more important to us in our everyday lives. Thus, each modality is becoming important to us, and we are often having to switch rapidly between these different perceptual modalities.

Heidegger calls the modality associated with the second kind of Being the "Ready-to-hand" as against the modality related to the first kind of Being called "Present-at-hand". He says these are two completely different types of modalities by which human beings, considered as processes, relate to entities within the world. This second kind of Being has a different criteria of truth. Instead of verification as its criteria of truth, Being^2 has as its criteria of truth whether the thing can be re-presented or not. Presentation is a constant process of re-

showing, or revealing, that entails showing and hiding. Things that can be re-shown over and over have a different kind of existence than those which cannot. For Heidegger, this second modality of Being^2 is the modality which all technology inhabits. Technology is primarily a mechanism for controlling showing and hiding. Tools are ready-to-hand and disappear from presence as they are being used to "work on" what is present-at-hand. Technology hides itself, so that something else can be shown. What is shown is the center of attention, the thing being verified over and over. Technology allows this center of attention to change by making it possible to replace the figure with a new figure from out of the background. Technology controls the coherence of the Gestalt. Like the Gestalt, it is hidden in the background. It is only when the technology breaks down that Being^2 is brought into focus and becomes a figure.

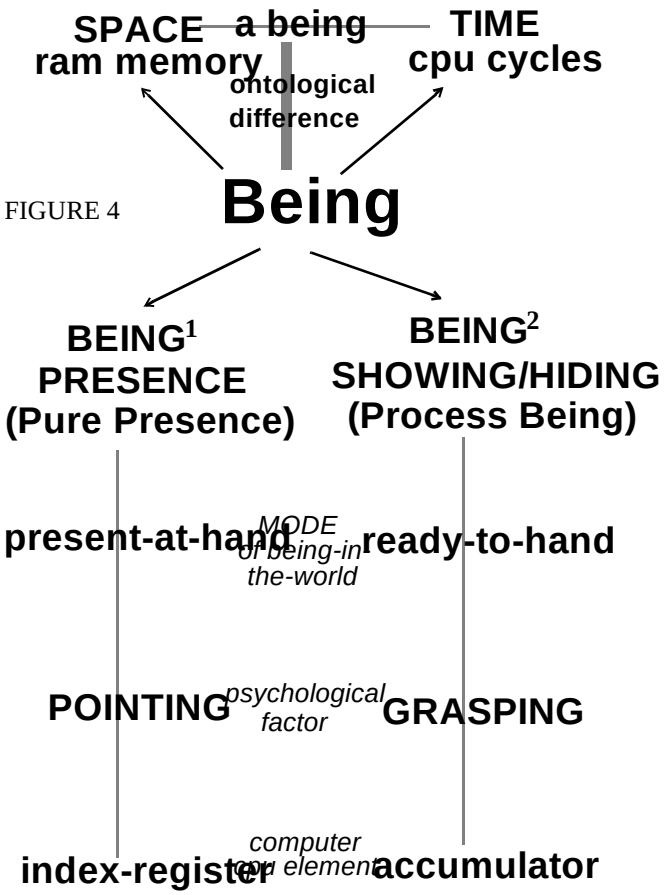


FIGURE 4

Maurice Merleau-Ponty, a French philosopher, wrote a companion book to Being and Time[52] called The Phenomenology of Perception[53]. This book showed the roots of the distinction between present-at-hand and ready-to-hand in human psychology as revealed by experiments on brain-damaged people. In that book, Merleau-Ponty shows that "Pointing" and "Grasping" in the psychological make-up of human beings have exactly the same attributes

52. M. Heidegger, op. cit.
53. (Evanston, IL: Northwestern U.P., 1966)

as the abstract philosophical concepts of Heidegger. This is important to us here because of the analogy we have drawn with computer science structures such as accumulators in artificial processors that grasp and manipulate values and index-registers that point at memory locations. Accumulators have a specific relation to transformations in time, whereas index-registers have a similar relation to positions in the space of the computer memory. Working together, they allow data to be manipulated in such a way as to constitute an independent processor. For software the hardware platform, which acts as an independent processor, provides the mechanism for showing/hiding. The purpose of the software is to control what is presented within the arena made possible by the hardware. The technology that makes software possible calls for a meta-technology that controls the showing and hiding process embedded in the hardware. It is the nature of this meta-technology that we must explore in order to understand the real meaning of software[54].

54. An in-depth analysis of meta-technology and its difference from just plain technology is contained in the second part of this series of essays.

Software needs this independent processor to exist. Software seems to be all purely present-at-hand as a pure plenum of text. We like to think of this text as if it were all perfectly observable at one time, as open to an ideal inspection that could encompass the whole. This is, of course, impossible. We can only inspect a small part of the whole at any one time because of human limitations. Software, when compiled, presents the processor sequentially with the operations to perform. When compilation occurs, we no longer are able to inspect the software. Machine code at the binary level is notoriously difficult to comprehend. Human beings cannot easily understand that magnitude of diverse detailed intrinsically meaningless data. Thus, when software is compiled, it becomes even more hidden. The performance of sequential machine level instructions appears as an illusory continuity of actions in time. When the software disappears as presentable text and is compiled, it becomes purely ready-to-hand. When it is doing its work properly, it is not seen itself. Only the images and sounds that it causes are made present-at-hand. Yet, when the software has defects, it becomes the focus of our attention and is rendered present-at-hand

itself. It ceases to be able to hide itself and show us something else. We relate to computer programs through our ability to point and grasp. For instance, when writing we grasp the pen which points to what we are writing. What we are saying in prose has our full attention, and we do not even notice the pen in our hand or our movement of it. This ability to do one thing while focusing on something else is what allows us to program computers whether we write code by hand or use a key-board. There is a radical discontinuity between words and action which has been appropriated by the computer technology. This strange relation between words and actions in relation to all processors, human or automata, is what we use every day without thinking twice. The secret of software engineering appears in this close juxtaposition of words and actions, and the differentiation of words into spoken and written.

Another point worth mentioning is that the model upon which the computer is built is the mnemonic techniques developed during the Renaissance. Francis Yates in the Art of Memory[55] describes these techniques in their

55. (London: Routledge, Kegan, Paul, 1966)

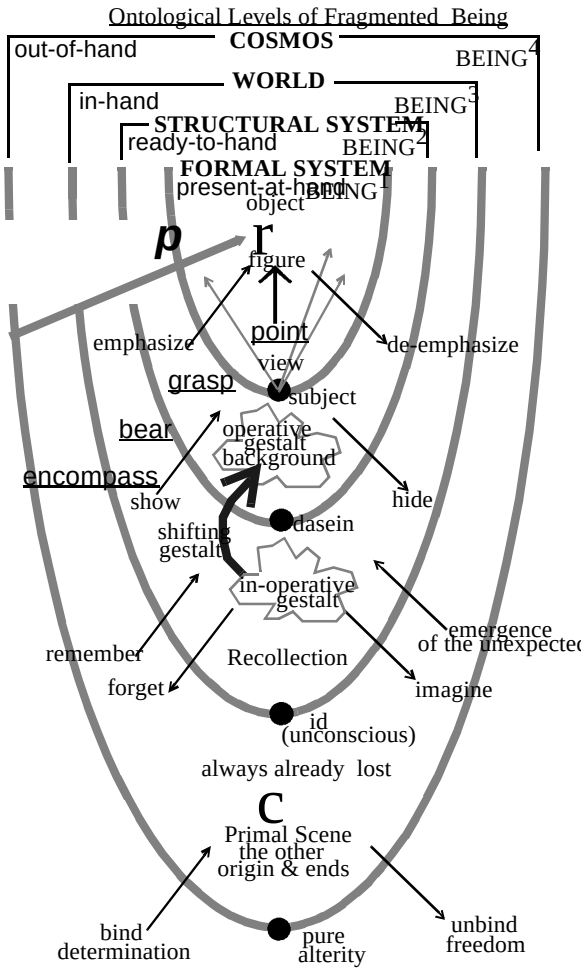


FIGURE 5

early forms. Essentially, the mnemonic device was a series of places like the rooms of a familiar house. In each place was mentally placed a fantastic object associated with and representing a fact that the person wanted to remember. The person would later mentally simulate a walk through the house saying, “what did I put in this room?” Amazingly an

individual's memory would yield up the secret he had hidden in each place as a recollection. A large number of facts could be memorized in this fashion. The structure of computers echoes this mnemonic model. The memory is an array of byte locations. Either the program counter or a moved index register allows the processor to walk through the rooms of the house to remember what was stored there. The things stored in those locations are fantastic objects made up of semiotic codes ordered by syntax. The codes and syntax allow a composite object to be built up of other objects or a parts of other objects. The composite figure placed in the memory location is the first glimmering of structuralism. Forms are made up of contents, which are merely rearranged contents taken from a whole set of similar objects, that are connected in a semiotic system. The mnemonic device is the first form of a showing/hiding mechanism designed to facilitate recollection by the human processor. It was also used for mental simulation. In the computer this mental simulation has been refined and externalized, but the structure is fundamentally the same.

The automation of the mnemonic device

through the use of pointing and grasping gave a concrete realization of the structural system as an independent processing machine. The differentiation of formal and structural systems is important for the understanding of the development of computer technology. The best example of a formal system is G. Spencer-Brown's Laws of Form[56]. A formal system concerns the rules for the generation of forms or figures. Geometry is the classical formal system. From a few axioms a myriad of forms are defined and manipulated. They have formal static relations between each other which can be described by theorems with logical proofs. Formal systems allow verification of things not immediately self evident. Formal systems, such as symbolic logic and most of mathematics, make strict verification possible. They extend the range of what can be verified beyond the immediately present and allow a system of links which make it possible to retrace our steps to a previous position. It is formal systems that make methods possible. A method is the application of a formal system to a problem to produce a representation of the problem. The formal system is independent of the problem space. It

56. (London: Allen & Unwin, 1969)

is used as a tool for representing and then verifying formal representations. Forms are represented using rules of formal systems to govern their composition and changes from one representation to the next.

