The Essential Nature of Product Traceability

and its relation to Agile Approaches

Kent Palmer
kent@palmer.name
714-633-9508
http://kdp.me
http://orcid.org/0000-0002-5298-4422

Draft 03 20121031 corrected 20130227 PTS01a09.doc
Corrected per CSER 2014 Conference Paper 2014.01.30 PTS01a17.doc
Edited and corrected 2014.03.11 PTS01a19.doc

Abstract. This is a discussion of the essential features of product development traceability maps in relation to requirements, architecture, functional models, components, and tests as a set of order type hierarchies and their cross-links. This paper lays out the structure of these ideal traceability relationships that define the essence of the product under development. The importance of the trace relationships to the product is clarified and then the abandonment of traceability in the Agile approach is discussed. Following that, a way to transform between synthetic canonical narrative (story) representations that appear in the product backlog and the traditionally separate hierarchical form of the trace structure of the product will be examined. The fact that it is possible to transform back and forth between the canonic narrative and traditional hierarchical representations of trace structures, and the fact that trace structures can be produced in a 'just in time' fashion that evolves during product development demonstrate that these trace structures can be used in both an Agile and Lean fashion within the development process. Also, we can show that when the trace structure is produced outside the narrative representation it can have the additional benefit of helping to determine the precedent order of development so that rework can be avoided. The lack of the extrinsic external trace structure of the product that gives access to its intelligibility is, in fact, a form of technical debt. Thus, traditional trace structures using this model can be seen as an essential tool for product owners to produce sound and coherent development narratives and for structuring and prioritizing the backlog in the Agile and Lean approaches to software and systems development.

The Essence of Traceability

What Is Traceability?

Introduction. This essay lays out a theory for traceability structures and how they may be transformed by Agile Approaches because many Agile approaches have jettisoned traditional traceability in the name of increased effectiveness. It is essential to understand that a form of Lean traceability is still needed in Agile development contexts in order to preserve the intelligibility of products and that it is possible to transform back and forth between the Agile context and the traditional traceability structures so that traceability loss does not impede agility. To forget traceability under the name of agility is a dangerous trend and this article prepares for the reassertion of the need for traceability, not just as something nice to have, or something that you must do because others demand it, but as an essential feature of Agile products that will increase our ability to realize the benefits of agility in Systems development.

What is Traceability? Normally we speak of Traceability in the context of Requirements and of the ability to trace to tests that verify requirements. However, traceability is really the product's highest level and most lasting structure and this structure is mathematically necessitated. So, this is really the essence of the product and it lasts as long as the product lasts. Traceability is the access we create for ourselves to the lasting essence of our product. We will speak of the essence of the product perduring, which means that it lasts throughout the life of the product. That essence exists whether we have access to it or not. The best thing is to build the access to the essence of the product as we are building the product itself. If we do not do this then we are blind to the essence of the product. So, traceability is the means of making the 'order instilling process' visible and that software process produces the 'strata of order' in the product during development.

