SOFTWARE ENGINEERING FOUNDATIONS

A Paradigm for Understanding Software Design Methods

Software Engineering Design Methodologies and General Systems Theory

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

1. SOFTWARE ENGINEERING: A YOUNG DISCIPLINE.

Software Engineering[1] is slowly emerging as a discipline in its own right from the craft of computer programming[2]. It is a discipline grown up in industry rather than invented in academia. It addresses the problems that occur when a group of people attempt to construct a large software system. The key word is ***large*** because when the software product gets large enough that it can no longer be completely understood by one person, or a few, certain problems emerge that are not seen in smaller systems. These are problems like the need to communicate the design to others; the need for

1. S. Pflaeger *Software Engineering: The Production of Quality Software*. Macmillan 1991.
R. Pressman *Software Engineering: A Practitioner's Approach*. McGraw-Hill 1987.
I. Sommerville *Software Engineering*. Addison-Wesley.
J.A. McDermid (ed.) *Software Engineer's Reference Book*. Butterworth-Heinemann 1991.
2. "The evolution of the Software Engineering Discipline" Mary Shaw ?????

concurrent cooperative software design by a large number of people working on different aspects of the same problem; the need to control the artifacts produced by the essential transformations in the software development process; and the need to reuse as much of previous software systems as possible. These are the kinds of issues that emerge when software development attacks large problems and builds hundreds of thousands or millions of lines of software code.

Software Engineering is still in its infancy as a discipline, yet already it is experiencing paradigm shifts[3]. A paradigm shift is a radical change in the way of looking at known facts of a discipline. In Software Engineering the current paradigm shift is away from functional design[4] toward what is called object-oriented design[5]. This is a shift away from looking at pieces of software as functions that transform a set of inputs into outputs, toward looking at the persistent data in a system, and seeing the functions that operate on these pieces of persistent data as being grouped around that private data. This paradigm shift is causing old

3. The Structure of Scientific Revolutions Thomas Kuhn ????
4. Structured Design Yourdon / Constantine ??????
5. Some ref on OOD ???

problems to be seen in a new light. And it is causing a rethinking of the way we produce software systems.

Although Software Engineering is developing and changing rapidly within itself, the relationship it has to other disciplines is still unclear because it is not well integrated into the academic establishment as yet. Thus, the new inventions of Software Engineering within its own field may seem baffling or be seen as an exercise in re-invention by other more established disciplines. This problem is exacerbated by the fact that Software Engineering is perceived as a renegade newcomer. At the same time these more established disciplines are themselves developing software systems to support their own applications. So, although they may not have a good connection to what is happening in Software Engineering as a discipline, they have a connection to software products that they use in their own ongoing work. There is a vital need for other disciplines to learn what is happening in Software Engineering, and for software engineers to understand what other disciplines have to offer toward the development of the Software Engineering

discipline.

One of the most vital of these connections is between Software Engineering and General System Theory. Software engineers profess to be building software *'**systems**.'* But, beyond the loose generic use of this term, there is not a lot of material within the corpus of Software Engineering literature concerning the generic nature of systems of which software systems is an instance. On the other hand, General Systems Theory, which gives itself wholly to attempting to understand generic systems like many other disciplines, uses software applications in its own work, and perhaps is unaware of the developments within Software Engineering as a discipline; developments which are interesting because of the additional light they shed on systems in general. Because software systems emulate many different types of systems and allow them to be tested and interrogated as simulations, these software simulations are a valuable tool that function as the instruments by which General Systems theorists do much of their investigation. Yet these tools may not be designed and implemented using the methods which are currently state-of-the-art within Software

Engineering. Those who build software models are generally operating within the programming craft stage of software discipline.

Thus, the question arises: What do General Systems Theory and Software Engineering have to learn form one another? The answer that will be explored in this paper is that these are both meta-disciplines, and that they actually have a symbiotic relation with each other which sets them apart from all the other disciplines. General Systems Theory looks at all kinds of systems that appear in the world, and attempts to abstract their generic features. This allows similar systems in widely different fields to be recognized. Software Engineering builds simulations of systems in widely varying fields by applying the same techniques for understanding and automating those systems. Thus, with General Systems Theory it is the representation of the systems that are abstracted, while in Software Engineering it is the means of embodying the abstracted or concrete system that is generic. The telling thing is that General Systems Theory itself uses software representations of abstract systems such as the General Systems Problem Solver (GSPS) provides. Software Engineering, on

the other hand, designs and builds software ***'systems'***. So, not only are both meta-disciplines which abstract from and are used by all other disciplines, but each of these meta-disciplines need and use each other. For General Systems Theory *software* is the means of simulation and instrumentation of the systems being studied. For Software Engineering, a specific kind of ***'system'*** is being built.

Software Engineering and General Systems Theory are symbiotic. The question is, what does this really mean for the relation between the two meta-disciplines? This will be the subject of the rest of this paper.

2. THE GENERAL SYSTEMS THEORY DISCIPLINE IN RELATION TO SOFTWARE ENGINEERING.

It is somewhat understandable why software engineers might ignore their sister meta-discipline. Generally Systems Theory itself has had a hard time getting recognition of its claim to a central role in science. It has been difficult to show that Systems Theory has anything to contribute other than generalizations which, because they are too abstract, do not really tell you anything. (Software Engineering, on the other hand, suffers from the criticism that everything it has to offer is too concrete.) Unfortunately, the connections between disparate disciplines that treat specific kinds of systems through General Systems Theory have not yet happened. Recognizing that the systems treated by different disciplines are really projections of the same general system has not yielded enough fruit to make this an important aspect of science.

General Systems Theory stands in the same relation to the sciences as Mathematical Category Theory does to the separate kinds of mathematics called categories[6]. We cannot yet

6. Arrows, Structures, and Functors, M.A. Arbib & E.G. Manes NY: Academic Press 1975

see the functors (homeomorphic relations) between different systems that General Systems Theory should eventually provide. However, with the advent of the structural General Systems Theory of George Klir, it is possible to see the outlines of this science emerging. It is only right that this system should be expressed in terms of Mathematical Category notation as it is in another article in this issue. We might expect software to have some affinity with category theory as well. Since software is the universal implementation medium of all mathematical categories, and because software has only mathematical limitations to its structure and shape, we can see that Category Theory is well suited to the understanding of the relationship between software designs and methods[7].

7. Categories, Types, Structures Andrea Asperti Giuseppe Longo QA76.7A76 1991

Figure 1:

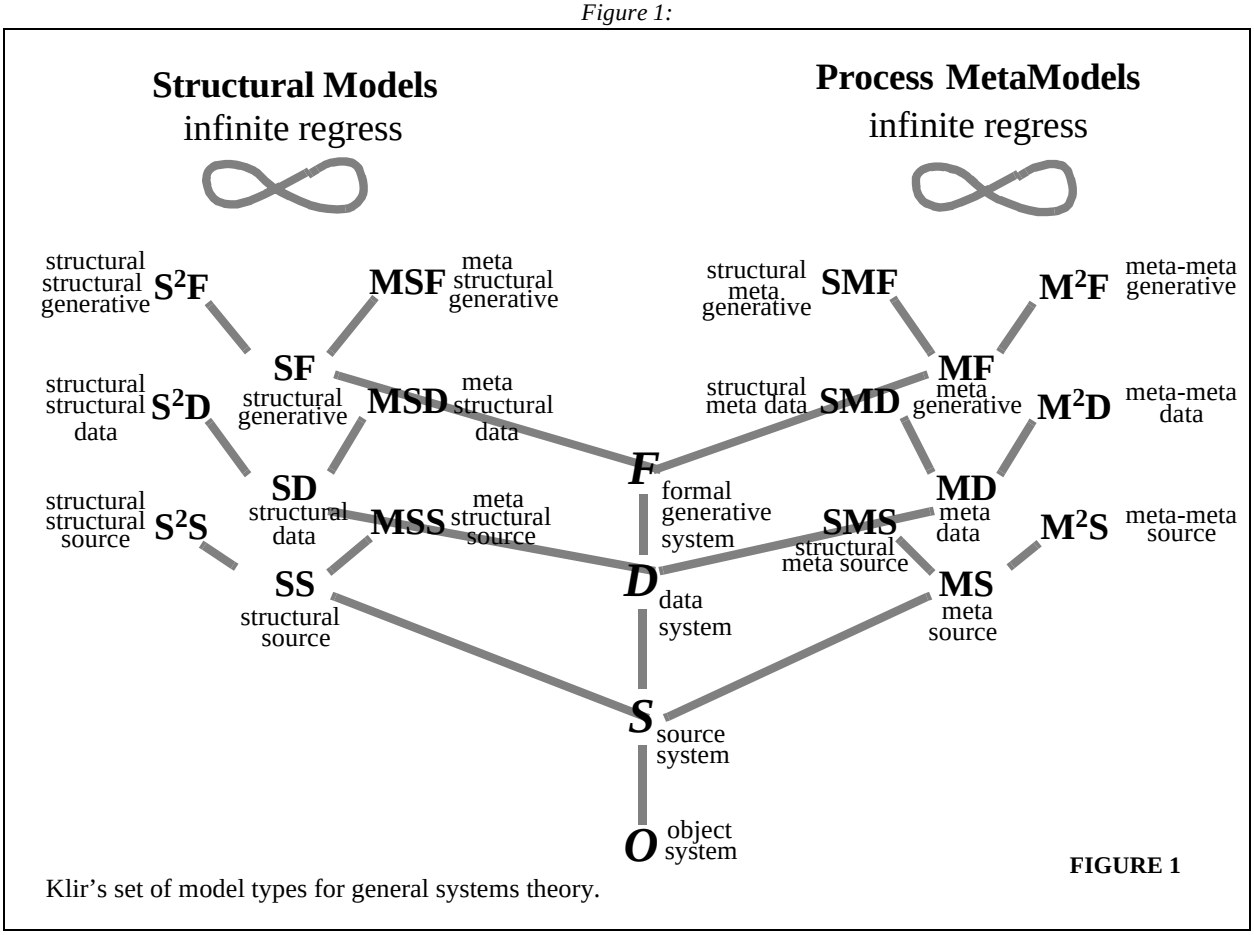


FIGURE 1

Klir's set of model types for general systems theory.

It is important to understand why GSPS is an advance over previous systems theories. I would characterize GSPS as a structural-formal systems representation, as opposed to merely a formal theory. Early systems theory really did little more than posit that "everything is a system." What GSPS adds is the structural dimension that differentiates between different systems architectures. This is accomplished by invoking a hierarchy of epistemological

levels[8]. The recognition of these levels is very important, as it gives clear definition to the subject matter of General Systems Theory and the differentiation of the field of possible systems architectures. Knowing what the structure of this field is, must be an essential step in the recognition of different systems with the same architecture. Since software allows any mathematically possible structure, software designs may be seen to cover the same field. It is the absolute malleability of software (within the constraints of hardware) that makes it fit to model all the diverse actual systems that are specific points in the field of all possible system designs mapped by General Systems Theory.

Given the epistemological levels posited by GSPS, the obvious question occurs as to what the relationship of software design methods (which allow different designs to be represented) is to the overall epistemological framework. But, first we must understand something about the underpinnings of the GSPS framework in its role as the articulator of design space for systems. There is a very important point about this framework which is

8. See Klir ASPS Figure 1.3 Hierarchy of epistemological levels of systems: a simplified overview. Page 16.Represented in Figure 1.

perhaps not widely appreciated: the framework goes to infinity in two directions. The semi-lattice diagram[9] shows meta-systems diverging from meta-structures as two completely different infinite regresses of meta-levels of description in orthogonal dimensions. The first dimension is structural with patterns within patterns in part-whole relationships. The second dimension is meta-systemic where discontinuous changes are handled by positing modal changes of the system. These two dimensions are very different. Meta-systemic patterns are diachronic and have to do with the changing of patterning regimes within time. Pattern meta-levels are synchronic embeddings of higher logical types of order. They have to do not with time, but with space where data has a different fractal patterning at each partial dimensional level. We might say that the second dimension that deals with meta-models is the process dimension because it allows embedded patterns or structures to vary in time The meta-models determine the discontinuous changes between synchronic structural transformations which appear as a changing process articulated diacronically. Together,

9. See Klir ASPS Figure 9.1; page 420

these two infinite regresses show that ultimately any system is unrepresentable in its entirety. This is because after just a few meta-levels of spatial or temporal embedding we get lost in the ultra complexity of all but the simplest dynamical systems. The most we can hope to do for all, but inherently deterministic systems, is give an approximation that goes up just a few meta-levels in relation to structural pattern and process meta-systems.

This framework in General Systems Theory both makes its limits clear, and also allows it to represent the architecture of diverse systems by allowing higher and higher logical types of description along both of these dimensions. But, we are led deeper to ask why this limitation exists and what it represents. This limitation has to do essentially with the way in which systems themselves are manifested. It is not merely an artifact of our descriptive conventions. Because General Systems Theory purports to be a science, it operates with the same self-imposed metaphysical blinders as all sciences. However, this does not mean that it does not have metaphysical presuppositions built into it. GSPS has its own metaphysical presuppositions, and one of those is that the

ultimate system can never be known. That limit system that is unknowable is an image of the Kantian *noumena*. This unknowablity is built into the theory as the twin infinite regresses. By building in this presupposition, GSPS is able to fine tune its description of systems by providing a structural model of their possible systems architectures. It is able to provide a representation that can track the changes in the system over time and to any level of sub-patterning. On the other hand, it can handle multiple interlaced part-whole patterns. However, the non-manifestation of the system as noumena is a steep price to pay for this structure and flexibility. It is this metaphysical dimension that needs to be appreciated because it is exactly here that the innate affinity with software can be shown to exist most strongly.

Dr. Klir has in several papers pointed out the new information aspect of science that he proposes to explore[10]. This new aspect has its own metaphysical attributes which it shares with software. We can best characterize this new aspect by relating it to the concept of "Being" which is generally thought to be the

10. Klir on information sciences

most general, and thus most empty, concept. Heidegger gives the concept of Being content by associating it with 'presencing'[11] or 'manifestation.'[12] Being is the process of manifestation of "whatever manifests." From the time of Aristotle, through Kant, and up to that of Husserl, this manifestation has been thought of as pure presencing of clear and distinct objects that can be seen from an ideal point of view that sees all sides of the object at once. In this ideal presencing the noumena, the inner coherence of the objects, remains hidden because it is on a transcendental plane. However, the object itself seems to be fully available. This is the type of manifestation that science assumes when it looks at its objects. It assumes that the object is fully available for inspection in every detail. This is the type of Being Klir assumes when he looks at his Object systems and turns them into Source systems. However, by differentiating the Object from the Source systems, he is making a crucial admission that the Source system is only seen to the extent it is turned into an object viewed by a passive observing subject. The process by which such a system is turned into an object is

11. R. Harper, On Presence: Variations and Reflections. Trinity Press International, Philadelphia, 1991.
12. Heidegger on Manifestation???

unspecified except that it entails some type of special focusing, such as that Descartes introduced into science, that makes clear and distinct (delimited) objects out of the blurry masses presented to the natural senses[13].

Seeing a system rather than an object is not differentiated by Klir. To him a system is an object which has been instrumented, or made available, for observation. However, I believe that a system is on a completely different ontological meta-level from the object, and I think this is implied by Klir's usage. I think this difference between object and system is the same difference that Heidegger[14] makes between Pure Presence type of Being and what may be called Process Being, which is "Being mixed with time." In Process Being the process of manifestation of the object is taken into account. The major effect in this manifestation is called showing and hiding[15]. In manifestation not all aspects of the system are seen at once. The way in which the object comes into and goes out of view is also taken into account. This has very important ramifications for systems science which is not

13. E. Zerubavel, The Fine Line: Making Distinctions in Everyday Life. The Free Press, New York, 1991.
14. Being & Time (NY: Harper & Row 1962)
15. A. Blum, Theorizing. Heinemann, London, 1974.

widely appreciated. Because systems science sees itself as science at a meta-level above all other disciplines, it is really ignoring the fact that it is a 'meta-physical' discipline. The word 'system' is the catch-all that attempts to bridge this gap. When you ask what a system is, you get back the answer that "everything is a system" that you consider as such. So 'system' is really a way of seeing the world: it is a worldview, a metaphysical approach to reality.