A structural system has a different purpose from a formal system. Structural systems arise at the limits of formal systems. Formal systems do not deal with the action of time. (Spencer-Brown stops the development of his formal system at exactly the point where time is introduced.) They express purely spatial or logically static relations between elements and are used for navigating the spaces inhabited by forms. Formal systems reach their limits when time is introduced into the system being verified. Structural systems are specifically designed to handle the case of temporal transformations. Good examples of structural systems are Transformational Grammar developed by N. Chomsky[57] in linguistics, the structural anthropology of C. Levi-Strauss[58], and J. Monod's genetic theory of evolution in biology[59]. Also Heidegger's theory of Hermeneutics in Being and Time[60] evokes the

57. Cartesian Linguistics (U. P. of America, 1983)
58. The Savage Mind (London: Weidenfeld & Nicholson, 1966); Structural Anthropology (NY: Basic Books, 1963)
59. Chance & Necessity (London: Collins-Fontana, 1974)
60. (London: Harper & Row, 1962); See also Gadamer, H.J., Truth & Method (NY: Crossroads, 1991)

same general conceptual structure. In General Systems Theory the best example of a structural system is the "General Systems Problem Solver" work of G. Klir as seen in his book Architecture of Systems Problem Solving[61]. To formal systems temporal changes appear as discontinuities within the system space. Structural systems allow the construction of bridges across these discontinuities. The best example of this is how atomic theory is a structural bridge across the discontinuity witnessed at the macro level in chemical reactions. Physics has been very successful at applying structural models to the description of nature at many levels. The key feature of a structural model is that the content of the forms are analyzed and specified by a semiotic system such that any desired form may be composed by a specified content substitution. Structural systems govern the transformation and transduction (change of media as well as content) of forms across temporal discontinuities.

The ASCII code is an excellent example of the kind of coding that is necessary to control the different letter forms. The transformation form

61. (NY: Plenum Press, 1985)

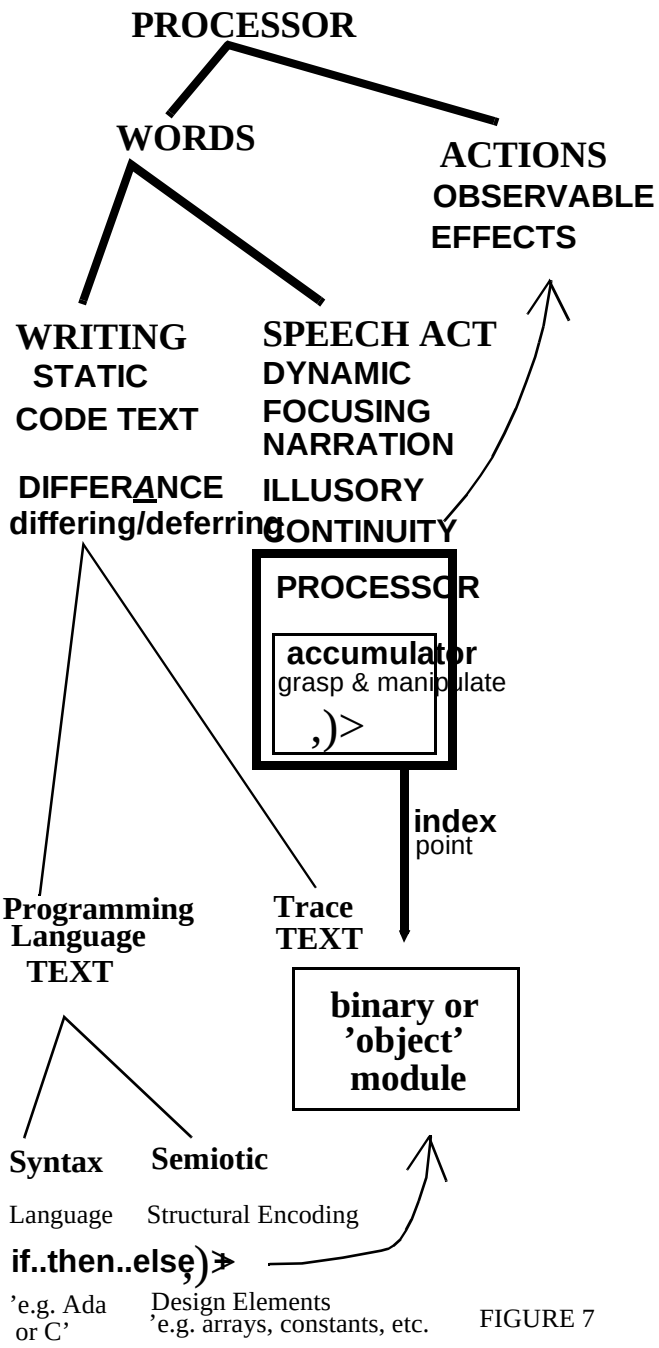


FIGURE 7

one from to an other is made by changing a given content bit. By changing regularized content codes, transformations from one form to another are made possible. The fantastic

figures planted in the mnemonic landscape (like the house layout) are structural in intent. They are pieces of different forms that are combined arbitrarily to create these fantastic figures. These pieces of content could be rearranged at will to transform one fantastic figure into another. Pointing to the memory location, we grasp its contents. Those contents are a figure made up of visual codes. The fantastic figure indicates the remembered fact by an established association, just as the structural system of software methods indicates the non-representable software theory. The structured form of the figure has a direct relation to the represented fact or idea which is to be recollected. By speeding up our movement through memory, and then executing the instructions found in a series of memory locations which modify forms by manipulating their semiotic infra-structures, and then by displaying the figures, we get a moving picture or a computer program. Within the computer the structural model has been animated and is executed at a speed greater than the eye can catch the discontinuities between formal presentations.

Software engineering mimics these same formal and structural underpinnings in the software we build. Using mathematics, logic, structural programming constructs, boolean operations we construct formal representations. These representations, however, are animated, and we use structural transformations of data to model changes required in the systems outputs. All the structural transformations of formally defined objects rely on coding of forms and the embedding of knowledge within the application. Thus, we are so immersed in formalism and structuralism that they are invisible to us like the air we breathe. We do not see them. They are too close to us. But we need to see the technological infra-structure of these tools and artifacts that engulf us, because something strange is happening while we are too preoccupied with our own productivity to notice. There are not just two types of Being. In fact, the concept of Being is rapidly fragmenting. M. Merleau-Ponty, J.P. Sartre, M. Heidegger, and others have discovered and described at least two other kinds of Being. There is a good possibility that the type of Being that software primarily exhibits is neither ready-to-hand nor present-at-hand. These are, in fact, the types of Being related to

hardware. This means software cannot really be described accurately, either as just something static, or as a merely a process. We need some other way of describing software because it is intrinsically based on a completely different type of Being. Thus, when we use formal and structural systems to build our software, it is possible that we have not realized what software really is all about. We are still just imitating the hardware technological infra-structure in the software systems we now build.

I will attempt to briefly sketch the way I see the relation between the different types of Being that seem to exist and their possible significance for software engineering. We have already described present-at-hand and ready-to-hand. I will continue to develop this theme using examples from mathematics. The type of mathematics related to the present-at-hand is Calculus. This is no surprise since Kant's philosophical system is intimately related to the physics of Newton and is, in fact, an idealization of the Calculus. This transformation of the calculus into a philosophical system, grounding all formal and structural systems, is a very significant event in Western history. Since this event we have been

working out its implications in the technological systems we have developed. Calculus allows the mechanism for constructing artificial continuities to be displayed for the first time. Ideas are forms given this attribute of illusory continuity generated by this mechanism called ideation. The structure of Kant's philosophy uses this mechanism to ground all formal systems. It makes possible structural systems because the discontinuities between transformations of forms are reduced via step-wise reduction taken to infinity. Once the calculus was thus embedded as the underlying structure for all ideation, a brilliant intellectual ploy, the stage was set for describing the whole of existence in terms of continuous functions that allow the mapping of forms in a very sophisticated manner. Calculus is totally determinate. It, and the illusory continuity of speech that it mimics, is used to present forms continuously in a seamless web of interlocking formal systems. From the point of view of determinism the world a mechanical clockwork that gives a continuous sweep to the hands of the clock This gives us the impression of the regularization and objectification of time. Besides its seeming mechanical perfection, this

deterministic system was hermetically sealed. Existence was locked out.

Strange as it may seem, this perfect world really attempted to, but ultimately could not, last eternally. There was no real relation between the perfect world of pure presentation of ideal forms and the observed world. Statistics then entered the picture. Heidegger's philosophy may be seen as doing for statistics what Kant did for calculus. The ready-to-hand is the modality of human relation to the world that gives statistics its foothold. It was noticed that identical acts had slightly varying results. Thus, forms did not move through the world flawlessly as functions suggested. A trajectory could be calculated precisely, but multiple shots would all be slightly different, varying according to a normal distribution. Actualizations of ideal forms are scattered in predictable patterns. Thus, although there were flaws in the relation between the pure world of ideas and the actual observed world, the differences were predictable. All technology is based on its imperfect, but predictable, mapping. The process of showing an ideal form repeatedly encountered resistance from the material world. That resistance reminded

us that the material substrate was there. At first this seemed like a minor side track to the program of constructing an ideal world of representations which controlled every aspect of the material world. However, as physics progressed, it found many strange phenomena at the limits of our abilities to grasp nature. These strange phenomena coalesced into quantum mechanics which perfectly exemplified statistical representations but challenge our abilities to make ideal representations that made sense. In fact, we still have no coherent explanation for these phenomena, even though we have found ways of dealing with the existence of these anomalies. There is a gigantic structural rift between the kind of idealizing determinate representations of the world fostered on the macro scale by Einstein that do not connect with the statistically correct but indeterminate quantum world. Ultimately the mechanism that underlies statistical variation in the universe is unknowable in any determinate way. Determinateness is the very essence of the present-at-hand. Determinate means are fully presentable in all its aspects. Thus, the statistically predictable yet intrinsically indeterminate falls outside this kind of truth.

The indeterminate has a different kind of truth. Its truth has to do with the process of repeated showing of the same thing--which is exactly what is needed for the act of verification to occur--and what it is that renders indeterminate is ultimately hidden. We see the individual actualizations as figures on a ground. We cannot see the actual process that causes the variation. This is particularly disquieting in the case of the double slit experiment.