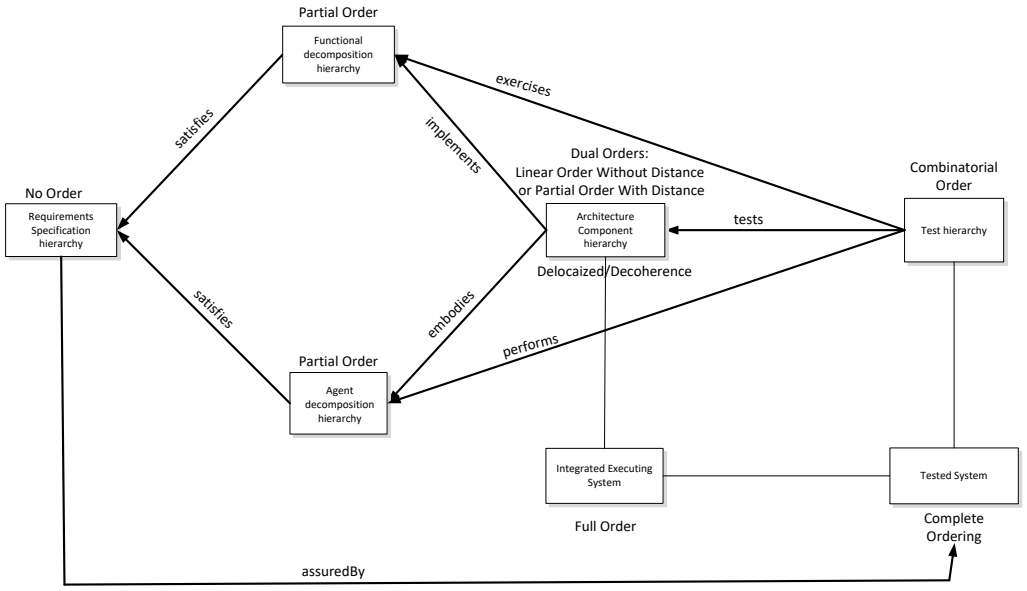


Figure 1

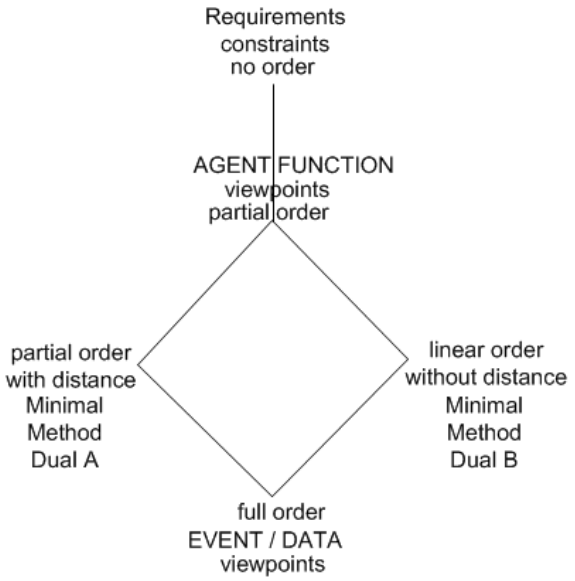
Trace Structure. We normally think of the trace structure as something extra added to the product during the development process. But, in fact, the trace structure only gives us access to

the essential structure of the product that is built into the product as we are doing development. This is necessitated by the mathematics underlying the development that we normally do not think about, but is, in fact, determining product development all the time.

Basically, the structure that we are talking about is one that links Requirements, Agents, Functions, Components, Code packages, and Tests together. But, each of these types of structure (to which we trace) has its own order. For instance, Requirements are unordered, while Agents and Functions are partially ordered, and Components are linearly ordered without distance, or partially ordered with distance. Executing Packages of source code in modules are fully ordered and Tests are combinatorially ordered. We could also add a heterarchy of Contexts that are probabilistic and randomly ordered. George Klir names some of these types of orders "Methodological Distinctions" in Architecture of Systems Problem Solving[1]. The basic message that they bring with them is that in the system production process, especially in relation to software, we are introducing order to the product step by step and that process is constrained by the lattice of the kinds of order that exist in mathematics that can be applied to our system and introduced into it. Essentially the structure of the development process is driven by particular orders that are necessary for us to introduce into the product. And when the product is done at a given release, it contains these kinds of order within it in the various strata of the product. Having access to these different order strata of the product is essential for the intelligibility of the product. So, for example, Requirements as axiom-like statements of constraint have no order, and we have access to those strata of the product when we have the Requirements listed and related to other parts of the product. On the other hand, Agents and Functions are partially ordered. Agents are the separate lines of computation within the Architecture. Functions, on the other hand, are the various features and capabilities of the System. Functions are diffused within the product and allocated to the Agents that have the capabilities to perform the intentional actions that are assigned to the system. It is possible for the Agents and Functions to be done in different orders, and this is what gives the system its flexibility. Components contain the two types of order that are parallel within the lattice of possible orders. This is linear order without distance and partial order with distance. These reciprocal orders show up as the minimal methods[2] that we use to visualize the variety of static design elements within the system such as in UML and SysML. Packages of Source Code arranged in modules are fully ordered as they are executed, which produces a signature in spacetime. Test cases, on the other hand, are combinatorially ordered, a kind of order not discussed by George Klir but essential to making sure that the system will work in all the combinatorially possible states that it may be executed in with regard to its context of use. It is in the testing that we assure the quality of the system and it is also in testing that we validate that the system will work when executed in its intended environment. In testing we present parameter inputs in probabilistic random sets in order to simulate the variability of the contexts in which the system must operate. The system must be able to operate in various contexts.

[1] Klir, George J. Architecture of Systems Problem Solving. New York: Plenum Press, 1985.
[2] Palmer, Kent, "Software Engineering Design Methods and General Systems Theory". International Journal of General Systems [Vol. 24 (1-2) 1996 pp.43-94].

Context is yet another heterarchy that we could separate out, but instead we will include it under the rubric of testing.