Fortunately, Nicholas Rescher[16] gives a clearer definition of the characteristics of a system:

> Lambert contrasted a system with its contraries, all "that one might call a chaos, a mere mixture, an aggregate, an agglomeration, a confusion, an uprooting, etc." ... And in synthesizing the discussions of the early theoreticians of the system-concept, one sees the following features emerge as the definitive characteristics of systematicity:
>
> 1. wholeness: unity and integrity as a genuine whole that embraces and integrates its constituent parts
>
> 2. completeness: comprehensiveness: avoidance of gaps or missing components, inclusiveness with nothing needful left out
>
> 3. self-sufficiency: independence, self-containment, autonomy

16. N. Rescher, Cognitive Systematization. Rowman & Littlefield, Totowa New Jersey, 1979, pages 10-11.

4. cohesiveness: connectedness, interrelationship, interlinkage, coherence (in one of its senses), a conjoining of the component parts, rules, laws, linking principles; if some components are changed or modified, then others will react to this alteration

5. consonance: consistency and compatibility, coherence (in another of its senses), absence of internal discord or dissonance; harmonious mutual collaboration or coordination of components "having all the pieces fall into place"

6. architectonic: a well-integrated structure of arrangement of duly ordered component parts; generally in an hierarchic ordering of sub- and super-ordination

7. functional unity: purposive interrelationship; a unifying rationale or telos that finds its expression in some synthesizing principle of functional purport

8. functional regularity: rulishness and lawfulness, orderliness of operation, uniformity, normality (conformity to "the usual course of things")

9. functional simplicity: elegance, harmony, and balance, structural economy, tidiness in the collaboration or coordination of components

10. mutual supportiveness: the components of a system are so combined under the aegis of a common purpose or principle as to conspire together in mutual collaboration of its realization; interrelatedness

11. functional efficacy: efficiency, effectiveness, adequacy to the common task.

> These are the definite parameters of systematization. A system, properly speaking, must exhibit all of these characteristics, but it need not do to the same extent -- let alone perfectly. These various facets of systematicity reflect matters of degree, and systems can certainly vary in their embodiment.

These characteristics of system are ignored by the objectivists which want a simple operational concept of system which is fully available, but essentially empty of content. These characteristics are implicit in the concept of system as it is actually used. This is true even of the objectivists, which is shown by the fact that the concept of an "object" remains different from that of "system." The added implicit meanings of system are those pointed out above by Rescher. All of these characteristics of the system relate to the difference between the object and what might be called the 'temporal gestalt.' The object is a synchronic slice out of the diachronic temporal gestalt at the idealized "now" point. The temporal gestalt itself includes the whole evolution of the object. It thus inhabits at least the so-called "specious present[17]," or the extended now, and moves out toward the entire temporal interval in which the object manifests.

17. William James ?????

The temporal gestalt is modeled by the process dimension of Klir's epistemological framework. The difference between Pure Presence and Process Being is the difference between the temporal slice and the whole process of unfolding. It is in this process of unfolding that the characteristics of system enumerated above unfold. As a temporal slice, the object seems to have almost arbitrary features because any slice of the whole process of unfolding might have been taken. But if you consider the whole process of temporal unfolding, you get a complete dynamic picture or gestalt. One gets a sense of the wholeness of the system and can judge the completeness of any slice. The temporal gestalt exhibits apparent self-sufficiency because it is an entirely self-contained process of unfolding, exhibiting its own boundaries rather than being delimited arbitrarily. As a gestalt pattern, it exhibits internal coherence or cohesiveness, and external coherence or consonance. It has its own architectonic with functional unity, regularity, simplicity, and efficacy. The system's parts exhibit mutual supportiveness. To the extent we want to know what is really "out there" in the world, we must take into account the temporal gestalts unfolding as

natural complexes[18]. To the extent we want to project our own order on the world and exclude whatever does not fit with our ordering, we will want to stick to the objectivist metaphysic that has the aim of controlling, rather than engaging in dialogue with phenomena. The object contrlos the phenomena by ordering it as a passive strawman determined by our subjective whim. The system as temporal gestaltallows the observed phenomena to have its own voice and speak for itself thorugh its own self-articulation over time.

If we release our metaphysical imaginations for a moment, we might see that systems are objects caught in webs of showing and hiding. Thus, the difference between an object and a system is whether it is looked on as something that is completely available or something that is only partially available and at least partially determines its own availability. The partial availability prevents us from capturing all of the Object systems in our source system. The partial availability is what separates the object from the source. They are really two ways of looking at the same thing. Moving from source system to object system, one has moved from

18. Natural Complexes

blurs of perception to idealized objects that are purely available as sources for observation and experimentation by a subject. We have put on the metaphysical blinders which makes us think our scientific objects are “objectively” accessible to an intersubjective community.

For Heidegger, the realm of showing and hiding is not just something to be escaped from by science. Instead, this realm is the infrastructure that science itself simultaneously depends on and ignores. It is the realm of technological means. A simple demonstration can give you the feeling for this. In writing, you are unaware of your pencil. Your focus is on what you want to say, not on the movement of your hand in recording what you are saying. This is the difference between present-at-hand (pure presence of the object) and the ready-to-hand way of relating to the pencil which is moving in time and supporting the objective focus. So it is with all technology. Technology supports the endeavors of science. And the major instrument through which this is done today is the computer. In fact, it is interesting that the computer has in its make up all the elements of the structural system which join present-at-

hand and ready-to-hand modes of relating to things in a single unified mechanism. This is what makes the computer a general purpose machine that is applicable to almost every task[19].

One might look at the difference between "object" and "system" as being like the difference between figure and ground in gestalt psychology. The object is the focus of attention. The system is the interaction of all the possible focuses in the field of the system. However, you can only see one or a few objects at one time, so in looking over the entire field, the center of focus must shift from object to object. This is the manifestation of showing and hiding within the field, that is, in fact, what we call the system. The system is a group of objects in the same field of showing and hiding relations.

Another way to look at this is by considering the difference between Formal systems and Structural systems. In a Formal system such as symbolic logic, geometry, or any mathematical category, there is a set of axioms, and it is through the relationship between the axioms that the full Formal system is built. However,

19. K. Palmer "SOFTWARE ENGINEERING FOUNDATIONS: Software Ontology" Unpublished Manuscript available for review from the author on request.

this system is made of purely logical relations, and time does not effect the system except to the extent it takes time to go through the proofs step by step. On the other hand, a Structural system is a powerful means for understanding discontinuous transformations. The Structural system attempts to apply formal rules to the content of the Formal system in order to track the changes that occur across discontinuities. These discontinuities can be structural or changes in pattern at different levels of magnification in one synchronic slice. The Structural system is basically descriptive. It gives up the rigor of deduction for the ability to handle change. George Klir's GSPS is essentially a Formal-Structural system. Its epistemological levels allow different Formal systems to be connected together to form a Structural system that can handle change. It is, in fact, the best example of a Structural system that I have found. Structuralism has been most effective in understanding the universe in the hands of physicists and chemists. It has even made strides in biology with the discovery of the DNA genetic code. And some use of it has been made in the social sciences with Structuralists like Noam Chomsky[20], Claude

20. N. Chomsky on Transformational Grammar

Levi-Strauss[21], Jean Piaget[22] and others, using it to describe aspects of human nature and culture.

However, the difference between object and system as two different ways of looking at things does not exhaust the implicit metaphysics in GSPS. By introducing the concept of the two infinite regresses, GSPS implicitly hints at the presence of another metaphysical assumption which has also appeared in Modern Ontology. The question occurs to us, where do these two infinite regresses lead? That place is where the system exists as unreachable noumena. The system as noumena is very different from the objective noumena of the idealists. In modern ontology, various philosophers have sought to describe what the next higher meta-level of Being is above Process Being. Heidegger calls that next meta-level ~~Being~~ (crossed out). Derrida has called it DifferAnce[23] which he describes as the differing/deferring of writing. In this conception we differentiate between speech and writing. In speech, a showing and hiding process occurs, and it carries us along,

21. C. Levi-Strauss, The Savage Mind. University of Chicago Press, Chicago, 1966.
22. J. Piaget, Structuralism. ????
23. J. Derrida, Of Grammatology. The John Hopkins University Press, Baltimore, 1974.

presenting us a series of ideas that appear fully available in language. The flow of language is the emanation of the temporal gestalt in action. It is the basic pattern for the ready-to-hand's technological support of ideation. However, writing is different. Writing makes it possible to create partial texts which are scrambled into a different order than that in which they were written. The text is a field of pure difference in which things that were together may appear apart, or in different orders, than those in which they were composed. Another ontologist, Michael Henry, speaks of what is called the Essence Of Manifestation[24] which is the pure immanence underlying the manifestation (aka transcendence) that we see as showing and hiding. What is purely immanent never appears, but shapes what does appear. It becomes particularly conspicuous in texts which are susceptible to unconscious distortion. You know it is there by the way what appears is distorted or malformed. Taking these two ideas (differAnce and the unconscious essence of presencing) together, we can see their relevance to GSPS. GSPS posits two infinite regresses which hide the

24. M. Henry, The Essence Of Manifestation. ????

noumena of the system. We would like to call the noumena of the system a meta-system or proto-gestalt. In other words the temporal gestalt arises from an always already lost origin from which many systems (gestalts) emanate and toward which they all point. This origin is th next higher meta-level beyone the system. The noumena of the system (essence of presencing as such) never appears because it has been first covered over multiple times. Firstly, it is covered over by the extraction of the object from the source system into the realm of pure availability. Secondly, it is covered over by the showing and hiding relations between multiple objects which makes the system itself something that is never seen because it is always in the background. All we really see is the multiple showing and hiding relations. Thirdly, the system is covered over by the infinite regress of meta-layers of descriptions. These twin infinite regresses (in space and time) have as their origin, which is never seen, a singularity which is the noumena that represents the inner coherence of the system itself. That singularity is the source of the unity of all the systems characteristics spoken of by Rescher. Rescher points out that even the systems characteristics appear as

independent of each other. This reminds us of Deleuze and Guattari's dictum[25] that all contents that genuinely emanate from the unconscious must be orthogonal and independent. If there is any relation between the apparently emanating contents then we ar dealing with projections of consciousness and not with the unconscious as such. This is why the systems characteristics are independent of each other. They emanate directly from the essence of manifestation or the unconscious origin of the system in the meta-system or proto-gestalt. They are ultimately empirically derived from the observation of ***organisms***[26]. The systems characteristics form a system which is a complex gestalt of "closely interrelated and mutually complementary" elements; but "the crucial characteristic of such cases is the conjoining of various factors that are in theory separable from one another but in practice generally found in conjunction[27]." So the systems characteristics themselves form a broken whole with no synthetic unity (totalization). Because of the discontinuities within the set of systems characteristics, there is room left for the action of the essence of

25. Deleuze & Guattari, *Anti-Oedipus.* University of Minnesota Press, Minneapolis
26. op. cit. page 12
27. N. Rescher Cognitive Systematization. page??

manifestation. The inner coherence of the system must be inferred from the fractal and temporal patternings, but nowhere does this appear explicitly. This is the realm of pure immanence that never appears except as distortions and breaks in the always imperfect descriptions, that may be fruitfully compared with the unconscious discovered by psychologists in the system of consciousness. The systemic unconscious (un-conscious means non-manifesting) was called by Freud[28] from the first, the *ID*, meaning "it." The "IT" of systems is the essence of their manifestation that no-where appears as concrete foci, but appears globally in the distortions and breaks in system functioning that causes all the meta-levels of description to be at least slightly out of "sync" with the real system.

You may begin to get the idea that the meta-levels of Being, as we see them covertly in General Systems Theory, have more to do with our perception and understanding of the system than with the things being observed themselves. Well, you would be right because meta-levels of Being have to do with the manifestation of everything, including

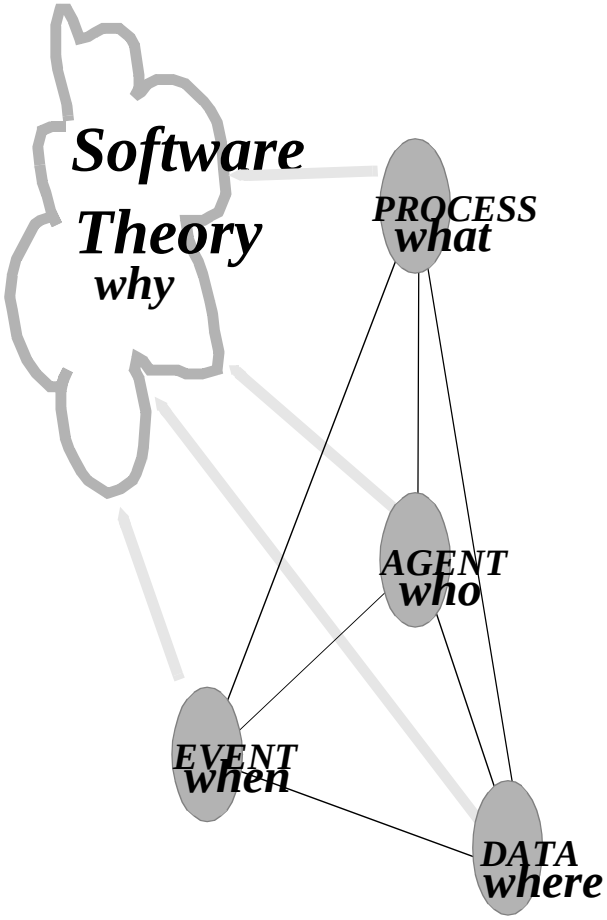
28. H. F. Ellenberger, The Discovery of the Unconscious: The History and Evolution of Dynamic Psychiatry. Basic Books, New York, 1970.

scientific objects. Science believes it can be clever by claiming metaphysical ignorance. But the best of scientific theory ends up mimicking the ontological distribution of entities in the world, and thus begrudgingly gives an accurate metaphysical picture in spite of claiming ignorance. So it is also with GSPS. GSPS accurately portrays three of the four possible[29] different meta-levels of Being discovered by modern ontologists. And it is through this accurate portrayal of these meta-levels that Klir manages to claim the high ground from which he can look down on all the other sciences supported by their implicit technologies. Each science has a focus, and it keeps that specialized focus and works on its object by applying the requisite technology. The application of that technology makes the science into a system, and becoming a system allows it to treat its object as a system. But General Systems Theory correctly thinks of itself at a meta-level above systems. It is the supra-system of all possible systems. It therefore maps out the terrain of all possible systems designs. By mapping this terrain, it allows the theorist to identify a particular instance of an abstract system and to recognize

29. Pure Presence, Process Being, Hyper Being, and Wild Being

the generalization of an instance.

Figure 2: *Essential View Points on Software Design Theory*



An important point is to recognize that software also operates on this third meta-level of Being. It is by representing this third meta-level, that software has its true kinship with General Systems Theory. However, software appears on the third meta-level of Being in a completely different way from General Systems Theory. Software is a form of writing. It is an automated writing. It is writing that can

change itself. Thus, Software is an excellent example in the real world of what Derrida calls "differAnce." In software the operation of differing and deferring appears in the spaghetti code of multiple intertwined GOTOs; in self-rewriting code; in side effects; and most importantly for us, in delocalization[30]. Delocalization is, in effect, exactly the type of phenomenon that Derrida wants to describe with the term "differAnce." Delocalization causes the ideas, or design elements, expressed in the text to be spread out and interspersed within the text. It represents the opacity of the text itself. Any one part of the text seems perfectly clear. But we cannot hold all of it in our memory at once. So the text is not all perfectly available. There is "showing and hiding" as we scroll the source code up and down in our editor. But there is an even more severe "showing and hiding" effect as the software executes. That execution may be very different than one imagines just looking at the text of the program. This is called a bug or sometimes a side-effect, depending on the context. It is the cognitive dissonance between what we see on the screen in the code and what the actual action of the code turns out to be

30. E. Soloway et al, "Designing Documentation to Compensate for Delocalized Plans". Communications of the ACM, 31, No. 11, November 1988, pp 1259-1266.

when it is complied and executed. Getting rid of these bugs, or dealing with side effects, normally determines the extent of delocalization as statements are moved so they do not conflict with each other. Between the code and the action in execution there is a blins spot in which differAnce appears.