Thus, it is proposed here that the two kinds of Being that Heidegger defines are, in fact, real phenomena with the psychological components of pointing and grasping described by Merleau-Ponty; and that these connect directly to the two major kinds of mathematics that engineers the world over use every day to make technology and physics happen. The idea that one is switching between different kinds of Being when using these two kinds of math does not have to be strange. Reality has multiple aspects as shown by the fact that truth is different when looked at in these different ways. We already know that reality is multifaceted. The fact that we have developed different kinds of mathematics to deal with its

different aspects should not be surprising. What is surprising is that observed nature should carry its adherence to the mathematical rigors suggested by these different kinds of being to such fanatical extremes. The key thing is not to get mixed up and apply the criteria of one kind of truth to the wrong aspect of reality. This, of course, happens all the time, since most of us operate as if there were only present-at-hand beings; the news of the discovery of the ready-to-hand has not filtered down to everyone yet, even though we deal with it every day. This is particularly the case in software engineering where we attempt to construct very large formally defined computer systems without realizing that once past a certain threshold (one screen full of text) the showing of the whole program at once is impossible. Immediately we are dealing with problems of showing and hiding, and are constantly switching back and forth between what is the immediately presented figure and another realm of Being that controls recollection. This other realm of Being gives the system we are dealing with a subtle partially uncontrollable aspect. There is always some part of the system hidden. We cannot make the whole thing visible, no matter what

we do, so we need to choose appropriate abstractions to represent the system. This inherently hidden character of a certain aspect of the software system is one of the things that makes the software product a non-representable theory. What is rendered visible is a portion of an abstract representation. Abstracting still hides some things while making visible other things. The larger the system, the more of a problem this becomes, because it soon becomes impossible for one human being to handle, and then teams working in close coordination become necessary. Software graduates into the realm of the social and organizational systems. This is where software engineering becomes necessary. It may be said that software engineering comes into being to handle the new modes of Being encountered in building by large systems. It is different from computer science because those systems studied by that discipline are normally small and tractable enough that the present-at-hand mode can be stuck to very closely. Software engineering comes into its own when the ready-to-hand mode must be faced directly and structural techniques developed for bridging the gaps caused by extreme showing and hiding problems. The software theory is totally

obscured in software engineering, whereas in computer science there seems to be a means of making it at least partially visible.

All of this is relatively sure ground, both ontologically, mathematically and in terms of software concepts. Now we enter a more hazy area were I will attempt to explain my view of how the other kinds of Being enter the software picture. My premise is that, although both ready-to-hand and present-at-hand effect software engineering, we are actually dealing primarily with yet another kind of Being which has not displayed itself fully as yet. These other kinds of Being are not as widely accepted as the previously discussed pair, so we must proceed slowly and carefully, continuing our mathematical analogies.

Each of these kinds of "Being" operate as if they are at a higher meta-level than the last kind of Being. The ready-to-hand is at a meta-level from the point of view of the theory of logical typing, as described most succinctly by I.M. Copi[62], that was inaugurated by Russell and Whitehead[63] to solve logical paradoxes.

62. Theory of Logical Types (London: Routledge, Kegan. Paul, 1971)
63. See their Principia Mathmatica (Cambridge U.P., 1925-27)

We are familiar with meta-levels from physics. For instance, we define stillness, motion, acceleration, and jerkiness as meta-levels of physical movement in relation to each other. The interesting thing is that we cannot "think" what is at the next meta-level, i.e. acceleration^{3} (acceleration of acceleration of acceleration). It is impossible to think any higher concept than acceleration^{2}. It is my hypothesis that it is exactly this kind of theoretical structure that is involved with the articulation of the different kinds of Being. There are only four kinds of Being, and we cannot think beyond the level Being^{4} [64]. This sets formal limits to the possibilities of the fragmentation in Western ontology. Being^{zero} is the most general characteristic shared by all entities which is their existence as a being among other beings. This characteristic is fundamentally related to the possibility of the presence of that entity within the field of consciousness of a human being. Unless being is related to presence, it is an empty conceptual construct with no real meaning. Being^{zero} is identified with the "being", with a small "b", that is an attribute of every entity. Being^{1} is the

64. I posit that what is beyond Being^{4} is Emptiness or Void in the sense propounded by the Buddhists were the concept of Emptiness is itself completely empty. So it is, in fact, possible to speculate on what is "beyond" all the meta-levels of Being but that takes outside the Western tradition.

present-at-hand formally called Pure Presence. It is what may appear clearly and distinctly, to quote Descartes, within the center of consciousness as a figure or center of focus. Being[1] is conferred upon whatever is the figure in any Gestalt as the center of attention. Being[2] is the ready-to-hand which is known as Process Being because it is a combination of Being[1] and Time. Being[2] entails the presentation of what is present and is conferred upon whatever is the Gestalt of a figure/ground relation. Being[3] will be identified as the "in-hand" known as either "~~Being~~ (crossed out)" by Heidegger[65] or "Hyper-Being" by Merleau-Ponty[66]. The term "in-hand" is used to describe how tools transform in our hands, and also transform our hands, giving us new and different ways of grasping the world. ~~Being~~[3] entails the offering of the presentation which occurs because of the differentiation between transcendence and immanence. ~~Being~~[3] is conferred upon the relation between what appears in the Gestalt and what is truly unconscious so that it never appears directly in a Gestalt. What never appears is truly immanent, while what appears is transcendent,

65. The Question of Being (London: Vision Press Ltd., 1956)
66. The Visible and the Invisible (Evanston IL: Northwestern U.P., 1968)

either as Being[1] or Being[2]. ~~Being~~[3] makes itself felt through the changes in the patterning of the Gestalt. Over time the patterning of the gestalt as a whole will change, usually discontinuously, so that different relations will exist between the possible forms which may be made figures in relation to the background. Kuhn's concept of scientific revolutions, which change the whole field in which theories operate, is an example of this phenomenon. In philosophy it has been noticed that the concept of Being has historically changed from time to time in the history of Western philosophy. Heidegger has called this phenomena the "epochs of Being"[67]. Foucault in The Order of Things[68] has identified what he calls "Epistemes" which, in various eras, have conditioned our understandings of the world. All of these examples display how the whole world can be transformed in a way recently portrayed very well by James Burke in The Day The Universe Changed[69]. The true nature of the "cutting edge" of any discipline or genuine nature of "Progress" must be based on ~~Being~~[3]. Also, ~~Being~~[3] has been discovered to be the formal cancellation or annihilation of

67. The End of Philosophy (London & NY: Harper & Row, 1973)
68. (London: Travistock, 1970)
69. (Boston: Little, Brown, & Co., 1985)

Being^{2} and its antithetical opposite, Nothingness as defined by J.P. Sartre[70]. E. Levinas[71] speaks of the human psychological referent of this level of Being as "bearing" related to the other psychological referents of the lower meta-levels identified by Merleau-Ponty as pointing and grasping.

Being^{4} will be called the "out-of-hand" modality named Wild Being by Merleau-Ponty. Being^{4} entails our relation with genuine impossibility, and thus expresses the reciprocality between limitation and freedom in our relation to the world. The patterning of the Gestalt may change periodically due to unconscious factors, but there are limitations to what changes are ultimately possible and what are impossible. The fundamental relation between our freedom and our determination is expressed in the idea of Being^{4}. Being^{4} is conferred upon the set of changes that occur to the Gestalt as it precesses through the deep structural changes caused by unconscious or immanent factors. In a dynamical system this appears as a series of ground states or regimes into which the system settles which alternate

70. See Being and Nothingness (London: Methuen & Co. 1958)
71. Otherwise than Being or Beyond Essence (The Hague: Nihjhoff, 1981); See also Totality and Infinity (Pittsburgh, PA: Duquesne U.P., 1969)

from time to time. These ground states represent the limitation of the possible changes in the Gestalt patterning. The progressive bifurcation of non-linear dynamical systems on the way to a degenerate chaotic state defines this series of ground states. The unlimited possibilities open to the dynamical system are structured into a highly ordered subset. In a sense Being^4 represents a contraction that balances the expansion represented by ~~Being~~^{3}. The psychological referent of this last level is "encompassing." When things get completely out of hand, one is overwhelmed and utterly encompassed.

An example of how these four meta-levels of Being work together can be given from physics. In physics there is the idea of virtual particles. These virtual particles manifest as opposites, and then they destroy each other again before the laws of conservation are disturbed. All space-time is thought to be a foam of virtual particles being created and annihilated. Occasionally a virtual particle escapes its annihilation, for instance at the event horizon of a black-hole, and becomes a real particle so that there can be some crossover between the vast potential energy of space-time

and normal matter. But this is a rare event. The intriguing thing about the concept of virtual particles is that in this theory all the elements of the fragmentation of the concept of Being observed in modern ontology are present. It is a theory that appears to be meant to display this fragmentation as it operates within the arena of physical science. Briefly, we can say that the two "laves"[72] (wavicles, i.e. particle/waves) that are produced are antithetical opposites. They each exist as both particle and wave. As waves they are probabilistic, while as particles they display discrete quantal characteristics. The discrete/ non-discrete natures are complementary dual natures of the same entity. Through these natures the intrinsic complementarity of Process Being and Pure Presence is displayed. Because the lave appears with its anti-lave, for example an electron/positron pair, annihilation will occur. The particle/anti-particle relation between the two laves expresses in their nature the possibility of ~~Being~~^{3} or the cancellation exhibited by Hyper-Being. There are myriad possibilities of particle/anti-particle pairs which may be realized in a given manifestation of a given virtual particle pair in spacetime. These

72. See Wilczek, F. & Devine, B.; Longing for the Harmonies (NY: W.W.Norton, 1988)

possibilities will all lead to annihilation, and thus uphold the law of conservation. The relation between the myriad possibilities and the law of conservation displays the relation between transcendence and immanence. The freedom of unlimited production of sets of particle/anti-particle pairs contrasts with the hidden immanent constraint that maintains the law of conservation. Finally, which particular pair of laves is actualized at a given point in space-time, out of all the possibilities, is a matter of the conversion of possibility into probability. This displays the chaotic nature of propensity that drives this transformation. In the epiphany of the particles we see the action of Being^{4} or Wild Being. ~~Being~~^{3} and Being^{4} display a similar intrinsic opposition similar to that of Being^{1} and Being^{2}. All four types of Being work together to make the virtual particles manifestation in Being as beings possible. A similar thing happens at all levels of existence. The four kinds of Being work together whenever the manifestation of anything occurs. In the manifestation of technical beings this process is particularly apparent, because the different kinds of beings are conditioned by the nature of the technological system.