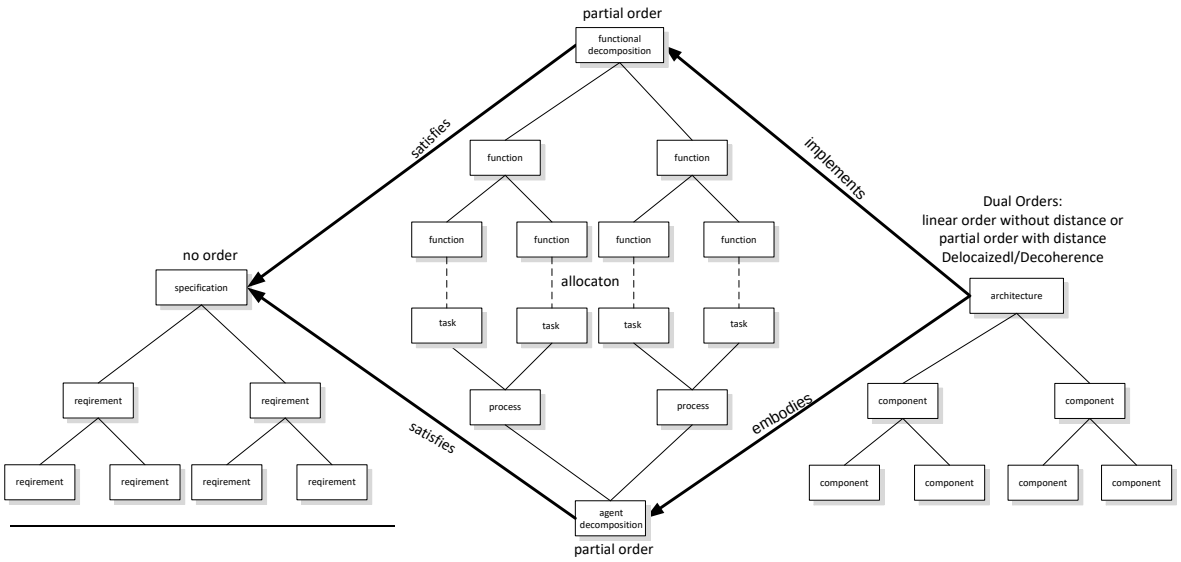


G. Klir's Methodological Distinctions and the relationships between the Viewpoints and Minimal Method Duals

Figure 2

These kinds of order need to be introduced in layers so that the development process may occur properly. This will lead to a working product that we can understand. The V of the Lifecycle is driven by these kinds of order and it is necessary to instill them sequentially in the product. So, for example, it is traditional that we start with requirements, then develop architectural designs that embody functions, and then decompose the product into levels that are then designed at each level until we get to the Code that we write, which is the embodiment of the identified components at the lowest level as they nest up into the higher level components of the architectural design. Once the source code is written for the implementation of the designed system, then we can start executing that code against module test cases, and then even higher level test cases until the entire system is tested. At the back end of the V lifecycle we are integrating and testing various levels of the system and this leads to verification and validation of the system. Verification makes sure that the system (as tested) meets the requirements. Validation makes sure that the system actually works in the environment for the intended purpose since the *test* environment should be as close as possible to the actual *intended* environment of use. The V lifecycle (or any other lifecycle) is constrained by the existence of the lattice of orders called the "Methodological Distinctions" by George Klir. Software production is the instilling of these mathematical orders in an orderly fashion. The 'ordering of order' is called organization. We must organize ourselves and while doing that we instill order into the product in an orderly way. When we organize ourselves and execute our plans to create order in the product it naturally occurs that the best way to do this is to lay down the 'strata of

order' in the product in a way that accords with the lattice of Methodological Distinctions. So, the 'strata of types of order' determine not only the structure of the product, but also the structure of the traditional process by which we produce the product. This is why products have structure, and why the processes by which those products are brought into existence have the structures that they do. By necessity they are supported by different kinds of work because it is written into the infra-structure of mathematics that these are the possible orders and that they are ordered by the lattice of Methodological Distinctions, and take note that we cannot change mathematics, we can only discover it and use it to our advantage. Software is an artifact that is extremely loose in its externally constrained structuring and it is primarily determined by mathematics in its essential nature while exhibiting the qualities of Hyper Being[3]. Any order of our development process that goes against the ordering of the Methodological Distinctions lattice is a kind of disorder and is not as effective nor as efficient as following the 'order of ordering' laid down in mathematics. Thus, we want to be Agile (effective) and Lean (efficient) or, what together might be called *efficacious*. When we say that we 'self-organize' as a team to produce the product, we are organizing around an intrinsic order that is predetermined by mathematics, we could apply this to everything we create, but it is particularly applicable to software because software has few other constraints beyond the mathematical constructs, the Turing machine[4], and the hardware architecture on which it is executed. Yet, just because the product is forced to take on its order in this fashion (one layer of order at a time), this does not mean we have access to those strata of ordering within the product that will become effective at different stages of production. To have access to this essential structure of the product we must build the software in tandem with its traceability structures.