So, Software and General Systems Theory are two completely different manifestations on the same meta-level of Being. There may be others at this level, but one thing for sure is that General Systems Theory and software are complementary and need each other. Software objects are always systems. They are inherently means of showing and hiding which are automated. General Systems Theory uses these mechanisms of showing and hiding to simulate concrete systems at some level of abstraction. General Systems Theory rises above all concrete systems by displaying the inherent mathematical structure of the field of all possible systems structures. So, where software is a very concrete manifestation of level three meta-level of Being, GSPS is a very abstract manifestation. But it is necessary to go to this third level because naive systems theories (that do not go to this level) tend to

appear empty. By describing the field of all possible systems structures, GSPS proves useful because it surveys the field of combinatorial structural differences that underlie all systems.

Here we do not explore the fourth meta-level of Being called Wild Being by Merleau-Ponty. This is because both GST and Software are articulated at the third meta-level and do not address the fourth meta-level. But we can point the reader to the work of Deleuze and Guattari who attempt to define a philosophy that addresses this meta-level of Being. This meta-level is the highest articulation of Being into kinds and beyond that lies the unthinkable. The meta-levels of Being are analogous to the meta-levels of learning that Bateson defines and which will appear later in our argument. Both hierarchies have a finite endpoint. For phenomena related to computation that exist at this highest level of the articulation of manifestation we must turn to artificial life and intelligence techniques as examples. Software engineering attempts to separate itself from all techniques of programming that do not have deterministic or easily predicted results. Those techniques tend to appear in Artificial

Intelligence systems.

The four kinds of Being arise from an analysis of modern Continental philosophies where the phenomena of the fragmentation of Being into different kinds was first noticed and appropriated as the central motif of modern phenomenological inquiry. The first philosopher who signaled this possibility was Husserl who was followed by his student Heidegger. Heidegger defined the difference between the first two meta-levels of Being in his seminal work Being and Time based on his teachers definition of essence perception (eidetic intuition) as a mode of apprehension radically different from the modes of apprehension associated with induction and deduction. From that beginning modern ontology has attempted to explore the relation between these to modes of being and their implication for our relation to the world. In the process other kinds of Being were discovered. Foremost in this work has been Merleau-Ponty who translated Heidegger's modalities of Being called present-at-hand and ready-to-hand into the psychologically understandable ways or relating to objects through pointing and grasping in his work The Phenomenology of

Perception. At the end of that work he discovers the next modality of relating to things in the world which I call the in-hand and which he later names Hyper Being. Finally in his last and incomplete work The Visible and the Invisible he defines Hyper Being formally and then points to the possibility of a fourth meta-level of Being called Wild Being. It is only with the work of Deleuze and Guattari in Anti-Oedipus that this last highest meta-level of being is fully explored philosophically. These meta-levels of Being discovered by modern European philosophers have deep significance for our understanding of how the world is projected within our worldview. Here we use them as a back drop for analyzing the relation of General Systems Theory to Software Engineering and show that both disciplines have the properties they do because they exist at the third meta-level of Being.

3.SOFTWARE DESIGN METHODS

Software Design Methods are the major contributions of Software Engineering as a discipline which sets it apart from Computer Science. Sometimes methods are referred to as programming-in-the-large, as opposed to programming-in-the-small[31]. Programming-in-the-small is done using computer languages and by writing source lines of code, one at a time. Programming-in-the-large attempts to see the global picture of what is happening and make it globally coherent. This means rising to higher levels of abstraction where delocalization does not effect design elements. Software Methodologies give the software designer a means of doing programming-in-the-large systematically. There are a myriad of such methodologies being proposed. The methodologies breakdown into two basic kinds:

Understanding the problem.

o Hatley-Pirbhai Structured Analysis [32]

o Ward-Mellor Structured Analysis [33]

o Shaler-Mellor Object Oriented Analysis [34]

31. F. DeRemer & H.H. Kron "Programming-in-the-large versus programming-in-the-small" *IEEE Transactions of Software Engineering*, 12, No. 2, 1976, pages 80-86
32. D. J. Hatley & I. A. Pirbhai, *Strategies for Real-Time System Specification*. Dorset House, New York, 1987.
33. P. T. Ward & S. J. Mellor, *Structured Development For Real-Time Systems*. Yourdon Press (Prentice Hall), Englewood Cliffs, New Jersey, 1985. Three Volumes.
34. S. Shlaer & S. J. Mellor, *Object Oriented Systems Analysis: Modelling the World in Data*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

Designing the Solution.

o SPC-Gomaa (Ada Design & Analysis for Real Time Systems) ADARTS [35]

o Yourdon-Constantine Structured Design [36]

o Neilson-Shumate Object Oriented Design/ Virtual Layered Machine [37]

These are some representative examples of current design methodologies. There is a lot of change in this field today. Many new methodologies are being proposed every year. Major paradigm changes are underway, such as that from functional design[38] to object-oriented design[39]. This field has not yet begun to look at itself critically. However, what we can see emerging is a series of different representations for abstractly portraying all automated real-time embedded systems. Embedded systems[40] function within machines by controlling sensors and actuators. Such systems must necessarily be real-time because they must react to actual events that are sensed and make things happen within the technological system. Thus, Software Design Methodologies

35. Software Productivity Consortium, *Adarts Users Guidebook*. Volume 1 & 2. [ADARTS_GUIDEBOOK_90031-MC] Herndon VA, 1990.
36. E. Yourdon & L. Constantine, Structured Deisgn Book ???
37. K. Nielsoon & K. Shumate, *Designing Large Real-time Systems with Ada*. McGraw-Hill, New York, 1988.
38. E. Yourdon & L. Constantine, Structured Design Book ?????
39. S. Khoshafian & R. Abnous, *Object Orientation*. John Wiley & Sons, New York, 1990.
40. D. M. Auslander & C. H. Tham, *Real-time Software for Control*. Prentice Hall, 1990.

(SDM) are the representation by which software systems will be adapted to myriad applications in the world. In this sense they are a specific type of systems architecture, but with very wide applicability. One might say that it is the class of architectural elements necessary for the technological system as a whole. The technological system as a whole, to the extent it becomes integrated, will be tied together through software which is designed using these representations. For this reason we can think of software as a meta-technology that integrates myriad systems within the technological infra-structure making it a whole.

Software Design Methodologies were not, for the most part, invented by academicians. Instead, they were developed by software engineering practitioners to deal with the problems of conceptualizing the abstract structure of software designs, and to allow those to be recorded and constructed. Software design methods allow major problems with the design of software to be addressed. Normally, for small systems programmers can design the software as they write it. This is called “direct programming” or sometimes “hacking.” They design the application, working within the

mesh of delocalization as they attempt to think about the whole design. This practice, which works for small systems, leads to disaster when large systems are designed. The heart of the problem is delocalization that cannot be controlled under the interaction of dispersed design elements. Any particular design element at the software architecture level may be spread out throughout the code at the implementation level so that its side effects become a totally unknown factor. This fragmentation of the design elements makes it nearly impossible for the design to be recovered from the code. Object-oriented design attempts to reduce the effects of delocalization, but there is always some residue left over within the software system. Design elements which interpenetrate within the code become indistinguishable, so that the architecture is submerged and cannot be reconstructed without a lot of effort. This problem is exacerbated when performance issues produce distortions in the software architecture that compound the problem of delocalization. Software methods solve this problem by using abstraction to let the designer look at the architecture before it becomes submerged and delocalized in

implementation. However, at the software design level there is another important effect: the fragmentation of views of the software design. Software design has been called non-representable by Peter Nauer[41]. This means that the design itself, in its entirety, cannot ever be represented. This is similar to the effect in General Systems theory in which we say that no system can be completely known. All representations of systems are partial, no matter how much they attempt to represent the details. Similarly, representations of software designs are also always partial. However, there are at least four points of view on any software system:

- o AGENT VIEW - WHO? - The view of the hierarchy of independent processing units.
- o FUNCTIONAL VIEW - WHAT? - The view of the transformations performed by the system on either material or information.
- o EVENT VIEW - WHEN? - The view of the temporal ordering of events in the system.
- o DATA VIEW - WHERE? - The view of the arrangement of design elements in the memory of the system.

These four views of every software system are probably canonical[42]. They are the principle

41. P. Nauer, "Programming as Theory Building". *Microprocessing & Microprogramming*, 15, 1985, pages 253-216, North Holland Publishing Company, Key Note Address EuroMicro'84 Copenhagen Denmark.

views from which each software system should be looked at as it is being designed. The design itself will have an endless number of WHYs or reasons for being exactly as it is and no other way. In fact, it is the infinite number of reasons for it being "just so" that cause the design to be ultimately unrepresentable. However, every design can be represented finitely from each of these views in spite of the fact that all its reasons cannot be enumerated. This fact allows software design to occur, but it is always limited by the fact that its representation is fragmented into at least these different views.

The fragmentation of the views onto the software design is the same effect as delocalization at the higher level of abstraction. The unconscious, or non-manifesting essential character of software does not go away at higher levels of abstraction, but merely transforms. Thus, the split in views is a manifestation of differAnce that hides the immanent essence of manifestation. It is the positive appearance of this form of unconsciousness. In effect there is appearing and disappearing if different kinds of design elements as we move from viewpoint to

42. Meaning that these are the only four possible views. This is still to be proven.

viewpoint when looking at any one design. The unconscious appears through the interaction of design elements that cannot appear in representations together yet still interact within the software systems as a whole. It, however, indicates the inner nature of technology as system. Technology forms systems which support various endeavors. Those systems are inherently perspectival[43] because it takes various specialties to keep them operating. The need for those specialties is built into the technology itself because it has so many aspects which are each very detailed, so that no one person can know it all. When we abstract away all this detail at the software design level, the perspectival nature still adheres to the abstract view of the system. This tells us that the perspectival nature is essential, not accidental. In General Systems Theory the perspectives are the various specific Sciences that it attempts to serve. The relation between General Systems and the special sciences is essential. Without the special sciences, no General Science of systems would be possible. On the contrary, the question is whether a General Systems Science is indeed possible or useful. If it is useful, one of those

43. P. R. Fandozi, *Nihilism & Technology*. University Press of America, Washington D.C. 1982.

uses is as a bridge between special sciences. As a bridge, General Systems Science must be built to move continuously between the perspectives of the special sciences. In that movement there is showing and hiding as different specific aspects of things move out of view when we switch from one way of looking at things (physics) to another (biology). The question is whether there is some area of exclusion that never comes into view at all, and is completely immanent. This is possible because one could say that the inner coherence of all the views taken together never appears anywhere in any of the views. Thus, the singularity to which the horns of infinite regress asymptotically approach is probably identical with the inner coherence of all the perspectives of the sciences which General Systems Theory attempts to integrate.

Now this brings us to the question of what software methodologies are, and how they work. Software methods are techniques for rationally proceeding to undertake the design of a software system. As such, they are composed of five fundamental categories of elements:

o Design Elements: The artifacts represented by

the notation and formed by the methods which are the components through which the design is articulated.

o Notations: The specific graphical or textual representations of the artifacts molded by the concepts in the method.

o Methods: The set of concepts and their relations used to understand and formulate the essential features of the design.

o Sequence: The steps in which the methods are applied to the design problem space.

o Heuristic: The rules to guide the process of discovering artifacts which tell you when you are done or what is good, aesthetic, etc.

Xiping Song, a Ph.D graduate from University of California at Irvine, along with his advisor Leon Osterweil has analyzed existing methodologies into the following more detailed set of categories of parts[44]:

Artifacts: which are descriptions/specifications of entities involved in software design activities (e.g.., the interface specification for modules).

Note that we did not decompose artifacts into sub-types. The reason for this is that our earlier experiments have indicated that the most effective way to aid the comparisons of artifacts is to examine the functions of the artifacts.

[Note: Artifacts in the present exposition are called design elements]

44. "A Framework for Classifying Parts of Software Design Methodologies" by Xiping Song & Leon J. Osterweil ISS'92 Second Irvine Software Symposium; Irvine Research Unit in Software University of California, Irvine

Concepts: which are ideas that underlie a Software Development Methodology (SDM). Concepts can be rationales and theories behind a SDM, or strategies and heuristics used in the SDM for specifying/evaluating artifacts (e.g.. abstraction, information hiding).

Properties: which are desired characteristics of artifacts. An SDM is always aimed at producing artifacts that have some superior properties, (e.g.. artifacts should be easy to understand and modify.

Principles: which are concepts used to help in producing artifacts that have the properties desired by customers and designers. An SDM either justifies new design principles or adopts some existing principles as the basis of the SDM, (e.g. object oriented SDMs use principles of information hiding and abstraction as their bases).

Criteria: which are rules advocated for use by designers in deciding what constitutes an artifact. An SDM usually provides a few criteria that serve as the necessary conditions for deciding what an artifact is, (e.g.. Jackson System Design (JSD)[45] defines the criteria for deciding a JSD entity. Rational Design Methodology[46] provides a set of rules used to decide how to decompose a design document into a tree of module specifications).

Guidelines: which are concrete strategies, heuristics or existing techniques advocated for use in identification and specification of artifacts. Guidelines are often described by

45. M. Jackson, *Jackson System Development*. Prentice-Hall International, 1983.

46. D. L. Parnas and P. C. Clements, "A Rational Design Process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12, No. 2, pages 251-257, Feb. 1986.

giving examples or artifacts. (For example, Booch Object Oriented Design (BOOD) [47] indicates that device, system, people, and location might be objects. BOOD also suggests a guideline that using informal English analysis technique can help in identifying objects.)

Measures: which are references with respect to some standard, or samples used for quantitative comparison or evaluation of the quality of artifacts. Some existing SDMs define measures to help in quantifying the degree to which various artifacts demonstrate desired properties, (e.g.. Structured Design defines different bindings (e.g.. functional and logical) and uses them as a basis for a measure for evaluating the cohesiveness of the program design).

Representations: which are the means used for expressing artifacts. They are aimed at improving the precision with which an artifact is specified and at improving comprehensibility of artifacts. They could be languages, sets of diagrammatic notations, (e.g.. data flow diagram), etc.

o Structural: which is a type of representation through which artifacts are captured in the form of diagrammatic notations.

o Mathematical: which is a type of representation through which artifacts are captured using mathematical notations and mathematical operations that are performed on them.

o Linguistic: which is a type of representation

47. G. Booch, *Object-oriented Design with Applications*. The Benjamin / Commings Publishing Company. Inc. 1991.

through which artifacts are captured in the form or language statements.

Actions: which are physical or mental processing steps used for the developing artifacts. As action may create or modify an artifact. An action may also evaluate an artifact and then decide if it needs further development.

o Develop
o Modeling
o Decompose
o Specify
o Define
o Derive
o Identify
o Select

This more elaborate analysis of what actual methodologies contain will give the reader a better idea of what is meant by a methodology. However, in this article we will focus on the five more broadly defined categories of elements described above which are implied within the Song/Osterweil breakdown. Here we are defining a methodology as composed of various methods or techniques, which are sequenced by the methodology, and to which the methodology adds some overarching heuristics, or tactics for discovery of the best possible design. Each method has an associated notation for the

representation of design elements which may take various forms. We assume that a method is composed of a set of actions by which the basic transformations of information defined by the methodology is undertaken. We will assume that the method is a theory of the best way to approach the transformation of the requirements into a feasible design. We will also assume that the method produces and manipulates design elements, so-called artifacts, according to the theory which, when operationalized, has it own properties and measures. The Song/Osterweil more detailed characterization is an important step in the evolution of design methods because they are the first to analyze the existing design methods in this manner. However, in this exposition we will settle for less precision concerning the categories of elements of any methodology.