Each kind of Being in this series is a meta-level in relation to the last with a concomitant refinement of meaning. For instance, in the same vein, G. Bateson describes meta-levels for "learning". We all know what learning is. It is the acquisition of knowledge. Building software entails learning how to fit the design to the requirements constraints in the best way. Learning to learn (learning^2) means finding out how to go about learning and getting better at it. It means rendering the process of learning visible and changing it. In software this occurs when new methods are introduced into the development process. Learning to learn to learn (learning^3) is more difficult to think about. It means something like changing ones way of learning to learn--adapting it to circumstances and new knowledge about the learning process--getting feedback from previous attempts to learn how to learn and making appropriate changes that show how one has learned to learn learning better. In software this means switching to a new software-centered paradigm. The shift from the currently posed paradigm to another one at the same level of abstraction would be an example of this kind of learning^3. One is not just adopting a new technique useful for

learning how a software system works or needs to work. Instead, one is shifting one's fundamental perspectives on how software methods need to be expressed. This is starting to get very difficult to express. Learning to learn to learn to learn (learning^4) is just barely comprehendible. It means something like revolutionizing one's whole worldview and self. A good example in relation to learning in everyday life might be the Zen Buddhist enlightenment. Beyond this fourth meta-level of learning it is impossible to even grasp what meta-level five might be. It is beyond human comprehension. A similar thing is happening with the various ontological categories of the fragmenting concept of Being.

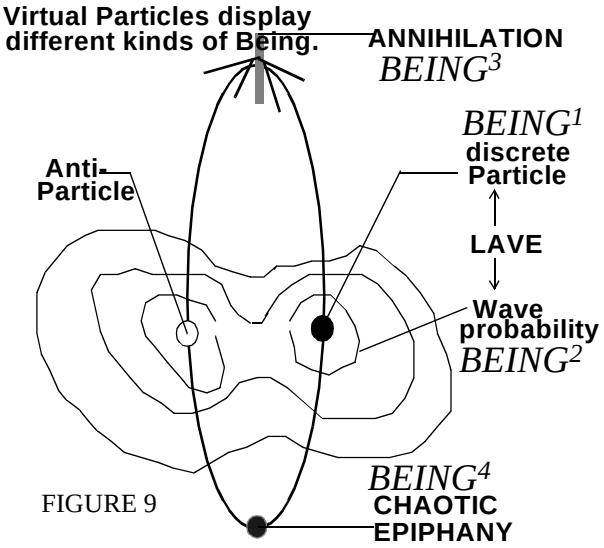


FIGURE 9

We do not need to go deeply into ontology in order to get the general idea and connect reasonably with our mathematical analogies. Being^{3} was discovered when Sartre wrote his book Being and Nothingness[73] in which he took Heidegger's notion of Process Being and turned it inside out. Nothingness was the totally antithetical opposite of Heidegger's concept of the mixture of Being and Time. After much scholarly debate, it was realized that these two concepts could not exist in the same conceptual system, and that they immediately annihilate each other like matter and anti-matter. This cancellation was realized to be another kind of Being, either Being^{2} as cancelled or Being^{2} as annihilated. It was thus given its own name and considered an ontological anomaly. Michael Henry in his work The Essence of Manifestation[74] showed that the fundamental assumption underlying all previous ontology was that of ontological monism. Ontological monism meant that Being was only defined as Transcendence, and Immanence was not considered to be an aspect of Being. It was clear that the cancellation of Being^{2} and Nothingness was the cover for pure

73. (London: Methuen, 1969)
74. (The Hague: Nijhoff, 1973)

Immanence which could never be revealed. This was a much stronger form of hiding than appeared in Heidegger's initial system. In attempting to deal with this new kind of Being, ~~Being~~^{3}, Merleau-Ponty in The Visible and Invisible[75] ran directly into the fourth kind of Being we will discuss. He called this Wild Being because it defined everything that was left after the annihilation of Being^2 and Nothingness occurred. He set out to define this final kind of Being in that unfinished book[76]. Looking at Hyper-Being and Wild Being, there is really a very simple explanation of what these esoteric terms mean. Consider a guitar player or a blind man. Each learns to use a piece of equipment to such a degree of proficiency that the equipment becomes part of their being. This expansion of the being of the person to encompass another piece of equipment has the status of ~~Being~~^{3} or Hyper-Being. This expansion is found whenever a new technical or scientific discovery expands our horizons. This is normally called the process of Technology Infusion or Transfer. There is a related contraction of being-in-the-

75. (Evanston: Northwestern U.P., 1968)

76. This unfinished work has been taken up from a new perspective by Deleuze & Guttari in Anti-Oedipus (1983) and Thousand Plateaus (1987) (both books; Minneapolis: U. Minnesota Press) their series on Capitalism and Schizophrenia. They solved the problem of how to describe cancellation within philosophy by allowing the disciplines of Marxian economics and Psychoanalysis to cancel instead of antimonies within philosophy so that philosophy was left free to describe what remains after the cancellation. See Massumi, B., A User's Guide to Capitalism and Schizophrenia (Cambridge MA: MIT, 1992); Bogue, R., Deleuze and Guttari (NY: Routledge, 1989)

world which is called Wild Being. That contraction is apparent as the inner coherence of the shifts of Gestalt formations over time. Each instrument we learn to play, or each apparatus the blind man adapts to in order to expand his horizons, further defines the being of the human in relation to his world. Thus, besides expanding our possibilities, discoveries also increase the specificity of our relation to the world, thus in a certain sense limiting us at the same time. This limitation of our freedom by the realization of possibilities is the core attribute of Being^4.

Lets go immediately to our mathematical metaphors for a more concrete explanation. The kind of mathematics associated with ~~Being~~^{3} is fuzzy sets which is called “possibility theory” and was developed initially by L. A. Zadeh[77]. The kind of mathematics associated with Wild Being^4 is Chaos[78], and is called “propensity theory” by Watanabe. These two recently developed kinds of mathematics stand out as examples of completely different ways of approaching reality with their own associated criteria for truth. Simply put,

77. "Fuzzy Sets" in Information & Control, Vol. 8, 338-353, 1965
78. Stewart, I. Does God Play Dice (Oxford: Basil Blackwell, 1989); Gleik, J., Chaos (NY: Viking, 1987)

probability theory deals only with actualizations and their after-the-fact distribution. It does not deal with the possibilities that resulted in the realized actualizations. These possibilities are endless, uncontained, and do not add up to one like probabilities. I have a myriad of possible routes home, yet my probable routes can be worked out by collecting actual routes taken. Fuzzy sets allow possibilities to be dealt with and mathematically described. They are analogous to linguistic hedges (very, sort of, etc.), and they are therefore very useful in describing situations that cannot be precisely defined. Yet, the transformation from possibility to probability is still unexplained. The transformation of possibilities into probabilities occurs through the introduction of a propensity, or tendency[79], to realize a certain possibility, converting it into an actualization. The individual actualizations are related by probability distributions. If you tried to convert the same possibility again, a distribution would occur, and the propensities would vary chaotically. The action of probability and chaos cause variations in the realization of possibilities from two independent directions.

79. See Ontological Investigations by. Johansson (London: Routledge1989)

Chaos causes inward variation in the actualization process with respect to possibilities, whereas probability causes outward variation.

Each of these kinds of mathematics are explicitly linked to automata theory by S. Watanabe in his article “Creative Learning and Propensity Automation”[80]. He develops definitions of deterministic, stochastic, and fuzzy automata and adds a definition of propensity automata to these to define the entire set. The propensity automata uses weightings on the states of the system determined by the previous history of the system to determine responses to current states. J. Monod called this a teleonomic filter[81]. By using a teleonomic filter, the range of possible system states are successively narrowed. Watanabe calls this learning. This is the phenomena exploited in a more complex manner by neural nets. What Watanabe does not do is relate propensity automata to the mathematics of Chaos[82]. Many deterministic systems have been discovered to be chaotic in certain regions. This chaos will cause the

80. IEEE SMC-5 #6 1975
81. Chance & Necessity, op. cit.
82. Watanabe, Satosi; "Creative Learning and Propensity Automation" in IEEE Transactions on Systems, Man, and Cybernetics; Volume SMC-5, No. 6, November 1975, pp. 603-609

system to have internal variation that will effect its teleonomic filter's performance. This ultimately means that the choice of possibilities is not fully deterministic. This means that in the actualization of any possibility, the propensity to realize a particular possibility cannot ever be known for sure. Propensities weight the possibilities and more or less determine which is more likely to be actualized in a given situation, but the chaotic residue hidden in deterministic systems makes this weighting ultimately non-deterministic. In different trainings the neural net weightings may end up slightly different. This is an extremely important connection which explains the variations in the actualizations of possibilities. It also means that this phenomenon of converting possibilities into probabilities will never be fully understood. Chaotic systems defy any ultimate determinate, stochastic, or fuzzy reduction. Each type of automaton has an important place in our modeling of technical beings.

Each type of automaton is connected intrinsically to a different perspective on existence by being related to a different kind of Being. Since all beings exhibit these

modalities in relation to humans, we cannot get a full picture of any system without using all of these different kinds of mathematics. This makes systems modeling an extremely complex business. The primitive state of our development of systems science flows from the lack of recognition of the importance of these ontological categories in our everyday practice of technological arts. We treat everything as if it were possible to make it present-at-hand. Because of this, we run into tremendous problems, especially in fields like software engineering where we are essentially dealing with text on computers which we feel we ought to be able to make totally present because it is merely written symbols. What we are ignoring are our own human limitations. As human beings, we relate to the world through the different kinds of Being mentioned above. It is our basic limitation; operating as we do within the auspices of the worldview defined by Kant, and elaborated by all others since him, as the dominant culture. This culture has been very powerful in the subjugation of nature, but we are beginning to reach the limits in the subjugation of nature because the nature we are subjugating is ourselves. The structure of our relation to the world is slowly becoming

visible, and this structure is effecting the very ambitious technical projects that have been undertaken. Parnas' doubts about the feasibility of fielding Star Wars sized systems of embedded software are well founded. We must recognize the importance of the fragments of Being for our software engineering endeavors because they tell us about our own intrinsic limits which will not be solved by any device or technique. This is because these fragments of Being tell us about our relation to technology itself. Technology is a "controlling nature" subdued and made part of our own most being in order to control other parts of nature. This "controlling nature" is forgotten in our project to control everything else. However, we have an essential relation to this "controlling nature" because it is now essentially defining our "human nature". Human nature is not something separate from technology. Human nature has been overwhelmed by technology as "controlling nature". This has been pointed out by philosophers who show the inner relation between Nihilism and Technology[83]. In the master/slave dialectic, human beings have become the slaves to "controlling nature"

83. See Fandozi, P.R. (Washington, D.C.: U.P. of America, 1982)

where they intended to be masters over "controlled nature". This is particularly evident in software engineering in which we have such absolute control over individual small fragments of text that make up programs. But in order to write that text and execute it, human beings must conform to a very alien computational environment where text is not prose, but formal computer languages that have no intrinsic meaning. These languages only have extrinsic, syntactically defined, meaning in terms of performed actions. Where the software system is more than can be represented on a screen at one time, elaborate means of recalling different parts of the software must be developed. The software must be described in different levels of abstract representations in order to be comprehended. Ultimately, unless we are very careful, the text of the software can become a morass which is incomprehensible, uncontrollable, alienating, and meaningless. Human beings confront their limits in software engineering in a way which is very extreme as compared to other disciplines in engineering. These limits show up in the modalities of Being which are displayed and which must be recognized in order for software engineering to be a

successful enterprise.

A way to make this argument more explicit is to consider the formula:

IDEA = FORM + SIGN + TRACE + NULL

An idea is a form given illusory continuity. This illusory continuity is achieved by adding to the form a substrata which functions in a way to confer the illusion of persisting through time. There are three layers that each correspond to a different meta-level of Being. For instance, the form of a letter on the screen of the computer which is presented has the modality of present-at-hand, and its Being[1] is that of Pure Presence. Normally this is all we care to know, and we use these characters in everything we do in software production. Yet the forms of the letters are represented by codes which are bit patterns in memory. These ASCII codes are patterns of signs that form a substrate of the forms of the letters. We have a different way of relating to these signs than we do to the forms of the letters. They have a ready-to-hand status, and they are generally invisible. Only occasionally do we have to deal

with the hexadecimal notation which rule these coded signs. The science of semiotics describes the use of signs in society in general, and here we are talking about a very specific example. Yet the connection between the hidden underlying diacritical sign system and the visible forms is undeniable. In computer systems both are necessary. One is for the human to perceive, and the other is for the machine to operate on. One is visible and part of the presentation of the computer system to the human being, while the other is not apparent but part of the technological substructure upon which the form depends. The system of binary codes is further dependent on the traces in the computer memory that allow its content to be retained over time. This is the bit level composed of off/on electromagnetic pulses. The trace level has yet a different modality in relation to the human being. This is the modality of the in-hand. We can hardly experience this level directly because the individual bit being set or not is normally not something which is important. When we compile or code, we produce the initial pattern of bit level traces which drives the computer. The “object” or “binary” module that is the result of

compilation is called that because it denotes the objectification of our representation as it is transferred into machine readable form. The incomprehensibility of compiled code is the establishment of the fundamental link in software with the strata of traces that exhibits the structure of ~~Being~~[3].

Error correcting codes allow even errors in bit settings to be ignored by the system. Bits are effected by electomagnetic fields, and can be changed by environmental conditions and media or equipment failures. For instance, when a disk is formatted, bad sectors are collected together and marked to be ignored. The myriad of bits within the computer may, at any instant, be in any pattern of off/on configuration. We do not know what that configuration is and do not care. We cannot relate to that much meaningless data directly. We are separated from it by many levels of abstraction. Yet, it is the changes in that overall patterning of traces which allow us to see what is presented to us on the computer screen. What we see on the computer screen must make sense. Anyone who sees a computer program garble the screen image gets an immediate image of the dynamism of this

myriad of traces. The myriad of traces displays the ultimate set of possible system states which is, in most computers, so high that they could not be all realized within the time the universe has existed. We quickly reach the limits of what is computable. Yet, out of all these myriad possible states, we are only interested in a very limited number, and we want to rigidly control the movement of these from one state to another. Yet, we at the same time want to be protected from arbitrary fluctuations in the traces. Thus, traces represent the myriad possibilities of system states. The signs which we use to program the computer on a level of binary codes restrict these possibilities to a very limited number. We use these sign systems through error correcting codes to protect us from random variations in trace phenomena. Thus, sign systems have an intrinsic ability to deal with probabilities. Finally, what the sign system will display is a determinate figure--this character, not any other--which has meaning in this context. A breakdown at the level of sign will cause the wrong form to appear. A breakdown at the level of trace will cause the sign system to compensate for the errors which can be tolerated only up to a certain threshold.

The final level of substrata underlying the form which confers illusory continuity is that named NULL in the proceeding formula. It could also be called “No Trace”. The null state in a computer system is experienced when the system goes down. If you have not backed up, and you suddenly lose an hour’s work, the feeling of utter unexpected loss is horrifying. Suddenly one experiences the ephemeral nature of the electronic media. All those traces suddenly are null, and no trace is left. In other media such as text written on paper this experience is not nearly as immediate. A fire or flood, or some other macro level disaster, must occur to lose the information. Thus, other media that we are used to are far more resilient than the electronic memory within the RAM of the computer. Thus, computers are a fragile technology as compared with pen and paper. One trades fragility for processing power. The null state shows the intrinsic limits of the possible states of the computer. As vast as it is, there are still the limitations imposed when the plug is pulled. The modality of the null state is out-of-hand. It, by definition, is the point at which we lose control completely. What is left when the null state occurs has to be the starting point for booting the system again. From the

point of view of software, the hardware when first turned on, is also in a null state even if it is a random pattern of off/on. This is why initialization is necessary. All on is just as constraining as all off. The initial all on, or random patterning of the memory bits, is the tablet upon which our software writes. The processor (CPU) writes on this tablet of memory locations like a pen on paper, changing each memory location to what its initialized values should be by following the programmed instructions "booted" into a segment of memory. In a computer the interesting aspect is that what controls the processor "pen" is a configuration of the very tablet that is being written on. This paradoxical nature of the software, which makes it possible to write self-modifying programs, has been explored fully in Hofstaders's book Godel Escher and Bach: Eternal Golden Braid[84]. This inherent paradoxicality is a unique feature of software that flows from its ground in Being[3]. Initialization gives form to the computer memory by defining the configuration of traces. Those traces are then manipulated in sets as signs by the boolean logic of the sign codes. The manipulation of the signs is

84. (Harvester Press, 1979)

controlled to present a stream of characters that are meaningful to the viewer. This stream is actually an illusory continuity created by the speed of screen refresh and the clever contrivance of the programmer in manipulating the stream of actions of the computer at an instruction level. What is seen by the user is a carefully contrived illusion.

What is said here about the ASCII code holds for whatever forms we project upon the sign and trace layers of the illusion. At various levels we create abstract forms (objects, databases, files, tools, etc.) whose contents are other forms (buffers, pointers, procedures, variables, data structures, etc.). All that constrains us are the structures inherent in finite mathematics and the structures of algorithms. Through the use of higher and higher level languages, we are able to work the signs and trace systems without dealing directly with them. Formal languages are our intermediary. Thus, formal languages allow us to express forms directly, and the aspects of sign, trace, and null are transformed so that they appear differently at these more abstract levels. For instance, signs appear in algorithms as the indexes of loops controlling iterations or

as the indexes of arrays. Traces appear as transitory state variables, and as momentary data configurations, or as the activated threads of tasks and as the current blocks of allocated memory. All systems have transitory configurations of elements which are not under direct control. These transitory states are controlled by the system completely, but their exact configuration is not relevant to the functioning of the system. In fact, software systems are specifically designed to handle these variations such as network traffic patterns. Embedded systems must even deal with degraded modes of operating, and thus deal explicitly with null states of the traces. These null states could occur by a computer going down within a network so that other machines must take over its functions. PM/FL software monitors the health of the software, watching for these glitches in normal operation, and exception handling in ADA is a specific mechanism for handling these situations. Thus, these same strata that can be identified in the different aspects of the lowly ASCII code can be identified throughout the realm of software. The different modes of relating to technology signified by the fragments of Being may be seen on all levels of software technology. It is

just a matter of reflecting upon the nature of each modality and noticing the unique way it manifests in each instance. Beyond programming constructs, these modalities are exhibited on the methodological level as well and this is the level in which we are the most interested here. Software engineering really exists because for large embedded software systems these modalities become very pronounced and have profound consequences for our ability to build software.

◆ ◆ ◆

Having outlined these different kinds of Being, and shown in simple terms how they may be identified as operating at different levels of software in something as simple as the ASCII code, the next point is to show that of these modalities it is the in-hand which is the most important in relation to software. The Being of software is determined by the in-hand modality. Although all the other modalities are operating within the software system, it is the in-hand modality that is crucial to the nature of software. For software the out-of-hand modality is null. The present-at-hand and ready-to-hand modalities are representations of

the formal and structural systems and the underlying hardware. It is the in-hand modality which is most strongly represented as bestowing the nature of the software system. Thus, all of what Derrida has to say about traces in his book Of Grammatology[85] with respect to writing are directly applicable. Software is written and its nature is determined by the incomprehensible trace that sustains the writing. In the trace is a certain opacity which cannot be captured by either signs or forms. It is in this opacity that the major problems of software find their origin. These problems are just now being felt as we attempt to build large systems and discover that the difficulties uncovered in that project are many times more than might be expected from similar small systems. That opacity is the manifestation of Being[3] which is also called Hyper-Being. This is the meta-level on which the distinction between immanence and transcendence occurs in ontology. The point is that the hiding of the essence of manifestation[86], i.e. pure immanence, is a much stronger hiding than that of Being[2]. This means that the opacity of the trace is a very severe meta-technological

85. (Baltimore: Johns Hopkins U.P., 1974)
86. See Michael Henry; The Essence of Manifestation (The Hague: Nijhoff, 1973)

phenomenon. It can be called meta-technological because it is at the next higher meta-level from Process Being which is where technology has its natural home. The problems of software engineering may be predicted to be very severe because its nature is meta-technical. We are not dealing merely with something that is hidden in order to allow something else to be shown, and which appears when the technical system breaks down. Instead, we are dealing with something that can never be made present. Thus, the work arounds that will make it possible to operate in spite of meta-technical constraints will have to be extreme. This is because, like in quantum mechanics and in relativity theory, there are absolute limits being expressed. These absolute limits are expressed because we have entered the realm of possibilities instead of determinism and probability. We can build any sort of system in software, and there are a myriad of possible ways to do that. The unlimited horizons of possibility contain within them a corollary of the confrontations of absolute limits like those of computation time and NP-completeness. Those absolute limits display an intransigence which is unparalleled by anything offered by lower meta-levels.