The reason for separating methodologies into the categories of design elements, methods, notation, sequence, and heuristic is that it gives a clear picture of the structure of the methodology. The methodology is not a monolithic way of approaching design. In almost every case it is a combination of approaches and techniques. Each of these

techniques of approach to the design within the methodology, called here methods, has their own representation which is separate from the theoretical structure of the method itself. A method may have many notations in various forms such as diagrammatic, formal language, mathematical formulation or prose description. The notation chosen is inessential. The essential thing is the set of interlocking concepts that form the foundation of any method. The next most basic aspect is the interface between methods which is usually specified in terms of the sequence of application of the methods to the problem. Methods must work together in order for the designer to get his arms around the problem. Finally, the methods are not algorithms that with certainty produce predetermined results. So, every set of methods that make up a methodology must be accompanied by a set of heuristics, or discovery guidelines, which show the way toward the best design in the face of the necessity for the partial solution of wicked problems[48]. In this case our dissection of methodologies at a grosser level into five elements is meant to focus in on the salient

48. H. A Simon, "The Structure of Ill Structured Problems". *Artificial Intelligence*, 4, 1973, pages 181-201. Also H. A. Simon, *The Sciences of the Artificial*. MIT Press, Cambridge Mass. 1969.

aspects for our consideration here. For other purposes it is correct to go to the lower level of analysis carried out by Song/Osterweil.

We must distinguish between software process, software methods, and software tools. Software process consists of the kinds of work that are done in order to produce software.[49] The activities identified by Song/Osterweil are examples of the kinds of core work undertaken in the design process. These are described by industry standards such as the IEEE *Standard for Software Development Process*[50]. Among these, one kind of work is Software Design which is a crucial transformation of the software system[51]. In doing design work, one uses software methodologies. Methodologies organize the design work, highlighting and abstracting essential features along with the creation of design elements that are represented using the notations associated with the methods. Normally, a methodology is made up of several methods that are sequenced in a particular order. By following this idealized order, one is led to think about different aspects of the system and how they

49. Besides the essential transformations there are other kinds of work such as Planning, Metrics, Monitoring and Control, Configuration Management, Reuse, Technology Infusion, Environmental Maintenance, and many others.
50. IEEE Software Process Lifecycle Standard, P1074
51. These essential transformations are Requirements Analysis, DESIGN, Code, Test and Integration.

interrelate. As the methods are applied, one uses the heuristics to find optimal designs within the design space of all possible systems designs. Designing a system in this way is greatly accelerated when the methodology is supported by automated design tools. These tools enable the methodology and support the design process. Thus, methodologies enhance and rationalize design work which are further enhanced by automation of the methodology by Computer Aided Software Engineering (CASE) tools.

Notice that the methods that are sequenced by the methodology allow the software designer to look at different aspects of the design one at a time. Thus, methods are broken apart from one another by perspectival views. In fact, we can say that the only views a designer has are those listed above (i.e.. Data, Agent, Function, and Event). Therefore, it is possible to see that methods are a way of rendering these views concrete by providing instantiated abstractions or design elements with specific well-defined characteristics for the designer to work with as he is considering those views. In fact, we can go further and say that methods are the bridges between points of view, and that the

sequencing of methods is the movement from viewpoint to viewpoint as one circles around the unrepresentable core of the design. That unrepresentable core is like the coherence of all the specialized sciences. The inner coherence of the views is never seen. However, one has the ability to circle around and around that non-representable core and make it partially visible from a variety of perspectives. The fact that methods are bridges between a finite number of viewpoints means that there are, in fact, a finite number of "minimal methods" out of which any particular methodology may be built. This is important because for the first time we can survey the whole field of methods and say how each particular methodology concocted by an expert fits into the whole field.

Given this framework of viewpoints and bridges between them, I have constructed a model of the field of all possible design methods. This framework[52] draws from a survey of existing methodologies, and attempts to identify "minimal methods" that allow movement from one conceptional viewpoint on the design to another. The results of this survey were very interesting because it showed that

52. See Figure 4.

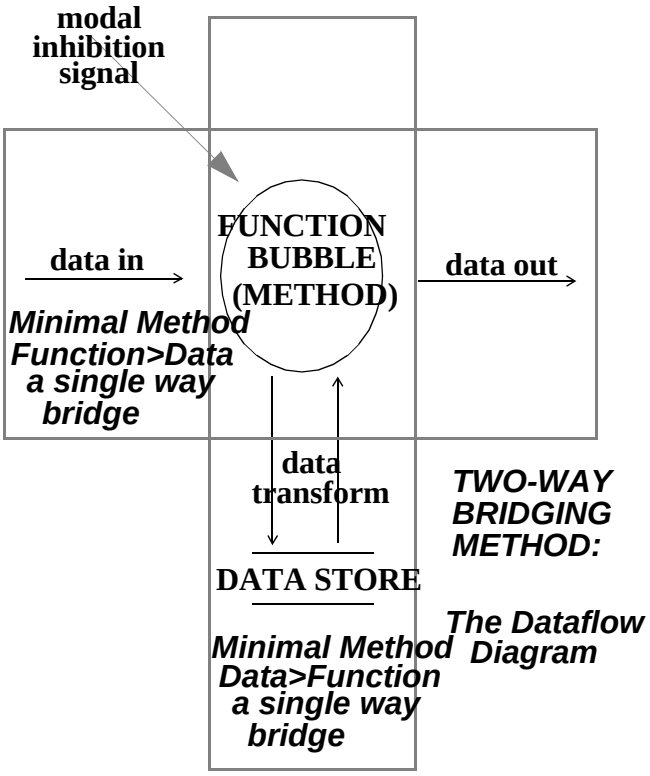
methodologies were amenable to this type of analysis. In fact, it revealed certain properties of well known methods which might not be appreciated otherwise. For instance, the best known of the design methods is the dataflow diagram[53]. It consists of bubbles that represent data transformations and lines that represent dataflows between transformations. Data stores may also be represented. It was quickly seen that the dataflow acted as a bridge between functional and data points of view. And the success of the dataflow diagram can be understood by the fact that it actually represents two "minimal methods" combined into one notational technique. A 'minimal method,' as I use the term, means that it is the simplest conceptual threshold that will allow the movement from one viewpoint to another. It is possible to endlessly add elegant notational or conceptual nuances to a method as seen by many methodologies that represent extremely subtle distinctions. However, there is some minimum theory that is needed to bridge the gap between two viewpoints that represents a critical threshold of conceptual complexity. The minimal method attempts to capture this simplest possible method that still allows the

53. T. De Marco, *Structured Analysis and System Specification*. Yourdon Press, New York, 1978.

transition between viewpoints. In the case of the dataflow, the data lines and transformational bubbles are enough to show the relation between data and function from the function's point of view. Data stores are unnecessary for this purpose. However, to represent function from the point of view of data, the datalines between bubbles are not necessary, and all one needs is the datalines between function bubbles and datastores. Thus, the complete dataflow diagram that has both datalines, datastores, and transformations allows one to consider data from the point of view of function or vice versa with equal ease. Thus, the dataflow diagram is really two minimal methods merged into one technique that has the extra advantage of conceptual reversibility. Looking at the data flow from this perspective shows how perfectly suited it is to the task of representing programs that are generally thought of as algorithms plus data.

Figure 3: Dataflow diagram minimal methods example.

The empirical survey of the available design methods was interesting in that in many instances it was obvious which methods were conducive to transition between viewpoints. In other cases it was difficult to fill in the blanks



Two way method bridge between viewpoin

from existing methods. In those cases, minimal methods had to be discovered which filled the blanks in the matrix. This occurred particularly for the transitions between the data and event views, which was quite unexpected. Another point of surprise was that what constituted minimality was different for different minimal methods. Some were very complex and others very simple so that the resultant field description appeared lopsided. Finally, while several of the minimal methods allowed two-way bridges between viewpoints, in other cases two completely separate minimal methods that

were very different constituted two independent one-way bridges. This occurred between the event-process, event-data, and process-agent transitions. The following is the list of minimal methods generated by this survey of the field of all possible software design methods:

1. Function alone: Functional Decomposition. Functional decomposition is a hierarchical tree of functions and sub-functions which is standard in structured analysis, such as that of Tom De Marco[54].

2. Agent alone: Tasking Tree. The Ada language[55] introduces into programming the concept of tasking trees, which is a hierarchical decomposition of tasks, each of which are an independent thread of execution.

3. Event alone: Interval Logic. Interval Logic was constructed by J.F. Allen[56] as a means of relating events in terms of their temporal dependencies (before, during, after, etc.).

4. Data alone: Entity, Relationship, Attribute Diagram. Chen[57] introduced Entity Relation diagrams as a way to represent the relations between data items.

5. Function --> Event: State Transition diagram. A standard part of the Hatley-Pirbhai[58] and Ward-Mellor[59] real-time

54. T. De Marco, *Structured Analysis and System Specification*. Yourdon Press, New York, 1978.
55. United States Department of Defense, Reference Manual for the Ada Programming Language. ANSI/MIL-STD-1815A-1983 Approved February 17, 1983. American National Standards Institute, Inc. 1980. Ada is a registered trade mark of the United States Government.
56. J. F. Allen, "Maintaining Knowledge About Temporal Intervals". *Communications of the ACM*, 26, No. 11, 1983.
57. ?????????
58. op. cit.
59. op. cit.

structured analysis method is to add state transition diagrams to dataflow diagrams. State diagrams are connected to the dataflow through decision tables and process activation tables. The State Transition Diagram (Finite State Machine) is the traditional way to represent computer algorithms abstractly.

6. Event --> Function: Petri Net[60]. Petri invented these nets to show how control flows are starting to be used more and more to analyze complex control structures. Colored Petri Nets[61] is the preferred representation. Petri nets are composed of places and transitions. Markers move though the places by the firing of the transitions. Petri nets are the dual of State Machines.

7. Function --> Data: Input/Output Dataflow. This is the dataflow diagram with only transforms and dataflow lines.

8. Data --> Function: Objects with operations[62]. This is essentially the data flow diagram with only datastores and transforms now called operations or even "methods." The operations are grouped around the data they change and the entire set is packaged together.

9. Function --> Agent: Mapping of function to task. Process Allocation is described by Mellor & Ward[63]. An explicit mapping is maintained as design progresses between the functional transforms and the discrete processors (tasks) that will perform those functions.

10. Agent --> Function: Virtual Machine

60. W. Reisig, *Petri Nets: An Introduction*. Springer-Verlag, New York, 1985.
61. Ref for Colored Petri Nets ?????
62. P. Wegner, "Object Oriented Concept Hierarchies", Brown University, Working Paper presented at CASE'88.
63. op. cit.

instruction set. This concept was introduced by Neilson & Schumate[64] and says that the implementation component is made up of instructions that solve a problem at a particular level of abstraction. The set of instructions working together solve the problem and constitute a virtual machine. Instructions are, in turn, lower level virtual machines, operations on data objects, or pure data transformations.

11. Agent --> Event: World Line. Explicitly introduced by Agha[65] in his exposition of ACTORS, the world line concept comes from relativity theory which seems to apply to multiple agents acting concurrently within a system. A world line follows all the events that occur to a single agent.

12. Event --> Agent: Scenarios. Scenarios cut across worldlines and follow a series of events in a causal chain. Normally this chain follows paths of communication between the agents.

13. Data --> Event: Data Transitions. This is a very simple method of watching a data value change in a set of variables as the program executes. It has dual configurations in which you follow the data from variable to variable. Or you follow the changes in the data values in one variable and compare that to the changes in other values. This method is not represented in the literature, but is practiced by every programmer who has ever debugged a program.

14. Event --> Data: Design Element Flow. This is a very simple method which says that system states and design element states are coordinated. It has dual configurations in which you compare

64. op. cit.
65. G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge Mass. 1986.

the transitions of design elements as the system state changes. Or you compare the states of the design elements when system transitions occur. This method is also not represented in the literature, but is implied by the fact that design elements are expected to cooperate in a software system.

15. Agent --> Data: Communication Transport Mechanisms. A key feature of the ADARTS [66] method developed by SPC/Gomaa is the display of communications mechanisms between multiple tasks. These are an implementation of the dataflow lines which specify the nature of the data channel as Flag, Semaphore, Mailbox, Rendezvous, etc.

16. Data --> Agent: Data Monitors. Introduced by Hoare[67] data monitors protect data from multiple conflicting access and are used in operating system design to allow different tasks to share the same data.

This set of minimal methods fulfills all the requirements of our description of the field of design methods. It allows each transition to be made by letting one viewpoint take the other viewpoint as an object. Thus, in the dataflow example, process reifies and objectifies data as dataflow lines. On the other hand, data objectifies and reifies process as operations clustered around data that has the sole purpose of transforming the data. The objectification of

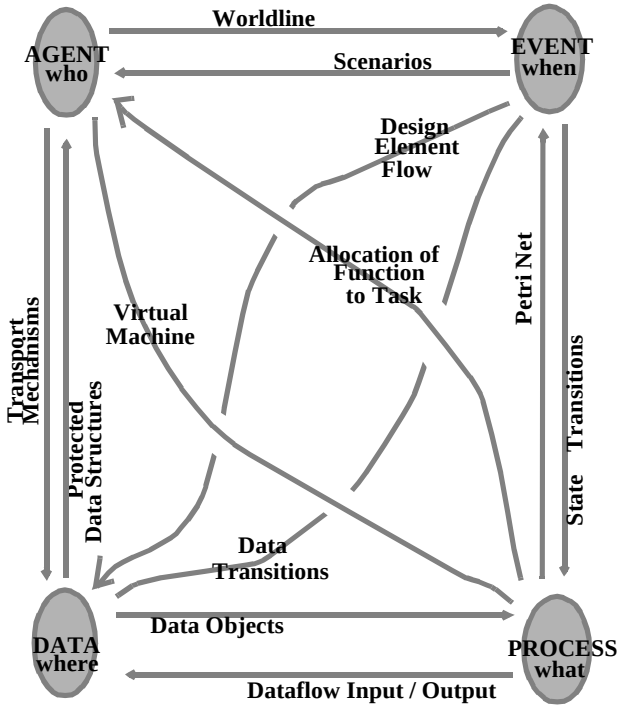
66. op. cit.
67. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, 17, No. 10, Oct. 1974. See also, "Communicating Sequential Processes", *Communications of the ACM*, 21, No. 8, Pages 666-677, August 1978.

one viewpoint by the other is a very interesting process. It has to do with stillness and motion in this case. The objectifying viewpoint sees itself as still, while the elements related to the other viewpoint are seen to be moving. Different criteria for objectification exist in each set of minimal methods. But the end result is that the active viewpoint uses the other viewpoints as its material which it organizes in order to get a view of the system. Viewpoints organize, and in these minimal methods we see that organization at work. Not all the design features of the system are visible at once, but it is possible to circle around a particular viewpoint moving through the other three possible viewpoints one at a time to get a coherent picture of the system from that particular viewpoint. As one rotates though the other viewpoints treated as material for the active viewpoint a coherent picture of the system from that active viewpoint becomes accessible. Then, at some point specified by the overall methodology, one changes to one of the other viewpoints and makes them active to repeat the process. The analogy is Rubric's Tetrahedron[68]. As one moves between active viewpoints, different design features rotate out

68. This is a toy related to the Rubric's cube which is turned to match up the colors on the sides.

of view, and others come into view. The whole thing is never visible at once, so showing and hiding is in effect. But all the views are connected and form a web around the designed software which can be traversed. With enough passes the one begins to intuit the complete picture of the software design which is never actually available. The coherence of that design, which lies hidden beyond the fragmented perspectives from which the infinite regress of WHYs issues, is the unmanifest non-representable part of the design, the essence of manifestation of the design.

Figure 4: View Points Combine to Give Minimal Design Methods



Now this description of the field of all methods was empirically derived. The main reason for doing it was to study the interaction between elements visible in different views. However, once it had been formulated, I began to wonder about the strange lopsided nature of the description of the field. I had expected all the minimal methods to be equally complex, for all minimal methods to support two-way bridges like the dataflow diagram. One good thing was that I could see how many of the minimal methods were duals of each other in the sense used in Mathematical Category Theory. For instance, the state machine and the Petri Net appeared to be duals. It did appear that methods connecting the same two viewpoints were of about the same complexity. It was only as one traversed the field, that the variations in threshold complexity appeared. Yet, there was enough lopsidedness and irregularity in my field description to make me doubt the result of this analysis on purely aesthetic grounds. It is always open for someone else to try to construct a different set of minimal methods that equally provide transition between viewpoints. But, if you accept that there are only four viewpoints on design, and that methods are transitions between viewpoints,

then it is just a matter of filling in the blanks, and this is the set of minimal methods that appeared most likely to me. The nagging question, though, is why the set is so lopsided and uneven. Is this due to my poor selection of candidates for minimal methods, or due to some inherent properties of the field itself?

4. DESIGN METHODS IN A GENERAL SYSTEMS THEORY CONTEXT.

It occurred to me that there was no way to answer this question of the adequacy of the set of minimal methods without a context in which to see these minimal methods functioning. Therefore, I decided to attempt to insert them into Klir's excellent model of a formal-structural system. This would provide a context that should make it clear whether the set fits together or not. GSPS was constructed, with description in mind to facilitate the two great descriptive operations of General Systems Theory: reconstruction and identification[69]. The amount of space dedicated in Klir's book to design of systems in the sense common in software engineering was minimal at best. In fact, it seems to me that the vocabulary is skewed too much toward description and would have to be changed somewhat to become more familiar to the software design community. Yet, GSPS provides something crucial that is missing from the software engineering community which is a universal vocabulary for describing systems. Thus, if it were possible for software engineers to learn

69. Ref reconstruction and identification. ?????

that vocabulary, it would be a boon to the profession. What is really needed is another version of that system which is presented in a way that would be easy for software engineers to relate to, and perhaps some of the terminology changed to be more what they are used to, especially where the different words mean the same thing. For instance, Klir speaks of the generative system, whereas for software engineers this is the software program. It is interesting that what appears as higher epistemological levels in General Systems Theory are more concrete levels from the point of view of software engineering. This is another indication of the symbiotic and complementary relation between the dual meta-disciplines.

Terminological issues aside, the architecture of the GSPS is an excellent view of what a system is, and the only question becomes, how does one use that General Systems view to guide the design of software? Searching though the ASPS book for a place to make the necessary connection between my set of minimal software design methods and the general structure of systems, I was gratified to find, as if the author knew exactly what I needed, a

ready-made place to plug in my set of viewpoints and minimal methods. That place was found in the section on "backgrounds" and "support variables."[70] When one lifts the source system out of the object system, it is by deciding which attributes of the object system will be included in the system to be studied. The object system is a bundle of attributes from which some are selected. How the observer recognizes which attributes form a coherent or "systematic" set is not treated. But, the narrower bundle of extracted attributes is the source system, and for each of those selected attributes, a variable needs to be defined and an observation channel constructed to create the data system which renders the system available. Here, Klir points out a fact usually overlooked, which is that for meaningful measurements to be made, there needs to be other special variables created that provide the context for measurement, the coordinates within which the measurements will be related to each other. These coordinate variables are also attributes taken from the global context of the source system. They must be attributes that are globally uniform. This means that background attributes are really a connection

70. ASPS Chapter 2 Section 2 "Variables and Supports" pp 38-44

between the system and its environment. This means that the gestalt of the system on the background of its environment is formalized by identifying the background attributes. It is interesting that by selecting a particular background for a system, one is, in effect, attaching it to a specific discipline as well so that in using the four viewpoints (data, event, function, and agent), one is connecting the General Systems Theory to Software Engineering. Other selected backgrounds would entail connection to other disciplines. Thus, the disciplines color the background against which the Object system is seen. By switching backgrounds, one immediately switches disciplines. The general point is though, that the particular background attributes furnished by Software Engineering are generic in the same sense that the Object systems of General Systems Theory are generic. Therefore, this connection of backgrounds to foreground attributes is the specific site for the marriage of the two meta-disciplines. Further, we see that the relation of the foreground treated by General Systems Theory, and the background treated by Software Engineering, is another instance of the object/supporting technology dialectical

relationship which is in some sense formalized by this proposed common law marriage of strange bedfellows. The background attributes, once selected, become special variables called "supports." Klir mentions that the normal kinds of support variables are measures of space, time and/or population.

This concept of a "support variable" provided exactly what I needed. Event and Data were already directly translatable, for they were the way time and space were represented in the minimal methods. Data always means where, in the space of memory, a particular pattern of information lies. Time is always conceived in terms of cycle times of the central processing unit (CPU). And when I looked closer at the concept of "population," I saw it could be construed to be made up of my two other viewpoints superimposed. Population usually refers to animals which are autonomous beings that move about independently. Here again, we see hints of Rescher's reduction of systems characteristics to the empirically discovered characteristics of organisms. Demographics follows these movements and categorizes the different types of organisms which are spatially distributed. However, essential to the concept

of population, as it is normally used to speak of organisms, autonomy of action is definitely implied. And all organisms do different things, and have very different behaviors as well as shapes, so the things they do can be construed as their functionality. In fact, there is a direct relation between functionality and intentionality. Functionality is an ill-defined and over-used concept in software engineering. Intentionality really refers to relevance and the distinguishing of kinds, as one's attention moves from one type of thing to another. Organisms all have some form of consciousness, and the focus of that consciousness is their intentionality. Functionality is the focus of the autonomous agents within the population which directly relates to changes in their behaviors. Thus, from this narrow perspective, population can be seen as made up of autonomous agents exhibiting different kinds of behaviors (functioning) expressing their individual intentions[71]. So I could see my way clear to saying that for each viewpoint on Software design there was at least one support variable. The interaction between these support variables is then described by the set of minimal

71. L. Kohout *A Perspective on Intelligent Systems*. Chapman & Hall 1990.

methods. In this way there was a precise interface between software design methods and Klir's formal-structural system.

However, once I had made this connection, I was fascinated by what immediately followed from it. Klir goes right on to discuss what he calls Methodological Distinctions[72]. This boils down to the idea that any particular support variable can exhibit different ordering characteristics. Basically the possibilities are as follows:

ORDERING

- o No Ordering (male/female type-pure diacritical distinctions)
- o Partial Ordering (a depends on b - dependencies)
- o Linear Order (a follows b - complete ordering)

DISTANCE

- o No Distance (do not have a scale)
- o Distance (have a scale)

These types of orders determine the effectiveness of the support variables in locating a measured event/entity (eventity) within the overall system. They form a lattice of possible orderings which has the following

72. Chapter 2 Section 3 Methodological Distinctions pp 44-51

elements:

a) No Order, No Distance: The coordinates are a myriad of distinctions which are not related to each other.

b) Partial Order, No Distance: Set theory produces partial ordering without distance.

c) Linear Order, No Distance: A continuous line exists, but how far apart the points are cannot be determined.

d) Partial Order with Distance: Sets in which you know how many boundaries you have crossed to get somewhere give this type of measure.

e) Linear Order with Distance: This is the type of measure we are used to in mathematics that uses the real number line.

In this lattice position ***c*** and ***d*** are at the same level, lying as two separate routes between ***b*** and ***e***. The methodological distinctions tell you how good your measurement system will be. We naturally assume that full ordering will be available to us in each support that we select. So we assume that we will not have any problems determining the exact position of each eventity in relation to our coordinates. However, we do not always have full choice as to what support variable from which we can choose, and this is why Klir explains the important distinctions between them. If it

hadn't been that support variables were such a good fit for my methods into the GSPS framework, I would not have thought twice about these distinctions between types of coordinate systems. But as I started looking at my proposed support variables and the methodological distinctions, I noticed that two types of support variables existed in my schema:

Support variables with Partial Order and no Distance

o AGENT

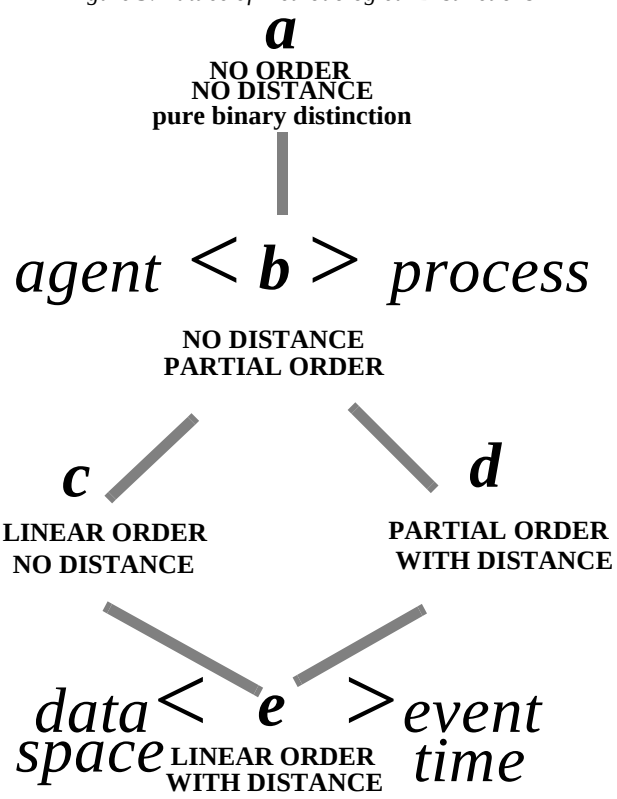
o FUNCTION

Fully ordered support variables

o EVENT

o DATA

Figure 5: Lattice of Methodological Distinctions



This caused me to look deeper into the relation between these two types of support variables, and what I discovered there, I think, is very significant for the development of software design methodologies. Basically what I saw was that there was an irreconcilable gulf between my two types of support variables, and that gulf intrinsically determined the nature of software designs. In building a software system, we start off with requirements which are basically a set of pure diacritical distinctions without order or distance. In systems design we decompose functionality

and pick a set of agents (boxes with multiple processors, processors, tasks within a processor) that become the architectural elements of the system. A miracle of design must occur. In hard real-time systems it is assumed that a certain event has to occur in the full ordering of spacetime. The point is, that there is no way to connect the architecture of agents and the functional decomposition to the necessities of spacetime occurrence. The design must perform a miracle, and there is no real support for that miracle. The designer must find a way to leap the gap from global architecture and functionality to spacetime occurrence, and this is the essence of the real-time design problem. Implied here is that this gap is essential and can never be bridged except through intuition and basically inadequate bridging techniques.

This said, it then quickly became clear that the intermediate positions in the lattice of methodological distinctions were very important. They were, in effect, the generic prototypes of each minimal method pair. And this is very significant because it means that minimal methods have a root in some sort of mathematical necessity. In each case they

attempt to describe an intermediary position between pure partial order without distance and full ordering. Now the lopsidedness of the set of minimal methods started to make sense. The agent-process (functionality) bridge which was composed of function-to-agent mapping and the virtual machine (augmented structure chart) was very complex in order to compensate for the lack of structure to the ordering itself; whereas the event-data bridge could afford to be simple because the ordering was robust. This means that the lopsidedness in the set of minimal methods was a direct result of the expressive poverty of partial ordering for which the agent-function methods attempted to make up in their higher threshold of inherent complexity.

Another important point was that the nature of the intermediate lattice positions could be understood as duals of each other. Linearity without order is the dual of partial ordering with distance. When you see that partial ordering basically means “sets,” or Venn Diagrams, it is possible to see each of these intermediate duals as a use of two sets against each other in the only two possible ways. In partial order with distance, the agent set is used

against the process set, and you get measurement by counting boundary crossings. In the case of linearity without distance, one set becomes the coordinate line, and the other becomes the thing arrayed on that coordinate line. Thus, the second set can be arrayed against the first in only two possible ways. It can become a coordinate line, in which case it has no tick marks to provide a scale. Or it can be used as an alternative categorization of the same elements, in which case it provides the scale without an independent coordinate reference. Either way, to move to full ordering, another element besides the original two sets must be provided. That third element is either an external coordinate or an external scale. This, in software, is provided by the real world itself. The spacetime continuum has the extra dimension necessary for full ordering to appear.

The minimal methods that connect event-data and process-agent strive to fill the gap between the two types of viewpoint. However for hard real-time systems the minimal-methods connecting event-data cannot be relaxed, and so the gap appears as an unreconcilable abyss across which the design must leap. This abyss,

in the heart of the viewpoints is the negative representation as an essential absence or lack of the non-manifesting essence of manifestation from which all the differAnce conjured up by software issues forth. It is an important absence haunting every software design, making it an essentially impossible task which we do anyway using our intuition (which has an crucial unconscious factor) as a guide.

The presence of this lack in the background of every Source system is important for General Systems Theory also. Because General Systems Theory generally detaches itself from all possible backgrounds associated with specific disciplines, that lack becomes observed. General Systems Theory seems to be able to detach itself from that blindspot in the background. But then the blindspot reappears at the center of General Systems Theory itself as the “always already lost” origin of the two infinite regresses of meta-structures and models. The blindspot appears both in the background and in the foreground, and cannot be escaped without lowering our expectations significantly with respect to the robustness of our descriptions of empirical Source systems. It is not much of a leap to realize that the fore

and background blindspots are merely two reflections of a single hidden source from which the whole temporal gestalt is emanating. Not only are we caught in webs of showing and hiding, we are also tied to something which never appears, but constantly distorts and conditions everything that does appear. This is hard for all naive realists to swallow, but is hidden in the very structures that objectivists build because it is an aspect of manifestation or Being itself, and whatever is defined by the objectivists undergoes manifestation or presenceing which even they cannot escape.

Each of the different meta-levels of Being can be derived naturally by first making the distinction of “ontological difference” between beings and Being. Once we recognize Being as different from the beings that participate in Being then we can begin to explore the nature of Being itself. For most of our philosophical tradition that Being was considered as unified and unitary. But since Parmenides it was considered as also frozen and static. It was Heidegger who exploited the insights of Husserl to show that there must be some kind of Being mixed with Time. This dynamic

Being which underlies all processes and support temporal gestalts that last longer than the Now moment was shown to have a radically different nature from the Being assumed by Aristotle and Kant and the other major philosophers in the Western tradition. Eventually it was shown by Heidegger than yet a further kind of Being could be defined called ~~Being~~ (crossed out) which Derrida went on to define as DifferAnce and exploit in his works such as Of Grammatology. Michael Henry also explored this level showing that it had some fundamental non-manifesting aspect which like the unconscious was always hidden from view and only seen as distortions within the things that were manifest. Henry called this always hidden aspect of Manifestation the Essence of Manifestation. As each meta-level of Being is broached it becomes successively harder to think about its nature and consequences for our comprehension of manifestation. The final stage in the unfolding of Manifestation was defined by Merleau-Ponty as Wild Being and explored by Deleuze and Guattari. Wild Being is the substrate of Being within perception and is exemplified by the phenomena of touch-touching itself. In that phenomena there is an opacity which cannot be made transparent to

itself which reveals itself as a chiasm or reversiblity between perceiver and perceived. Delueze and Guattari speak of this same phenomena in terms of the relation between desiring machines and the body without organs or the embedding of partial objects within the rhizome. It is difficult to think about this level of the infra-structure of ideation except in terms of the negation of dualities by the realization that duals imply each other and mutually define each other so that what really exists is the relation between the duals not the duals themselves. Thus at this level everything is composed of partialities and propensities which exist in a Chaotic field which combines order and disorder in which order is embedded in disorder and vice versa. Both GST and Software Engineering attempt to delimit themselves in relation to the meta-level of Wild Being and exclude its chaotic nature from their consideration. For Software Engineering the catch all category for all such phenomena associated with Wild Being is Artificial Intelligence. For General Systems Theory it is Complexity Theory. The limit of GST is systems too complex to be analyzed into component parts and synthesized as formal-structural systems. Such systems must be

considered at a macro level without looking in detail at their parts or must be considered at a micro level and generated by uncontrolled interactions to produce macro phenomena. Complexity Theory assumes that certain emergent phenomena emerge from very complex systems that cannot be reduced by analysis to constituent parts. These emergences are the way that Wild Being manifests in these systems. In this paper very complex systems and their emergent phenomena are not considered further but the reader should be aware of the existence of this further horizon of exploration that opens up out of GST and Software Engineering as we attempt to think what the next meta-level of Being from that at which they are situated must be like.