Forms contain meaning. Signs contain significance, i.e. differences that make a difference. Traces are mute. That muteness offers almost infinite possibilities, but at the same time there is a incomprehensibleness which cannot be escaped which flows from the fact that the muteness hides the essence of manifestation, i.e. pure immanence.

The image of the essence of software is peculiar. It is the image of a singularity constrained by a formal-structural system which has been fragmented by multiple perspectives. Normally we think of formal-structural systems as combining present-at-hand and ready-to-hand modalities in a specific way to produce a technical system. Computer hardware is a good example of this, as has been explained. The formal-structural systems are not in this case fragmented into aspects that can only be seen from different perspectives. On the level of software though, the formal-structural system is repeated in a strange kind of mirroring which causes us to impose structural and formal models on the new realm of software as our means of dominating it. However, at the software level we discover that the nature of the formal-structural systems that

we create here is radically transformed by the necessity of different perspectives. The formal-structural system is discovered to be fragmented at the meta-technical level. This fragmentation hides a singularity that is embedded in the formal-structural system. This singularity has the same properties as those described in physics in relation to black holes. It is a point where all the rules fail which can never be seen. The singularity in a black hole is surrounded by the event horizon from which no light escapes. Beyond the event horizon all the laws of physics no longer apply. The singularity within a black hole, and that before the Big Bang, share this characteristic of being beyond the laws of physics. Singularities are truly meta-physical entities which we may by definition, never experience directly. They are images of pure immanence that appear in our physical theories because it is difficult to explain certain things, like what existed before the Big Bang, without their presence. It is interesting to note that Sartre's philosophy of "nothingness," which contains a similar structure to the theoretical concept of the singularity, predates its introduction into physics. The theory of "nothingness" cancels with Heidegger's idea of Process Being to

produce the anomaly of Hyper Being. With the introduction of Hyper Being, the possibility of pure immanence having indirectly observable effects on observable phenomena first appears. These effects, which were developed in psychology as the theory of the unconscious, have had a profound effect upon our thinking about ourselves and the world in this century. We speak of singularities in physics without considering how we are really speaking about a meta-physical entity which is a token representing pure immanence. The suggestion that such a conceptual token is also necessary to explain the essence of software should hardly be surprising.

The singularity at the software level does not appear directly. It appears indirectly as the fragmentation of perspectives on the software theory. This fragmentation effects our apprehension of the formal-structural system represented at the software level. This fragmentation does not occur at the hardware level. The formal-structural system at the hardware level is monolithic. We see hardware as given in the world which adheres to formality of boolean logic, and the structuring of space/time as cycles and memory. The

formal-structural system is then applied again to software in the structuring of such artifacts as the ASCII code. At first it appears as if the formal-structural system will be monolithic at this level as well. However, as we begin to enter the realm of software engineering and begin to build large, complex real-time systems, we encounter problems. Slowly it dawns on us that we are dealing with something different from the monolithic formal structural systems we are used to building without difficulty. As chip sizes increase, we apply the same concepts of logic and layout to greater and greater densities without encountering the same problems. Chip layouts are complex and intricate but involve the same principles that were first developed when the first chip was produced. The increasing complexity of hardware does not run into the same problems as the complexity of software. This is why reducing software solutions to hardware implementations can be profitable. Besides speed improvements, there is a conceptual simplification that occurs when we cross the boundary from software back to hardware. The non-monotonicity of the structural system at the software level appears gradually, but fundamentally alters the nature

of the structural-system at the software level.

The non-monotonicity of the formal-structural system appears as the necessity of a set of viewpoints. The software formal-structural system has different aspects which can only be seen by certain viewpoints. These viewpoints require that not all of the software formal-structural system may be seen at once. This allows the differing/deferring at the event horizon that hides advent of pure immanence within the formal-structural system. Software theory is ultimately non-representable because of this fragmentation of partial perspectives. Exactly what these perspectives are, and how they relate to each other, will be treated in the second part of this paper. Briefly though, it can be said that the perspectives are related to the pointing, grasping, space, and time. These are related to the four basic perspectives on software theory which are Agent, Function, Data, and Event. The fundamental supports of the formal-structural system become the basis for the articulation of these perspectives. When these supports that are articulated so that Being^1 and Being^2 become perspectives on the formal-structural system itself, then it becomes possible for $\underline{\text{Being}}^3$ to appear indirectly. This is

really equivalent to turning the formal-structural system inside-out. The formal-structural system is used as a basis for viewing itself. It becomes fragmented in the process, and in that fragmentation that mirrors the fragmentation of Being[87], allows ~~Being~~^{3} to become manifest.

As will be shown in the next section of this paper, once the perspectives are articulated, they become related to each other by method bridges that allow movement between perspectives. These methods, taken together, form the representation of the software formal-structural system. When this representation is analyzed, it has some interesting peculiarities. From this analysis the structural infrastructure (Shell) can be isolated which, with further analysis, reveals the inner coherence (Core) of software theory. This inner coherence of software theory masks the singularity (Kernel) which is the token of the advent of pure immanence. This progressive analysis, which will be carried out in detail in the next section of this paper, shows exactly how pure immanence appears within the software formal-

87. See the forthcoming book by the author titled: The Fragmentation of Being and The Path Beyond the Void: Speculations in an Emergent Onto-mythology. This book explores the foundations of the Western worldview and shows the meta-levels of Being to be an ancient structure in Indo-European culture the meaning of which it attempts to unravel.

structural system from a theoretical perspective. However, it is difficult for us to imagine how the pure immanence of the meta-technical will effect our ability to design software systems. It is being recognized, though, that in distributed systems individual processors are relativistically related[88]. Also, there are many kinds of indeterminacy which can be related to quantum dynamical effects[89]. Relativity is a determinate system for the transformation between the perspectives of different actors. The lack of global system time makes this effect occur. Quantum indeterminateness, on the other hand, occurs because the relation of actors to each other may be constantly changing (this is similar to the problems of self-modifying code). The inherent dynamism of the system causes probabilities to escalate until the effect is indeterminate. These effects are beginning to be seen, and are proper to the expression of the first two ontological meta-levels in the software system. The third meta-level effects are not yet obvious. They will probably appear mostly in the design realm. In design of software there are unlimited possibilities.

88. See Agha, G; Actors (Cambridge MA: MIT Press, 1986)
89. "The Computational Metaphor and Quantum Physics," Manthey, M.J. & Moret, B.M.E. in Communications of ACM, Vol. 26, #2, 1983, pp. 137-145.

These are radically constrained by the requirements and hardware performance. This causes the occurrence of wicked problems for which there is no good solution. Thus, the natural software problems of indeterminacy and relativity of actors, i.e. the space-time constraints on the system and the non-linear dynamics of the system, when realized on a specific hardware platform and to do a specific job, cause major constraints on the problem space which severely limit the possibilities of system design. Here we have a design effect where the nearly unlimited possibilities that exist on the trace level are suddenly constrained very radically, limiting design possibilities. This intrusion of constraints within design is the first major glimpse of the effects of the meta-technical. Pure immanence has effects similar to those of the unconscious discovered by Freud and elaborated by Jung and others. The effects are seen in secondary alterations of phenomena, i.e. displacements. There will be no direct display of effects. All that can be observed are the displacements which when observed closely, can be seen to be strangely systematic. The phenomena of the constraints on design which are normally very profound, when in fact the nature of the design

space is almost unbounded, is just such a strange effect. Design spaces are complex, highly idiosyncratic, multi-dimensionally constrained surfaces. Human beings are expected to deal with these strange problem spaces and get a best fit solution. The nature of the pattern of constraints is highly irregular, and finding a solution at all is sometimes impossible. The play between the seeming freedom of the designer and the irrational constraints imposed that steals that freedom, secretly is exactly an image of the relation between immanence and transcendence. Unlimited possibilities and freedom are signs of transcendence. The constraints imposed that limit this freedom in peculiar, irregular ways are signs of immanence.

We return now to the question of the non-representable nature of software design. Software design is non-representable because of the interference of the essence of manifestation, pure immanence, in its essential nature. It is non-representable because there is some aspect of it that can never be made present, and we are constantly working around this aspect of the design, attempting to get at it all, and being continually thwarted. Like a race

around a ragged rock, as we move this way to see all of the design, there is some aspect of it that moves in anticipation, making some other part of the design disappear. This shows up as the four perspectives on software design that will be fully explored in the rest of the essays in this series. Ultimately, a language for stating software designs is constructed so that this appearance and disappearance of design element templates may be studied in detail. But Peter Nauer's insight into the impossibility of making all of the design perfectly present-at-hand must be extended to say that even a process will not manifest the whole of a design. So the design is not ready-to-hand either. There is always some aspect of the design hidden, and this ultimately shows up as defects which cannot be proven not to exist. Software design is constantly struggling with this non-representableness that ultimately shows up as deeply embedded errors that may or may not be found. Our methodologies seek to reveal as many of these errors as possible as far upstream in the production process as possible. But hidden in the software lurks all these possibilities for error which come from the fact that all the design could not be seen at once nor be revealed by any methodological process.