5. SYSTEMS ARCHITECTURE FROM THE POINT OF VIEW OF SOFTWARE

Now that we have seen how the software methods plug into the support variables of the GSPS architecture, it is necessary to interpret the GSPS semi-lattice of meta-levels from the point of view of software. This interpretation will include the mapping of the software design minimal methods into the support variables so that, in effect, a complete software design framework is produced out of the generalized GSPS structure. Here we are turning GSPS into a specific design system for software. However, since software is also at a meta-level above all the specific sciences, we could be seen as extending the realm of GSPS to produce a broader set of modeling systems, by which discrete simulations might be built. As such GSPS is extended by attaching a certain set of generic modeling methods found useful in designing software, but which could be applied to the modeling of any discrete system. This combination of the two meta-disciplines, which takes into account both foreground, background, and their relations, might be called Software Systems Meta-Methodology[73].

73. There is a paper in the series Software Engineering Foundations on this topic which is available from the author on request.

5.1. OBJECT SYSTEM

The object system is what is differentiated from the environment and designated as a “system.” As has been said, if this is a system instead of an object then it exists in a web of showing and hiding relations as a complete gestalt. As such it is not just a gestalt in space but also in time. If the system can be represented in space alone then a formal system is adequate to describe it, but if the system is dynamic then we need a structural-formal description of it.

The formal-structural description has two components related to its patterning and its formation. Patterning considers order on the backdrop of disorder while Formation considers disorder on the backdrop of order. These are complementary views of the same system which combine to give the formal-structural description of the system.

The “object” system is delimited as a form on the background of everything else. That form may move and change over time and so we might have structural descriptions of it and process models of its changes. Within the form the contents may move and change over time in which case we construct micro-formalisms to

describe transformations in content which delineate the structure of the content and model its changes over time. When we combine these two descriptions of the form and its content then we have a complete formal-structural description. But this is not a description of a system unless we include the showing and hiding webs that the objects within this system participate in through the dynamics of its form and pattern configurations.

At the level of the “object” system we have merely focused in on the target system as a gestalt and discriminated its orthogonal pattern and form components that blend together in webs of showing and hiding to create a formal-structural system.

5.2. SOURCE SYSTEM

Having isolated the object system we are ready to look for its attributes and the backgrounds that give significance to those attributes. As we have seen for the software system those backgrounds become visible as we observe the system from the four canonical viewpoints: Agent, Function, Data, and Event. These viewpoints each reveal a background specific to it upon which the system might be seen. In

the light of that viewpoint on a specific background the system will have certain significant attributes which will appear relevant to our modeling of the system. At the level of the source system we isolate those attributes against the system wide backgrounds. So the source system becomes a bundle of selected attributes in relation to a few well selected global system backgrounds.

At this level the system description begins to ramify. That ramification is precisely the same kind of ramification that occurs in Russell's theory of logical types. In other words we can see the source system as attempting to resolve all the paradoxes that it finds in the object system by applying something similar to Russell's ramified Type Theory. In the case of the Klir epistemological lattice the ramification occurs in a spacelike direction and in a timelike direction. In the spacelike direction variables change on constant backgrounds while in the timelike direction backgrounds change in relation to constant variable sets. At the point of fusion both backgrounds and variable sets vary together. The spacelike direction relates to the structuring of the parts and wholes within the system into echelons of holons. In this case

we are calling the “structure” the combination of formation and patterning within the source system. The timelike direction considers the models by which different structural (form/ pattern) configurations are changed over time. A model is the same as a structural configuration in space only applied to time. Models are composed of processes and sub-processes. During a certain sub-process a specific part-whole holonic structure is in force and a set of transformations are applied to it. When this sub-process changes to a different sub-process then a new set of transformations are applied to the holonic structure. Transformations may be of a number of different kinds. Transhaping changes the form from one formation to another. Transcription which changes the patterning from one ordering scheme to another. Transfiguration the changes of both form and pattern simultaneously. We would call transubstantiation the change of the medium which supports the form and pattern configuration. The change of form, pattern and substance all simultaneously we might dub transmutation. All these kinds of transformations are possible that bring us from one form-pattern configuration to another as

the system changes as a dynamical process according to some meta-model of changes. We can see this in terms of the system "working" where it brings to bear different kinds of work at different times in a specific sequences in order to effect changes. We notice that "work" can effect changes in pattern, form, or substance according to goals. When we look at a system from a process perspective we project these goals in order to see the coherent kinds of work being applied to effect transformations.

Since we understand that space and time dimensions are intimately related in a fusion which we normally think of as "spacetime," it is clear that the structural and process modeled dimensions of ramification must be interlinked. We see exactly this in Klir's epistemological levels where there are certain nodes that mix the spacelike and timelike aspects of the system description emphasizing one or the other. This means that the ramification of the structures and models in the epistemological lattice are mutually entailing which means that the spacelike characteristics articulate the time like characteristics and vice versa. Now we also notice that of our four viewpoints there are two that represent spacelike (Data) and timelike

(Event) characteristics of our design. This indicates that the embedding in spacetime is complete in as much as the total dynamic system is encompassed by structures and process models but also there are design elements that specifically embody the architecture of the system in spacetime. We also need to note that spacetime has a dual called timespace[74] which looks at the system and the propagation of events within it from a causal perspective. Thus the MATRIX of spacetime/timespace has two meta-viewpoints that encompass the specific background viewpoints on software design. The epistemological lattice exists as a half way house between these two levels of viewpoints[75]. In other words the epistemological lattice exists only as a spacetime structure which encompasses the four viewpoints on software design two of which embody the spacetime aspect of the system itself. We might ask why the spacetime component of the system differentiates itself on these three different levels. We note that embodiment within spacetime through the DATA and EVENT viewpoints is

74. Normally called Minkowoski "spacetime."
75. Spacetime / Timespace viewpoints ***or*** Design viewpoints including Agent, Function, Data (space) and Event (time).

encompassed along with the AGENT and FUNCTION viewpoints by the structural and process model ways of looking at the system as a whole which in turn gives us a view of the whole within the ultimate background of spacetime/timespace. If we want to look at the causal relations between elements through the timespace view then in effect we switch from emphasizing the independence of the structures and process models to the emphasis of the fused structural-processes or process-structures that exist in the epistemological lattice. At the level of viewpoints this fusion comes about though the agent and function views in relation to timespace embodiments by design elements. The timespace view is almost like the fusion of the spacelike and timelike orthogonal components that can be seen as separate in spacetime. The epistemological lattice allows us to see this fusion when viewing the system from the outside in terms of the fusion of process models and structures of pattern and form.

5.3. DATA SYSTEM

Klir calls the data system the observed data streams flowing though variables (that characterize attributes) and supports (that

characterize backgrounds). Here he has already reduced the system in his mind into a data image. But we have seen that a software system has four viewpoints of which data is only one. All of those can be reduced to data images but this reduction does violence to our understanding of the workings of the system. Instead we prefer to understand that these data images preserve the different kinds of information related to the different viewpoints that were projected on the object system to produce the source system. At the data system level we are reducing the system to streams of data that are recorded and analyzed in order to discover the workings of the system. But from a software point of view we are seeing the software system as flowing streams of data related to the different views on the dynamic system we are designing. There is information about the Agents within the system, about the functions they are performing, about the events that are occurring and the data that the system is consuming and producing. We note that all this data about the different viewpoints are mixed together and they ramify and fuse in exactly the same way as the source system did in spacelike and timelike dimensions. The difference is now that the whole system is seen

as a set of variables taking on values at specific locations in time and space. If we are studying such a system then we wish to understand the configuration of structures and process models that are implicit in the multitude of changing patterns of information in the forms of the variables over time. If we are designing such a system then it is our hope to get these changing patterns of data in the forms of the variables to behave correctly which is not an easy task.

What we notice at the data level is that the whole object system which was reduced to attributes has now again been reduced to a string of memory locations and their values over time. The next question becomes for the Systems Scientist if we can create a generative system that simulates the data values changing over time. For the designer the question is whether it is possible to write a program that creates precise configuration of values not as a simulation but as the origin of that pattern.

5.4. GENERATIVE SYSTEM

The program is the generative system that produces the required data streams when it executes. The program can be either compiled or interpreted. The difference is whether it

executes in small snatches or as a monolithic block. An interpreted program is more malleable and flexible but is normally slower because it is compiling as it is going along. Either way the program when it is executing becomes pure behavior which acts blindly on the instructions that occur in the source code. It produces configurations of changing values in spacetime and if those correspond to those that are needed then we say that the program is running without defects. To write a program that imitates a value sequence exactly is taken as demonstration that one understands that data sequence. This is the notion of understanding within the objectivist viewpoint on existence. This is a very limited or one dimensional view of understanding. Exact imitation is not necessarily a display of understanding. Instead going beyond the information given is understanding. In other words real understanding is not merely the production of a frozen image of the thing being understood that can be made perfectly available as something Purely Present, but must exemplify a process of moving beyond what is given through the application of a hermeneutic circle which interprets what is given in different contexts to give an overall interpretation which is never

complete. So we should posit a stage beyond the generative system. That next stage is the Knowledge stage. It moves beyond the epistemological lattice for general systems into Knowledge systems which exhibit artificial intelligence and life. Klir indicates this level when he deals with goal seeking systems which use feed forward and feedback separately or together to produce meta levels of the generative system which display goal seeking or adaptive behavior. He ends that exposition with an explanation of the autopoietic system whose goal is to maintain its own organization and especially its boundary with the environment.

But here at the generative level we must attempt to put our design for the dynamical formal-structural system into a program that produces the right data pattern. That program when it is not executing is itself a data pattern as well. The data pattern of the designed program exemplifies the interembedded viewpoints of Agent, Function, Data, and Event. When we reduce the design to a program then delocalization begins to play a role as design elements actually interfere with each other within the source code of the

program. It is the warps of delocalization which are increased exponentially as performance characteristics become more and more important. These warpages account for the difficulty of understanding code and many of the errors that creep in despite the rigors of structured programming and other good software engineering practices.

What for General systems theory is the most general level of understanding of the system is for software engineering the most concrete where the system is represented as an executable program. Software has its own special ontological basis which exemplifies what Derrida calls DifferAnce and makes it different from most things in the world that are either like nouns (frozen objects) or verbs (processes). The nouns are locatable in the spacelike dimension whereas the processes are locatable in the timelike dimension. Software with its branches and while loops that alter sequence instead has traces which place it between a noun and a verb. We can signify it with words like ***shape*** which can function as both a noun or a verb. The trace is the sum total of the side effects or mutual interferences of the running program. We look at the listing

and compare it to the trace that occurs during execution. So . . . Shape shapes. The source listing executes and produces a trace. We trace our way thorough the source listing looking for what it is doing when in order to try to understand the complexities of its mechanism in operation. We might say . . . Trace Traces. There is a cognitive dissonance between the cumulative side effects as observed in execution and the flat text of the source code of the program. Due to delocalization and the mixture of design elements, that object oriented design can only partially stop, it is very difficult to understand the homeomorphisms between the two kinds of trace (noun and verb). We reach the point where we are lost between them, where we enter the realm of tracelessness in which it is impossible to decide what in the program is leaving a certain side effect. As we approach this undecidability we get closer and closer to the actual essence of software.

Attempts are being made to come to terms with the nature of software and its strange essence. The seminal article was written by F.P. Brooks called “No Silver Bullet.” A recent follow up article by R. G. Mays called “Forging a Silver Bullet from the Essence of Software” continues

the introspection of the discipline into the nature of the peculiar object they are engaged in creating. The author has presented philosophical arguments that attempt to define the essence of software in the article “Software Ontology” which is the first part of the series on Software Engineering Foundations. In that article the author identifies the object of software with the third meta-level of Being called Hyper Being by Merleau-Ponty which exemplifies what Derrida calls DifferAnce with its chiasmic attributes of differing and deferring. This definition draws on work in modern ontology rather than the old Aristotelian roots of ontology that is used by Brooks and Mays. But if we look at the work of Mays we see that he points out three basic sources of the essence of software:

- **Conceptual Content**

  “A software entity is characterized by concepts that come from both the problem domain and the surrounding software entities with which it interfaces.” [page 21-22]

  I would go further to say that software is in every case an embodied theory. Besides the concepts that Mays mentions there is the

theoretical concepts that control the structure of the software itself that is inherent to the software which are represented by the concepts in Design Methodologies.

- **Representation.**

  "The concepts of a software entity are expressed as representations of both the data it uses and the function it performs." [page 22]

  I would go further and say following Naur that software design is inherently non-representable in toto and that all representations are partial. Those representations do not just represent Data and Function but also the viewpoints of Agents and Events. These viewpoints are "canonical" in that they are sufficient to represent the essentials of any real-time software design. Thus we would include the viewpoints and the minimal methods that span between them within a finer definition of the software essence.

- **Multiple Subdomains**

  "A software entity performs functions that consist of transformations on its data, based on conditions present at the time of execution. The presence of conditions splits the input

domain into multiple subdomains of the function." [page 22]

This is the combinatorial aspect of software that makes it impossible to formally prove any but the simplest systems. This also is the connection between the software and events, data, actions and transformations in the real world to which its events, data, actions, and transformations must be harmonized to operate properly.

Besides the splitting of subdomains over inputs of data to transformations there is also splitting of subdomains over event signals. Thus the combinatorial explosion of possible states is intimately related to the embedding of the real-time software system in spacetime.

- **Delocalized Incarnation**

To the three major aspects of the software essence mentioned by Mays I would add the effect of delocalization which causes the design elements at the conceptual level embodied by partial representations to be smeared out within the actual source code that incarnates the software design. Mays theory of software essence seems to ignore the incarnation of it within the text of the source code which is then

compiled and executed. The major effects of cognitive dissonance occur because of our inability to see the relation between what is in the delocalized source code constructs and what actually occurs when the software is run. Also this is the place where the differences between Higher Order languages occur and their relation to assembly language and binary code. All software can be represented by the three constructs of structured programming (sequence, if, and while statements) but all languages have a bewildering array of constructs which gives rich choice as to how to implement any given feature of the design.

We can augment Mays' summary by adding to his words saying, "Thus the software entity is in essence a construct of interlocking concepts characterized by a conceptual content" revolving around a non-representable kernel of the design "derived from its problem domain and the milieu of other software entities with which it interfaces" as well as software design specific constructs "by" partial "representations of its concepts both in the data it uses and in the functions it performs" as well as in terms of agents and events", and by the multiple subdomains of its input domain that

characterize the different transformations that will occur, depending on the conditions that are present during execution" which is carried out by delocalized design elements embedded within a text that is compiled and run which creates cognitive dissonance between the text and the actions of the program. [page 24-25]

From these he derives the following inherent properties:

1. The conceptual construct of the software is held in the developers thinking. Therefore the Software is Malleable and Changeable. (conceptual content)

2. The conditions in the software combine multiplicatively. Therefore the software is complex. (multiple subdomains)

3. The software representations are a "crystallization" of the conceptual construct. (representation)

4. The developer must anticipate the behavior of the software beforehand under all cases and conditions. (multiple subdomains)

5. Software is more broadly conceptual than mathematical or graphical. Therefore software is unvisualizable. (conceptual content)

6. Software development is an intensive activity of thinking. (multiple subdomains)

7. A higher-order verification coccus in the developers thinking. Formal verification is

done relative to the software representations. (conceptual content)

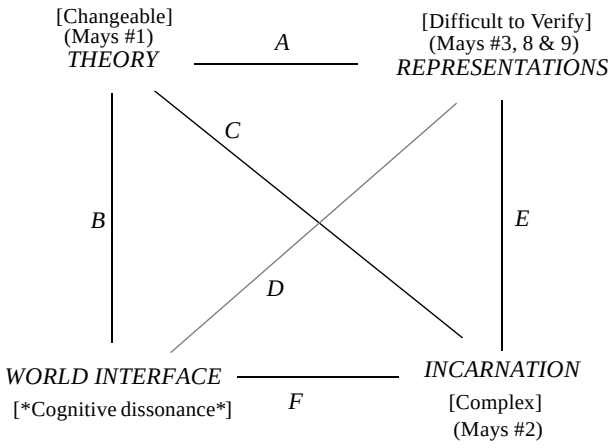
8. Programs are objects in the world and operate in the world. Therefore software needs constantly to conform and change. (representation)

9. Software will always be an asset that must be maintained and enhanced. Therefore software developed is primarily though incremental enhancements. (representation)

10. The first task of development is to re-enliven the conceptual construct in the developers thinking. (multiple subdomains)

This series of deduced properties is very uneven. We could instead look at the minimal system of properties that occur in the essence of software constructing a minimal system.

Figure 6:



Mays draws the conclusion that because software is primarily theory that it is very changeable and malleable. Also because

software has a world interface that is complex and that combinatorially explodes then it will be very complex. Representations need to be constantly verified in the face of this change and complexity. And as we have added the software incarnation leads to cognitive dissonance between source code and execution side effects. When we combine these essential factors we get even more difficult problems rearing their ugly heads.

A. Theory <-> Representation

In Mays's scheme Theory does not interact with representation at all which is very odd. A non-represented theory is completely ethereal. Such designs only exist in the heads of the designers, where a good many designs of actual programs continue to live to this day. Without representation a theory cannot be shared or recorded or adequately studied even by the original designer himself. But these attributes are separated because the whole design theory cannot be captured by the representations. Representations are always partial and as we move from one representation to another different parts of the design appear and disappear. This is because the theories of

design must be organized around the canonical viewpoints that organize the universe of discourse within which we build up our theories of designs.

B. Theory <-> World Interface (Mays #5, 6, & 7)

Theories are abstract precisely because we wish to cover the multiple subdomains of inputs and represent the system that can cover them nicely. But due to combinatorial explosion we cannot actually test all the interactions of constructs that we can finitely write down to occur in the operation of the software. Software cannot be adequately portrayed either graphically or mathematically because it has a behavioral aspect that cannot be portrayed adequately by these means. When we look at representations we are not visualizing the software itself but partial representations of aspects of it that can never be brought together in any satisfactory way to tell the whole story at once as graphics attempt to do. Nor can it be formally proved due to combinatorial explosion of states. Theories are by their nature abstract which as its good points and bad points. The good point is that we can say

general things that apply in most cases about the software through the language of software methods. The bad point is that the world is full of exceptions that must all be handled and software theory cannot do that they must be handled at the level of implementation. The world as it exists as a collection of states of affairs presents software that tries to harmonize with the world a very severe problem.

C. Theory <-> Incarnation

Theory as design elements are encoded into software source code that embodies it and actually runs when it is compiled. Due to delocalization which smears the design elements out in the code it is very difficult to maintain the design once the code has been created. The code is in constant flux and the interference of design elements with each other especially in high performance code can cause design elements to be severely mangled in the this process of incarnation. Code and Design theory are fundamentally at odds and the code always wins because it actually does something whereas the design only makes things understandable to the humans that are creating the program.

D. Representation <-> World Interface (Mays #4 & 10)

Designers must attempt to use mental simulation to anticipate he workings of the software. That mental simulation needs to be revisited over and over again and remain true to the software design and the world that the program is going to function in. The representations no matter how augmented cannot show the complexity of the world completely. All that they can do is provide a gloss that stands halfway between the world and theory that is manipulable by the designer.

E. Representation <-> Incarnation

Representations by minimal methods create abstract design elements that gloss implementation issues which in turn gloss the actual implementation constructs at the level of delocalization which in turn glosses assembly code and which in turn glosses binary instructions on a particular hardware platform. What we have is a series of levels of representation. Within the design level we have hierarchies of Data, Events, Functions and Agents. These levels of design and implementation can be very complex and

stacked deep. But each level must be perfectly matched to those above and below it for a good design to occur and for a complex system this is very difficult to do and sustain once accomplished in the face of changes.

F. World Interface <-> Incarnation

Not only is the world complex and unpredictable and the Incarnation of software on particular hardware platforms complex but these two things add together to produce a super complexity where the hardware platform is engaged in the world which the software is attempting to control things and make them play together properly. In another paper that the author has written this role of software is called meta-technology. Software binds together different technological structures into a meta-system which it attempts to operate so that harmony between the systems is maintained. It is the role of software to do this. So software is incarnated in the technological world as a meta-technology that makes other pieces of technology play together and integrate while performing complex tasks that can be safety critical. As such the software is embedded in spacetime/timespace making

specific things happen at particular points in time and space among the hardware ensemble. Bridging the gap between the incarnated theories of its design and its embedding in the world in a way that works properly is a very difficult task for something as brittle as software.

We have gone into the nature of the essence of software to show what the generative system looks like from the point of view of software. The essence of the generative system relates our theories of the world to representations which in turn are incarnated within the world and tested against that world. Previously I have said in another paper that this gives software many of the same attributes that appear in the philosophy of science as problems. Science produces hypotheses from theories which it tests. Software produces representations from theories which it incarnates and tests against the world. So software has similar problems determining what is a good theory and how should it be tested. The difference is that with software you get more immediate feedback and one is not trying to extend knowledge but merely embody knowledge.

What is interesting is that one cannot

demonstrate understanding by mere imitation. All that we have said about software merely shows it to us as something that imitates the world's states in order to harmonize with it. But imitation even though it is a sincere form of flattery does not necessarily display true understanding of the world and it is only by embedding understanding in programs do they become less brittle. Software by itself is very brittle. Anything that is not explicitly programmed into it causes it to fail. In order to create robust programs we must produce those that imitate the living and the knowing things in the universe. That is we must strive to make them true systems that imitate the organisms that Rescher points out are the source of our ideas of systems. Thus to give robustness to software meta-systems we would extend Klir's epistemological hierarchy by placing a level of Knowledge beyond the generative level. To display understanding we must go beyond the mere imitation of data systems and produce systems with knowledge and flexibility. Thus we arrive at the need for a living/cognitive epistemological level beyond the generative level.

5.5. ARTIFICIAL LIVING KNOWLEDGE SYSTEM

To have deep understanding of a generative system requires a knowledge representation scheme to be overlaid on the generative system. That supplemental system displays understanding of the workings of the generative system. It needs to ultimately be living/cognitive or what is called Autopoietic. Autopoiesis means self-producing or self-organizing. Thus we posit that the next level is most like an organism that is the root metaphor for the system. It is not just a knowledge level added to the generative but the knowledge is activated by being the self-knowledge of an autonomous being. This level actually allows us to understand software better because it is the next higher meta-level above software called the proto-technical and operating at the next higher meta-level of Being which is Wild Being as defined by Merleau-Ponty.

When we think of software we notice that the attempt is made to define it in such a way to get rid of all the paradoxes like self-modifying code and spaghetti goto statements and others. When we move to the Artificial Intelligence and Life level beyond software what we see is a

mosaic of techniques with nothing like methodologies for us to hang our hats on. Each AI or ALife technique competes with all the others in a bewildering array of sophisticated but very basic programming techniques mostly realized at the implementation level. After studying this area for a long time I realized that there was a reason there were no equivalents to minimal methods for AI and ALife. That is because all the paradoxes that were pushed out of the software layer by the discipine of Software Engineering were pushed into AI and ALife. Each of these techniques revolved around some paradox in the software layer and because they were paradoxes they could never be resolved into a simple method that is easily represented. All the monstrous aspects of software are collected here and combined to create specific techniques that will use the side effects of software to create imitations of life or cognition.

Another point about this level is that it uses software as a enabling machine instead of hardware. Because of that it is free to create theoretical structures that are completely disconnected from reality. Thus Virtual worlds arise as the abodes of artificial living and

intelligent creatures that can be completely disconnected from any kind of recognizable reality enforced by the world we live in mundanely. When this detachment from reality is combined with network technology then you get the advent of cyberspace as the realm of all possible virtual worlds. Within these worlds artificial intelligent and living creatures roam which will be created by the opaque AI and ALife techniques that arise from the paradoxes in the software layer. Combinations of opaque techniques will render these creatures even more opaque and incomprehensible. Thus we are engaged in creating alien creatures within our virtual worlds which we can never understand. They are inherently incomprehensible since they are created using all the techniques banished from software engineering because they are not trusted to produce assured results in the real world.

Between the fantasy virtual world and the real world stands what Geleterner calls the “mirror world[76]” which attempts to render an image of the real world in virtual reality. Mirror worlds stand between the real world and the fantasy worlds disconnected from reality. Mirror

76. Gelertner. *Mirror Worlds*

worlds give us more knowledge about the actual world than we would normally possess. They are worlds with superabundance of information and real-time connection to the actual world. They are the mirror between our world and the fantasy worlds that depart from reality in significant ways. We can say that the mirror worlds are super-real and form the reversible interface between reality and irreality. For instance a fantasy world may be a world where a fundamental assumption that is made in the designated as real world is changed to see what would happen. These fantasy worlds give us the possibility of conducting experiments in worlds that do not exist which will shed more light on the world that does exist thought intersubjective agreement. It is though mirror worlds and fantasy worlds that our ability to socially construct worlds is unleashed into realms that it was impossible to enter before. These mirror worlds and fantasy worlds will have a profound impact on the designated as real world as a hyper extension which when treated as part of the designated as real world actually has profound effects on that to which it is supplemented. This is because all of these worlds function in the realm of Hyper Being which as Derrida has shown has the form

of a supplement which changes the meaning of the thing to which it is attached.

At the this level generators become imitations of living knowning organisms. That is they imitate the most sophisticated systems we know which are living creatures. Thus it is only at this level that we have a true attempt to portray systems in relation to the root metaphor of organisms with cognitive capacity. These organisms have a fundamental ability to learn and adapt. And this must be taken into account in our model. Therefore an important part of this level of manifestation are the meta-levels of learning which were first defined by Bateson[77]. There are four of these meta-levels of learning which scale the ladder of meta-levels until they reach the unthinkable which lies at the fifth meta-level beyond all forms of learning.

5.5.1.LEARNING SYSTEM

The knowledge system may learn about other systems or may expand to cover a domain of systems rather than a single system of a particular kind. Thus Learning systems supplement Knowledge systems. When

77. Bateson *Steps to the Ecology of the Mind*

software systems display learning then they cease to be fragile with respect to changes in their environment. A learning software system may also exhibit this learning with respect to itself producing internal images of itself and learning about itself.

5.5.2.META-LEARNING SYSTEMS

These systems as Bateson shows learn to learn. Learning to learn means exploring new ways of learning. This allows such a software system to cope with discontinuous changes in its environment and within itself. When we learn to learn we increase our learning capacity and also gain new learning skills. Such a software system would be very robust with respect to its environment being able to cope with environmental changes and changes in itself that are unexpected.

5.5.3.META-META-LEARNING SYSTEMS

Learning how to learn can be supplemented by Learning at the next meta-level which means changing paradigms of learning how to learn. There may be different paradigms of how to learn to learn which is to say different approaches to learning to learn. At this meta-

meta-level the difference between self and environment become irrelevant. The environment and the self is considered a single meta-system where the environment learns from the self and vice versa. At this meta-levels the differences in paradigms in learning become important and the ability to switch paradigms of learning so that new self-other configurations become possible becomes important.

5.5.4.META-META-META-LEARNING SYSTEMS

Bateson says that the next level is one in which ones whole worldview changes and that this is the highest meta-level of learning. Beyond this is only the unthinkable. It is at this level that the projection of the world by the self-other meta-system is accomplished. The key feature of this level is the appearance of the emergent event. The emergent event is the possibility of a genuinely new thing to come into existence. A meta-system that operates at this meta-level could handle the appearance of the genuinely emergent event. The genuinely emergent event is defined as one that moves thought all four meta-levels of Being as it enters the clearing-in-Being and becomes part of the World.

An example of a Meta-meta-meta learning system is Western science. In school we are taught things in a certain pedological style. But as we encounter different teachers we realize that there are different ways of learning and we attempt to learn how to learn in these different ways. For instance, there are ways of learning suited to those who are language oriented, graphically oriented, and kinetically oriented. But we may combine these different ways of learning to achieve particular learning effects that are difficult to achieve in any other way. As an example, audio visual materials may be combined with an exercise. But eventually as we begin to achieve mastery of subjects we realize that we need to produce our own synthesis of the materials in order to show mastery. These syntheses appear like paradigms in that they go beyond the information given to posit theories which are not contained in what we have learned to learn. When we can advance these paradigms then we have in effect reached the fourth meta-level of learning where we advance the state of the discipline in which we are engaged. Finding these cutting edges at the fourth meta-level of learning is very difficult. In fact one can say that the whole problem of intellectual advance

is to locate these cutting edges and make progress with respect to the disciplines at those edges. Persons who do not learn to learn to learn to learn cannot locate these cutting edges. Those do locate them and contribute to our understanding at those cutting edges are the ones who bring genuinely new things into existence. They are the ones who transform the world.

5.5.5.THE UNTHINKABLE

The unthinkable is the meta-level beyond which we can create learning representations.

Notice that we have gone beyond Klir's original formation to add levels of learning until we reached the unthinkable. We note that the unthinkable is equivalent to the infinite meta-levels to which structural and process models ramify and fuse.

We have also noted that when we reach the infinite meta-structures or meta-process models or the unthinkable we have reached a point identical with the "essence of manifestation" described by Henry that is the point of pure immanence which never manifests.

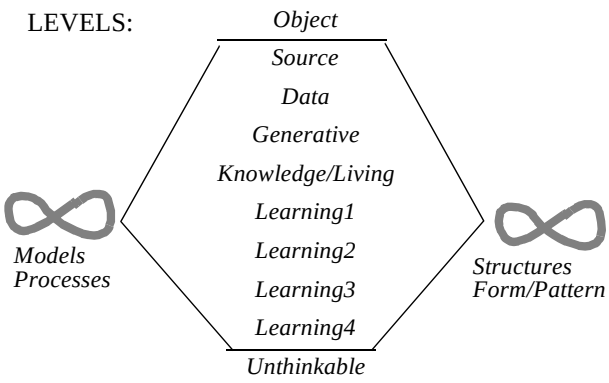
The unthinkable may be considered identical

with the Buddhist non-concept non-experience called Emptiness. Emptiness is itself empty. It is the expression of the absolute middle between all nihilistic opposites. Emptiness is the center of the vortex around which the dynamic of worldview projection at each of the meta-levels of learning revolves. Understanding Emptiness is essential to understanding the projection of the worldview because Emptiness balances the whole action of worldview projection.

5.6.WINGS TO INFINITY

Now we will explore each of the wings that take us to infinity of process and structural meta-levels and see how they function at the multiple levels of the epistemological hierarchy. In what follows the word LEVEL can be replaced with any of the following levels we have discovered:

Figure 7:



5.6.1.STRUCTURAL LEVEL SYSTEM

For software the structure appears as multiple whole-part relations exemplifying the relations between patterns and forms.

5.6.2.META-STRUCTURAL LEVEL SYSTEM

These whole-part relations become ever more inter-embedded. At the first level of inter-embedding there are structures within structures.

5.6.3.META-META-STRUCTURAL LEVEL SYSTEM

At the next interembedding level there are structures within structures within structures. This regress is infinite because we can imagine structures embedded within each other to

infinite levels of logical typing. Ultimately these meta-levels of structures approach the unthinkable, which is a complexity of structure beyond which the human mind cannot conceive.

5.6.4.MODELED LEVEL SYSTEM

Models are the temporal structuring by which the spatial structuring is controlled and changed over time. We call these process models.

5.6.5.META-MODELED LEVEL SYSTEM

We can think of processes within processes controlling structures over time.

5.6.6.META-META-MODELED LEVEL SYSTEM

There is also an infinite regress for models of processes as we can think of processes within processes infinitely. These also approach the infinity of meta-levels of process which is unthinkable.