These traces of the discontinuities between views taken of the design in its development, which during implementation become smeared out through the code due to delocalization,[90] lead to completely unexpected breakdowns in the software system as it actually interacts with the world. Finding and fixing these problems may be very difficult; doing it early in software design may be impossible.

◆ ◆ ◆

From the point of view of this new paradigm for software engineering, it is now possible to answer the questions posed at the beginning of this paper. Some of these answers will be presentiments of positions fully explored in the next section of this paper:

o What is a software engineering method?

> A software engineering method is a means of partially representing an inherently unrepresentable software design theory from a specific design perspective (i.e. AGENT, DATA, FUNCTION, or EVENT). Methods either purely represent a design perspective, or act as a one-

90. "Designing Documentation to compensate for delocalized plans" Soloway, E. et. al. in Communications of the ACM; November 1988; Vol. 31, #11, pp. 1259-1267

way bridge between design viewpoints. These one-way methods are called minimal when they have only the information necessary to perform their bridging function. Minimal methods may be concatenated to create more robust methods that act as two-way methods.

Xaiping Song, who did his Ph.D. dissertation on the comparison of methodologies, has developed a profile which offers the best definition to date of the components of a methodology.[91]

o What is the relation between software engineering methods?

Software engineering methods are related via the intertransformability between design perspectives. Macro methods described by authorities, such as Ward & Mellor, Hatley & Perbhai, Shaler & Mellor, are essentially programs for applying minimal methods. Normally, they give a sequence of applying methods that moves between the design viewpoints in a certain order. The fundamental orientation of this paradigm is that all minimal methods are necessary tools, and that they need to be applied in different orders on different problems. Minimal methods should be seen as a tool kit for solving design problems, and the inter-transformability of design perspectives should be seen as a map whose viewpoints can

91. op. cit.

be traversed in any order. The point that must be remembered, though, is that each different path between viewpoints will give a fundamentally different perspective on the system under design.

o Is there a minimal necessary set of software engineering methods?

Different applications will emphasize more or less the need of the different design perspectives. For instance, data base design will emphasize data-centered methods. For some problems a particular design viewpoint may be hardly needed at all. The design perspectives are meant to act as a reminder as to other possible perspectives on the design problem.

There are twelve bridges between the four software design perspectives. When four minimal methods are added to these twelve which represent each perspective in isolation from the others, this gives us sixteen minimal methods necessary to represent software designs.

o Are there criteria for judging the quality and effectiveness of software engineering methods?

The software engineering methods used to design a program have a direct effect on the quality of that software. Ignored perspectives will tend to cause incoherence in the design. Design should move iteratively between the different design methods traversing different paths between the viewpoints by using the minimal methods that connect the viewpoints. The skill of the designer is seen in the ability to pick the best path between the perspectives in order to unfold the constraints operating on the design. However, the methods themselves are only representational tools. The methods will not lead to the best design. The quality of macro methods depends upon their flexibility in moving between design perspectives and the fullness with which they incorporate all the minimal methods.

o How do software engineering methods relate to the foundations of computer science?

Software methods display an emergent, sui generis, reality which cannot be reduced to computer science theoretical constructs. Software engineering confronts primarily a particular type of Being which computer science does not have to confront directly. Computer science entails the recreation in the software realm of the formal and structural systems related to Being[1] and Being[2]. Computer science also attempts to explore alternative computational

paradigms. Software engineering primarily deals with only one computational paradigm (networked von Neuman), and is oriented toward phenomena arising from Being[3]. Software engineering has a large social and psychological content, and thus is related to the social sciences as well as computer science.

o How should software engineering methods be taught?

Software engineering should be taught as a mixture of social and computer sciences. Software process models should form the blueprint of the curriculum. Software methods should be systematically presented in terms of the concept of switching viewpoints. The skills needed in software engineers are those of theory building which should be given precedence over concrete computational structures such as diverse languages, databases, etc. Methods should be emphasized over programming skills. Programming in the large should be emphasized. Software engineering should include instruction in domain related knowledge.

Software design methods should be taught using the framework of all possible minimal methods as the context for teaching any particular software methodology composed of sequenced and elaborated minimal methods. This will allow the

student to compare and contrast the different methodologies in order to select the best one for the job at hand.

o How do software engineering methods relate to the definition of the discipline of software engineering?

Software engineering is a new type of discipline which will become very important as systems of the Star Wars (SDI) magnitude become commonplace. Methods are crucial to this discipline because they allow abstraction at various levels of very large systems. Unless we formulate workable sets of methods which have automated support, we will not be able to build very large systems. Software methods are the core of the software engineering discipline and display its emergent aspect over against computer science. Knowledge engineering is also a sui generis discipline which has software engineering as its basis.

There will be many different answers to these questions proposed in the coming years. Most will assume that Software is a merely present-at-hand phenomenon which is easily controlled if we will only exert ourselves enough. Some more sophisticated answers will recognize the

ready-to-hand nature of structural systems built in software. To these paradigm builders the dynamic nature of software will be its most interesting aspect. However, there may be some who recognize the inherent blind spot within any structural system built in software. These few will wonder at the strangeness of software and will attempt to discover the noumenal essence of software. This essence will appear to them as a singularity of pure immanence. They will be software ontologists.

◆ ◆ ◆

EPILOGUE:

In this paper the ontological basis of software has been described. That basis appears at the third meta-level of Being. We have not described what appears at the fourth meta-level of Being. For those who are curious, it can be said that this level describes the fundamental basis of Artificial Intelligence (AI) techniques[92]. It is interesting that there appears to be nothing in AI equivalent to a software engineering method. This is because AI techniques represent paradoxes in the software layer. All paradoxes are rigorously excluded

92. See "The Knowledge Level" Kelly, D.A. in Artificial Intelligence; Vol. 18; 1982; pp. 87-127

from software engineering[93]. As E.W. Dijkstra[94] says, software engineering is itself a paradox because it entails the attempt to program when it is impossible. It does this by excluding everything in programming that may cause problems like spaghetti code, self-modifying code, etc. You will notice that the minimal methods which are rigorously defined in the next essay all use formalisms which do not have paradoxical features. There is nothing like self-reflexive reference in any of the minimal methods. All the excluded paradoxicality is pushed into AI's domain where it is used to imitate knowledge and intelligent or living systems. These paradoxes, such as "if statements" used against "if statements" to produce expert systems and all other non-deterministic techniques, are marked by their opacity. They are impossible for humans to understand. Each one is like a cognitive singularity within the field of software methods. Thus, we might say that they are like what Deleuze and Guattari call desiring machines. They are orthogonal protrubrances out of the essence of manifestation, or the unconscious, of

93. AI techniques may be turned back on Software Engineering to aid in its work. See "A Paradigm for Artificial Intelligence in Software Engineering" in Advances in AI in SE, Vol. 1, pp. 1-55, 1990, JAI Press Inc.

94. "On the Cruelty of Really Teaching Computing Science"; Communications of the ACM, Vol. 32, No. 12; Dec. 1989, pp. 1398-1414

software. Their mode of being is the out-of-hand, and their psychological referent is encompassing. In short, they have their foundation in what Merleau-Ponty calls Wild Being. If this theory of AI techniques is correct, then it means that they will never be described fully by any method, formal or structural, and that they will remain incomprehensible to human beings. It is ironic that these are the very structures used by cognitive science to model the human mind because they are intrinsically non-understandable in their operation. AI techniques are right on the edge of the unknown; they are the points where things get completely out of control in the software realm.

There is a continuum which is differentiated by the meta-levels of Being. That continuum is best understood using what I call the Geode theory of meaning. A geode is a rock which when cut open, is discovered to be filled with crystals and many times is empty in the center. The geode looks like any other rounded rock on the outside. But instead of being worn to that shape by the action of water in a stream bed, it is formed in a bubble in mud which, over eons, has water flowing through it, leaving mineral

deposits that form the crystals. All things that have meaning are empty like the geode. Thus, when we look at the different kinds of Being we see that Pure Presence represents the surface of that geode. At this level things have significance by their diacritical relations with other things. When any thing within the web of associations changes, all the associated things change their meanings. Process Being considers the geode from the point of view of the laying down of the mineral deposits that become the crystalline infrastructure. From the point of view of Process Being the geode is a temporal gestalt that only has significance when you look at the whole life-cycle of the formation of the particular rock. It is a system that has a structure. That structure is the crystals inside the rock which are a picture of the deep structural relations that have built up over time. Thus, process, as temporal gestalt, is a system with a deep structure. In the deep structure of things, the associations are constrained by a weighting of some relations as more significant than others. At this level significance is intensified as we learn the underlying relations between things which are most important. Hyper-Being exists in terms of the analogy of the geode as the flaws in the

crystalline structure. Different minerals interact to cause the crystals of different minerals to interfere with each other within the geode. This can cause the crystalline structure to be very irregular and impure. These impurities are the traces of other minerals within the lattice of the predominant mineral. Likewise, as we explore the deep structures of things, we find anomalies which cause us to call into question the regularity of the deep structures we find. When we see the significance of those anomalies, we have an increase of our apprehension of significance through the contrast between the regularity of the deep structures and the inexplicable exceptions that occur because of the differing and deferring of the essence of manifestation. The pattern of the anomalies give us some feeling of the presence of what cannot be known which interferes with the deep structures of what can be known. Wild Being sees the play of light off the crystals within the geode itself. The crystals are semi-regular multi-colored faceted forms. In the Geode there is never any light. It is always dark, and so those faceted forms never catch the light unless we cut them open. However, there is always that possibility of the light bouncing off

their jagged forms from many different angles. This is possible because the geode is normally hollow inside. Cancellation of the antimonies of Hyper-Being is the equivalent to the manifestation of immanence. All the flaws in the crystals, if they are brought together, form a pattern. This would be the attempt to make what can never be seen appear. Since this is impossible, cancellation occurs to prevent the betrayal of immanence. What is left after this occurrence is merely the reflections of the surfaces of the crystals. Ideation vanishes into the flesh of perception turned in on itself without the domination of the ideas. Ultimately, Wild Being is merely the interface with the Void.