5.6.7.FUSION OF PROCESS AND STRUCTURE

There is an interference between our structural

and process model reifications which sees them as fused. This fusion represents the timespace causal view of the system. In one fused view structure dominates time whereas in the other time dominates structure. These are equivalent to the proto-imaginaries found in Spencer-Brown's Laws of Form. We follow Merleau-Ponty in calling these points of fusion between process and structure chiasms or points of reversibility. In fact, we will coin a new term called intaglio for the fused relation between process and structure. Intaglio is the engraving of an image within a stone so that it appears three dimensional usually though the other side of the transparent stone. Many times the intaglio is frosted to produce the appearance of solidity to the image. There are sculptures that exist made of glass where intaglio is used on both sides to give the appearance of intertwined figures connected thought the medium of the glass. Many times these are figures of men and women intertwined in some exotic fashion. In other words in these intaglio sculptures what exists is a fusion of the figures thought the connecting medium. The figures themselves have no reality other than the medium that holds the carving of the intaglio. So it is with the fusion of process and structure. They do

not exist as separate entities but only exist as the chiasm or reversibility between them. We can talk of this fusion at three levels.

- **PATTERN/FORM CHIASM = structuralized forms**
- **LIVING/COGNITIVE CHIASM = autopoietic systems**
- **SOCIAL/PSYCHIC CHIASM = reflexive systems**

Here we understand that form and pattern together produces structures of forms and that processes model these over time. But form and pattern also have an intaglio relation in which one cannot be completely separated from the other. In that relation they exhibit interferences which reveal the trace structures below the level of manifestation of form and pattern. In those trace structures the intaglio of form and pattern as interference patterns between disorder and order appear. It is this trace level that give us the foundation for the understanding of the autopoietic systems that imitate living/cognitive organisms. The living and the cognitive also produce a fusion of process and structure that has a qualitative difference from process or structure in isolation. The autopoietic theory of Maturana and Varela display these features of reversibility very well. However, these

theories break down when we move to consider the social. Thus the social must be a new level of organization that goes beyond the autopoietic. Autopoietic system maintain their organization homeostatically. A reflexive system is defined as the next level beyond the autopoietic and it is seen as heterodynamic instead of homeostatic with respect to its organization. This means a reflexive system is ecstatic in projecting the world and changes its organization dynamically to different organizational regimes. Thus the reflexive system can accept emergent events as the way the worldview is projected changes radically over time. We say that such a fusion of process and structure lies right on the brink of the unthinkable because it accepts changes from the region of what is incomprehensible in relation to it and deals with these changes which are called emergent events. At this level there is a chiasm between the social and the psychic. From one point of view reflexive systems are social but from another point of view they are psychological. Thus there is a psychosocial dual-intaglio at the level of the reflexive heterodynamic system. The understanding of heterodynamic systems is the furthest reaches of all systems theory.

Each of these levels of dual intaglio that we have been laying out are extensions of General Systems Theory. They lay beyond the understanding of structural-process fusion. Structural-process and process-structural fusion exist at each level of the epistemological framework. We can view these merely as reversible process and structural modes of the framework or we can look beyond that to see the qualitative difference between the fused and the unfused aspects of structure and process. This qualitative difference points us toward the special systems that emerge from General Systems Theory. These are the systems theories regarding dissipative, autopoietic and reflexive systems. They appear as the fusion of process and structure from the timespace perspective. This fusion has a qualitative difference that expresses itself quantitatively as well[78]. We see here that Dissipative systems can be looked at from the point of view of the object, source, data, and generative systems. The Autopoietic system can be looked at from the point of view of all these systems as well as from the point of view of Knowledge and Life. The levels of learning

78. For further details see the author's two series of papers On the Social Construction of Emergent Worlds and Steps Toward the Threshold of the Social (unpublished manuscripts).

are the province of the reflexive system and can be considered up to the point of unthinkablity.

Now that we have defined the special systems and their chiasmic fusion we can go back to consider the generic fusion from the process and structural perspectives.

5.6.8.STRUCTURAL MODELED LEVEL SYSTEM

At each level there is a fusion which emphasizes structure over process and one which emphasizes process over structure. These take on a different quality from the timespace viewpoint that reveals the special systems that emerge from GST. However if we go back and look at the structural-modeled system that exists at each level from the spacetime viewpoint we see that when space dominates time we get the equivalent of a knowledge representation system as in Prolog where connections in space are more important than the processing in time. In knowledge representation schemes the knowledge is coded into structures which are unified by a single logical algorithm. There is only one process and multiple knowledge representations on which it does its work.

5.6.9.MODELED STRUCTURAL LEVEL SYSTEM

When time dominates space from a spacetime perspective we see that we get a normal relation in programming between processing and memory where the processing controls the memory rather than the configuration of memory controlling the processing. But here we have an interpreted system where data and processing are more intimately connected rather than a precompiled program which operates on completely separate data.

Knowledge representation that emphasizes space over time is independent of interpretation which emphasizes time over space but still allows fusion of data and processing. These two fusions are orthogonal to each other in every case at each level of the epistemological framework.

So at the data level there can be control data and non-control data. This means that non-control data is dominated by processes while control data dominates processes and contains in the data stream the structure that controls processes.

At the generative level we see that data can be

coded into tables which control processing or we can allow processing to contain many more control statements and we can code the functioning of the software into source code algorithm.

At the knowledge level we get the difference between Prolog which uses the unification algorithm to process static knowledge structures and Lisp which does its processing on lists where the list itself can be the program being executed. Thus list processing algorithms dominate the data representation but they are fused. Prolog expresses this fusion in the way it rewrites its knowledge representation causing the unification algorithm to give different results from pass to pass.

At the levels of learning we can either emphasize the materials being learned or the learning process itself. If we emphasize the materials being learned then the drive to learn is external and we call this teaching. If we emphasize the process of learning over the materials learned then the drive is internal and we call this self-realization which Maslow called a drive. This ramifies to all the meta-levels of learning. The drive to learn at any

meta-level can be either internal or external but whatever the driving force learning has to be reciprocal and social. When we see this learning mirrored within the individual we call that the psychological realm. The psychological and the social are mirror opposites.

The fusion of the structure and process represents yet another way in which the unthinkable enters the epistemological framework. We already noted that the framework itself extends past the generative to the knowledge level and on up the hierarchy of the meta-levels of learning to the unthinkable. Then we saw that at each epistemological level there are two wings of extension to infinity. The point of infinity for both wings of meta-level extension is the same and is identical with the unthinkable. Now we see that each wing fuses with the other wing of the epistemological framework in a way that can either be seen causally from the point of view of timespace or in terms of separation from the viewpoint of spacetime. When we interpreted fusion from the point of view of timespace we recognized the levels of chiasm related to the generative system, the knowledge and living

level and the levels of learning. These we defined as the special systems that emanate from General Systems Theory. The we turned around and saw that these fusions of the wings can be seen from spacetime viewpoint instead in terms of separation and we saw how that meant the difference between coding action into spatial configurations rather than writing algorithms and we can see how these may be expressed at every level of the epistemological hierarchy. But the reversibility between the spacetime and timespace views of fusion also points us toward the essence of manifestation because of the qualitative and quantitative differences between these two views of fusion that produce a blind spot in our view of chiasmically fused aspects of systems. We cannot understand easily the connection between timespacelike fusion and spacetimelike fusion of the two wings that tend toward and infinity of meta-levels.

Figure 8:

Wild Being	**Hyper Being**
Chiasm	Finite Unthinkability
process/structure	Beyond the meta-levels
internal	of learning

Magicians meta-system
proto-gestalt

Reversible Aspects process over structure structure over process interferences external	Infinite Complexity Separate structure and process wings of meta-levels

In effect this shows that we need to understand better the extension of General Systems Theory into the realm of the special systems. The means for doing this is Software Engineering because it is software engineering that provides the connection to computability of systems. The special systems appear when we consider the fusion of structure and process from the causal or timespace perspective. They do not appear when we consider the spacetime perspective on fusion. Instead there we get a view of the computability of the combination of structure and process. Thus the special systems are bound to computability in a mysterious fashion which is not clear as we reverse our perspective from timespace to spacetime emphasis. In effect this calls for the development of a computational meta-system orthogonal to General Systems Theory. That computational meta-system is embodied in Goertzel's Magician Systems first proposed in his work Chaotic Logic. If we see the expansion of the epistemological framework

toward the unthinkable and the spreading of the wings of meta-levels of structure and process toward infinity as the dual opposites of the timespace and spacetime views of fusion then we see that these two duals define an interface which is orthogonal to GST within which the dual to GST must exist. That dual must deal with chaotic processes in a structured way which is computable. Goertzel's Magician meta-systems is the only candidate yet found that fulfills the conditions that this dual must fulfill. And it turns out that the Magician meta-system is intimately connected to the special systems when it is expressed meta-algebraically. Magician systems also have the characteristic that they express formally all the different kinds of Being. So magician systems provide us with a model of the balance of heterodynamics and homeostasis within the realm of dynamical dissipation.

Therefore we see that General Systems Theory as the theory of gestalts or showing and hiding systems must have a dual which expresses the meta-system and meta-gestalt within which gestalts form. We posit that this dual is the Magician meta-system and that it is defined negatively by the relation of the fall into the

essence of manifestation via infinite meta-levels and finite meta-levels to the embedding of fusion between process and structure seen in terms of spacetime and timespace. This reversibility between two views of fusion and two approaches to the essence of manifestation defines possibility of the Magician meta-system negatively. It is by studying the relation of the Magician meta-system and the special systems in this context that we realize their inner connection. And that connection is made possible by computability and ultimately by software as an embodiment within the matrix of spacetime and timespace.

GST is formed completely in the realm of Pure Presence. But it attempts to deal with processes in terms of models of temporal structuring as opposed to spatial structuring. Thus as a formal-structural system it gives us a view of processes while attempting to not fall into Process Being. When we realize that systems are gestalts of showing and hiding processes we fall into Process Being and we must reinterpret GST within that context. When we extend the GST epistemological framework we see that it truncates in the unthinkable which is either finite or infinite.

We have seen that this is an expression of the essence of manifestation and that is what takes GST to the third meta-level of Hyper Being where the software essence also resides. At that level we see GST and Software Engineering as duals. But then when we look at the Epistemological Framework we see that there are nodes of fusion between process and structure. We can see these in terms of spacetime or timespace as we look at the embedding of the GST epistemological framework in the timespace/spacetime matrix. These two views show us the place of the special systems that emerge from GST with their chiasmic relations between fused components. But if we look at them from another angle we get a view of the computability of these fused structures seen externally in terms of process and structure. These two views of fusion indicate the presence of Wild Being which is the highest meta-level of Being beyond Hyper Being. Within the gap between fusion and the essence of manifestation the possibility of a dual to GST arises and we posit that this dual is a Magician meta-system that combines the inscription of traces with the computational emulation of chaotic processes. We posit that

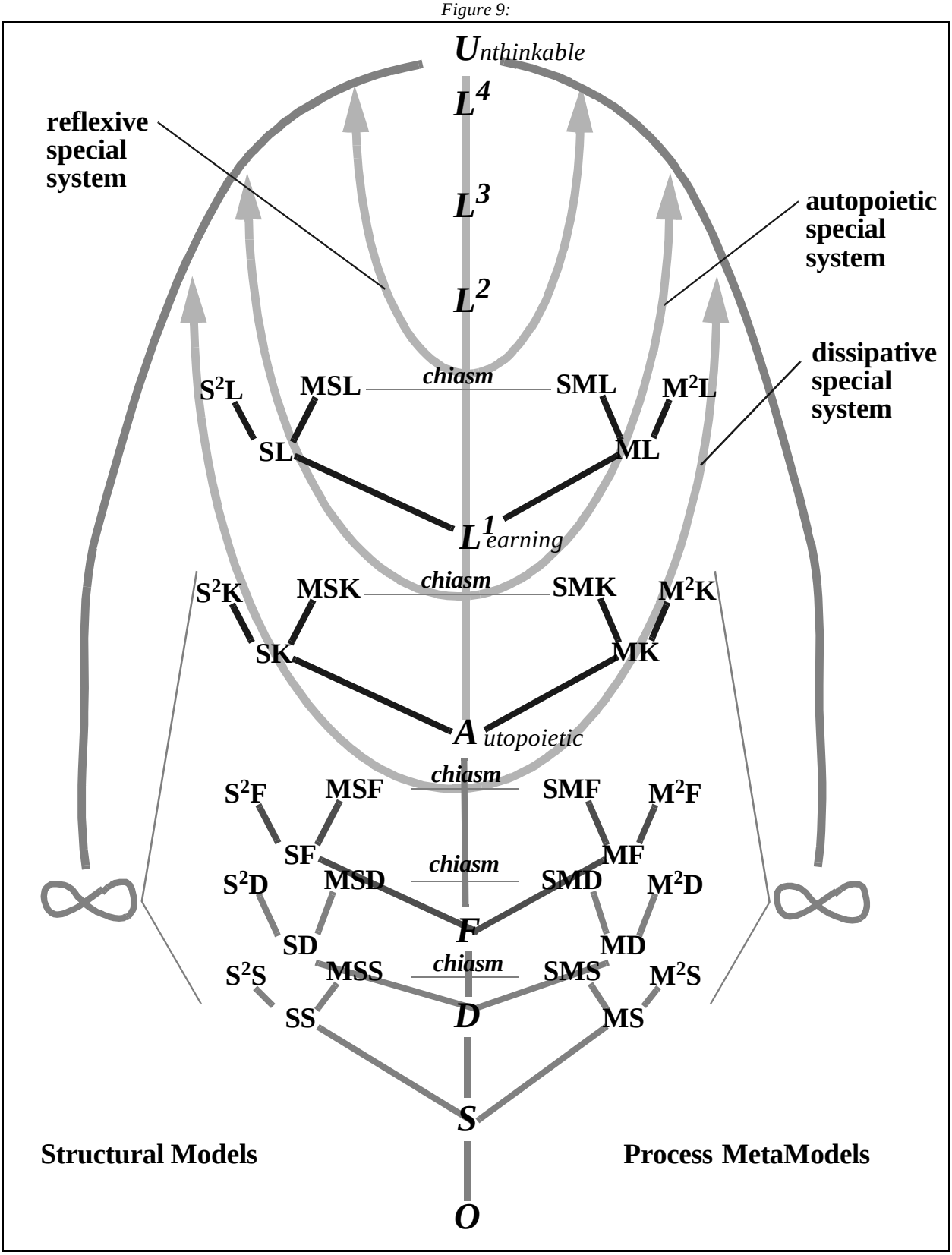
Magicians are the meta-system which combines all the special systems into a single proto-gestalt from which all the gestalts of systems arise within the clearing-in-Being. Magician systems combine all the kinds of Being into a single computable formal meta-system. The meta-system of Magicians is the dual of the structural-formal system of GST and it is software than provides the interface between them as the means of conferring computability to both.

The discovery of a dual to General Systems Theory is a surprising result when needs further study in order to explore all of its ramifications. That dual is a meta-system that defines the basis for the emanation of all the systems that arise within General Systems Theory. This dual of GST can only be appreciated from the point of view articulated by the definition of the different kinds of Being. But once this perspective on systems that looks explicitly at their ontological basis has been established it becomes clear that GST needs underpinnings that attach it to all the more fundamental ontological levels. Magician meta-systems perform that role. They unify all special systems theories and provide a meta system

that defines their ontological basis. It is clear that a major extension to the foundations of General Systems Theory has been proposed based on these ontological ramifications of the fragmentation of Being which relates the most general system to a computational infrastructure and also to the thresholds of complexity that provide the basis for the emanation of dissipative, autopoietic and reflexive systems.

June 10, 1996

Figure 9:


Unthinkable
L4
L3
L2
reflexive special system
autopoietic special system
dissipative special system
S2L
MSL
chiasm
SML
M2L
SL
ML
L1earning
S2K
MSK
chiasm
SMK
M2K
SK
MK
Autopoietic
chiasm
S2F
MSF
SMF
M2F
SF
MF
chiasm
S2D
MSD
SMD
M2D
F
SD
MD
chiasm
S2S
MSS
SMS
M2S
SS
D
MS
S
O
Structural Models
Process MetaModels

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

Note: A later version of this chapter was published in The International Journal of General Systems, Vol 24(1-2), pp 43-94. It is copyrighted by Overseas Publishers Association (OPA) Amsterdam B.V. of The Netherlands under license by Gordon and Breach Science Publishers SA

c:\wildsoft.src\tastle_2.doc

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-xxxx
ISBN 0-xxx-xxxxx-x

Keywords:

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