From the point of view of the technological system, the surface is the myriad relations between the different components of the technology. These relations are dynamic and produce a temporal gestalt in which the technical system evolves. In the evolution of the technical system, there are deep structural relations that hold throughout the evolution. The meta-technical system of software is what integrates and holds together the components of the formal structural system. The software is

ultimately only the traces of ones and zeros within the hardware system. These traces, by their differences, show the fragmentation which reveals the absence that cannot be made present within the traces. At the level of abstraction of software design, this is the different design perspectives; at the code level, this is delocalization. By using structural formal systems, we attempt to isolate these anomalies and separate them from the code, calling them errors. When the possibilities for these errors are themselves studied and exploited, we call these artificial intelligence techniques. They are paradoxes in the software code or methods layers. They have interesting properties all their own. When we delve into them, they are opaque like human error proneness itself. They are the interface with the unknown within the technological system. That system is empty like everything else. It is because of this emptiness that things like the technological system can have meaning. As the Tao Te Ching[95] says, "The wheel, bowl, door, and window are all useful because they are hollow." The emptiness of things is what allows them to have unending unfolding realms of meaning. Meaning is the infinite limit of

95. This is a paraphrase. Henricks, R.G. Laotzu Te-Tao Ching (Bantam 1989); p. 63; Chapter 11.

significance that lies at the heart of all things including the technological system. That hollowness is like a cornucopia from which meaning pours out into the world, and appears as pure meaning as well as the different kinds of significance related to the four meta-levels of Being that ground the technological system. Each kind of significance is related to a particular kind of truth that is associated with each meta-level. The truth of Pure Presence is verification. The truth of Process Being is manifestation itself. There is a truth in the mere presencing of something whether it can be verified or not. The truth of Hyper Being is the same as the truth of the individual unconscious. It is a truth of the inexplicable coherence of distortions within consciousness. The truth of Wild Being is the same as the truth of the collective unconscious. It is a truth which is related to the species and universal human experience. When those archetypal forces seize us they cannot be denied. Ultimately there is the truth of the emptiness of all things. The Buddhist concept of Emptiness, which is itself empty, was constructed as an antidote for Being in all its kinds. But it is because everything is empty that they can interpenetrate like a hologram, where each part

carries a partial patterning of the whole. This is possible because each thing is only the sum of its difference from every other thing, and those differences are discontinuities between kinds that are ultimately just voids. To the Buddhist the whole world is an illusion because this network of differences has no foundation and ultimately collapses into itself as the emptiness within the emptiness manifests. Notice that each type of truth associated with a meta-level of Being takes us deeper into the human being. The levels of significance within the technological system externalize these layers which are the different modes with which the human being projects his world.

If we take as an example[96] the computer program that plays chess better than a chess master, we can see all the strata of the different meta-levels of Being within this cultural object. On the surface there would be no executing program without the hardware. The hardware is the formal structural platform of the program. The program itself is at a meta-technical level, pulling together all the hardware resources into a functioning whole. The program will normally use search

96. I am indebted to John Irwin for the suggestion of this example.

algorithms to look at the space of all possible future games with the initial conditions of the pieces where they are now. The use of the search algorithms on possible future moves is considered an artificial intelligence technique, especially since it is necessary to constrain the search space by heuristics representing human knowledge. The software program is poised at the meta-level of Hyper Being, while the AI techniques used to approximate human cognition are at the meta-level of Wild Being. But ultimately the Chess program is empty ,and because it is empty, it may be seen by human beings to have meaning as they interact with it. The meaning of a game of chess flows out through the illusion that it is playing as a human would. All the layers of significance and their corresponding truths are there in that game. But it only has meaning for the human being who is playing the game with the computer or observing two computers play. The technological system lacks intentionality and therefore cannot tap into the emptiness from which meaning arrives. The imitation of human responses is not the right criteria for judging intelligent or life-like activity.

The Turing test is not the proper measure of the

technological system with all its meta-levels of Being. Actually, the technological system is utterly alien and exhibits only an alien intelligence as we combine different AI techniques which are all opaque in themselves as paradoxes. The fact that these alien intelligences can imitate human response is not the important point. The important point is that they can do intelligent things which humans cannot do. Thus, when the human confronts the technological system, it is confronting the alien-ness of technology itself, which is radically alien because it unfolds from out of the essence of humanity itself as a cultural product that is out of control. Utter alien-ness comes from ourselves, not from outer space. It will manifest in the social inner space of Virtual Reality. Cyberspace is the embodiment of the emptiness of the technological system with all its layers of difference and meta-levels of Being. Only humans have the ability to project a world. Projecting a world means tapping into the horizons of meaning beyond the technological system. Those horizons of meaning give a context to the significances of the technological system but that system, itself is allopoietic (other made), not self-organizing. If the humans went away and left

these artifacts, they would be without meaning. WIthout the constant input of attention by the humans who see relevance, the meaning dries up, and the significances within the layers of the technological system lose whatever meaning has been attached to them. Only humans can discover and tap the sources of meaning which lies in the emptiness of all things. Thus, no technological artifact will be able to capture semantics except in a peripheral sense as embedded significances. Semantics are embedded in the human projection of the world. The projection of the world on the emptiness of things is the function of Ideation and expresses all the meta-levels of Being. The world is full of sources of meaning which the human being discovers. The technological system does not project a world. Without the human being, the technological system becomes discarded heaps of junk. Unused software is even more pathetic as it merely appears as random magnetic impulses on some discarded medium. All the embedded significances become static without the life given to them by human attention and valuing.

The technological system is a mirror for Western man. It is not a man, but his opposite.

As his opposite, it embodies pure alterity which appears in the technological system as alien intelligences of artificial techniques. That mirror is ultimately groundless. As we stare into the groundlessness, we realize that it is groundless because Being, in all its kinds, is cancelling with Emptiness, which is empty. The confrontation between man and technology is the realization that this cancellation of Being and Emptiness is the essence of Western man himself. Suddenly we realize that it is only what is beyond this ultimate cancellation beyond the cancellation of Antimonic concepts that can be the source of meaning and our world. So all derivative types of meaning, i.e. significances, within the technological system have no foundations. Semantics will always be the fundamental problem of the technological system because it points back to the human who is projecting the world and finding the sources of meaning within that world. Today we find the technological system to be our greatest source of meaning. But that is ultimately because it is, like everything else within the world, empty. That is to say, it manifests above all its own groundlessness that comes from the fragmentation of Being and the cancellation of the different kinds of being with

emptiness. The fragmentation of Being is itself the manifestation of the void within Being itself.

Just as the fragmentation of perspectives on software designs reveals the unconscious, unexpectedly within the technological system, where we thought we were safe from the vagaries of the human self discovered by psychology, so we discover the archetypes of the collective unconscious as artificial intelligence techniques. The artifacts of the allopoiesis of the socio-technological system have not undergone the same critique as literature where the structures of the individual and collective unconscious reveal human foibles. But these adumbrations of the human spirit surely exist in this realm as well. All human cultural products can be expected to demonstrate the same universal structures that sociology and psychology find outside the realm of the technological system. But the technological system takes certain aspects of our world to extremes, so that we see the groundlessness of Being appear in surprising forms as technology strives against its limits. One of those limits is the expression of meaning. Semantics are successively

embedded into the products of the socio-technological system as layers of significance, as it strives against the limits of the fact that meaning only pours out of emptiness. The emptiness of the emptiness is the very embodiment of that groundlessness of all things within the world, especially formal structural systems and their meta-technical web that reveals points of paradoxicality. The articulation of the points of paradoxicality and their combination into alien intelligences reaches toward the expression of meaning. But since the technological system isolated from its human sustenance does not have a world nor project a world, but is, in fact, only the remnants of a particular way Westerners project their world, there is no chance for meaning to arise from the technological system itself apart from the sociological and psychological realities of the humans projecting their world through the lens of that empty technological system.

■ ■ ■

KEYWORDS:

Software Engineering, Software Design, Software Methodologies, Ontology, Theory, Being, Philosophy of Science, Technology, Formal Systems, Structural Systems, Systems Meta-methodology, Grammatology.

ABSTRACT OF SERIES:

Software Engineering exemplifies many of the major issues in Philosophy of Science. This is because of its close relation between a non-representable software theory that is the product of design and the testing of the software. Software Engineering methods are being developed to make various representations of the software design theory. These methods need to be related to each other by a general paradigm. This paper proposes a deep paradigm motivated by recent developments in Western ontology which explains the generally recognized difficulties developing software. The paradigm situates the underlying field within which methods operate and explains the nature of that field which necessitates the multiple perspectives on the software design theory. Part One deals with new perspectives on the ontological

foundations of software engineering based on recent developments in Western philosophy. Part Two deals with the meta-methodology of software engineering by laying out the structural relations between various software architectural methods. Part Three presents an Integral Software Engineering Methodology based on these foundations.

OVERVIEW OF PART ONE:

In this first part a deep fundamental paradigm for the software engineering discipline will be proposed. The development of this paradigm will draw upon insights from modern ontology of various recent Western philosophers which will be treated systematically. A heuristic conceptual model of kinds of Being will be developed to facilitate the understanding of ontology and its relation to software engineering. Connections between the heuristic model and different forms of modern mathematics will be made to make the intellectual transition from a philosophical to an engineering universe of discourse possible. The relation of philosophical concepts which underlie philosophy of science to the software

engineering discipline will be the main focus of this essay.

[sefp1_1draf65/930517kdp (X89-784/301)]

Author's address: P.O. Box 4402, Garden Grove CA, 92642
[palmer@exo.com]

Apeiron Press

PO Box 4402
Garden Grove,
California 92842-4402

714-638-7376
714-638-1210
palmer@think.net
palmer@netcom.com
palmer@exo.com
Dataline 714-638-0876

Draft #1 950710 Editorial Copy.
Not for distribution.

This book was set using Framemaker document publishing software by the author.

Electronic Version in Adobe Acrobat PDF available at http://server.snni.com:80/~palmer/homepage.html

Library of Congress
Cataloging in Publication Data

Palmer, Kent Duane

WILD SOFTWARE META-SYSTEMS

Bibliography
Includes Index

1. Philosophy-- Ontology
2. Software Engineering
3. Software Design Methods

I. Title

[XXX000.X00 199x]
9x-xxxxx
ISBN 0-xxx-xxxxx-x

Keywords:

Software, Design Methods, Ontology, Integral Software Engineering Methodology, Systems Theory