[3] Hyper Being is what J. Derrida calls *Differance* which is differing and deferring which we talk about under the rubric of delocalization and decoherence of software code that appears in the realization of the design. See Derrida, Jacques. Of Grammatology. Baltimore: Johns Hopkins University Press, 1976. See also 'Software Ontology' essay which is part of Wild Software Meta-systems at http://works.bepress.com/kent_palmer . See Emergent Design dissertation of the author for more detail on how the essence is software is based on Hyper Being at http://emergentdesign.net

[4] See 'Hacking the Essence of Software' by the author at http://kentpalmer.name

Figure 3

Traceability structures involve a series of hierarchies that reflect the different types of order elements within the system under development. For example, we have a series of hierarchies for requirements, functions, agents, components, modules, and tests. Each of these hierarchies has its own anchors and exists as having nodes of the given type that are linked to each other by parent/child relationships all the way down to the leaf nodes. All the nodes in a given hierarchy are of a designated type related to certain kind of order. There may also be other performance, context, and interface hierarchies, but these are optional. We establish these hierarchies as they become necessary in the development process when the type of order that they define has been created within the product. The key point is that it is the cross-links at the leaf node level between these hierarchies that form the traceability structure of the product. The hierarchies 'themselves' merely represent the elements of a given type of ordering. The product's unique signature is defined by the set of nodes in those hierarchies as well as the way that they are cross-linked together. The cross-links are different sorts of 'equivalence relations' (such as assures and satisfies) between elements of different types (such as function and agent) that are determined by the different orders (such as partial and full order) that ideally exist in their own hierarchies.

To illustrate, when we are given a requirements hierarchy and the functional hierarchy is known, we are able to crosslink them so that we can say that a given function **satisfies** a requirement. But, if it is a non-functional requirement, then it would instead be linked to a performance hierarchy, such as the agent hierarchy, so that a given architecture of the functions would **satisfy** a non-functional requirement. Agents and Functions are both partially ordered and together they may model a Domain. Functions are allocated to agents within the performance hierarchy and it is the entanglement of these two partial orders that gives us the architecture expressed by the components of the system under development. The fact that both agents and functions are partially ordered is taken advantage of by Agile approaches that say that these can be put into a backlog and done according to the priorities that give the most value to the customer first. *Yet, the fact that this ordering of the agents and functions is flexible does not mean they can be done in any order. Rather, there is a set of constraints on that developmental order, which, if violated, will cause excessive rework because some things must be done before others as dictated by the architectural dependencies as well as the exigencies of the technologies being used.* But, once we know what functions and what architectural structures satisfy a requirement, then we can go on to define the components of the hierarchy. Components contain elements that are either partially ordered with distance, or linearly ordered without distance. This difference shows up in the complementarity of the minimal methods[5][2] that we employ when the product is being designed. This is to say that when we write the code the design components become delocalized and decoherent within the source code, which

[5] Palmer, Kent, "Software Engineering Design Methods and General Systems Theory". International Journal of General Systems [Vol. 24 (1-2) 1996 pp.43-94].

means that within the source code there is always some ‘spreading out’ of the elements that are necessary for realizing a given component. This is made easier to deal with through object oriented designs, although these delocalization and decoherent properties of the system merely shift to being the order of the method calls between objects and the aspects related to cross-cutting concerns. When we have defined those components then we can say that they **embody** certain agency nodes in the architectural hierarchy made necessary for performance reasons and that they **implement** certain functions in the functional hierarchy. Once the components have been defined, then it is possible to go on to write the source code that **realizes** them in ‘Turing complete’ programming language as software configuration items, which are functions within those agents comprised of those components. Realization combines both embodiment and implementation into a single source code base that has linear ordering within the source files, but, in terms of execution, this could take on an extremely complex structure. Thus, in order to make sure that the realized system works, we need to create a test plan with multiple levels of test cases. When these test cases exist at each level they **exercise** the functions, **perform** the agent’s duties, and **test** the components. Finally, when we complete the circle and connect the test cases back to the requirements, we say that the tests **assure** the requirements. Fully closed circuits among the hierarchies of types of order will ensure that the structure of the product is sound.

This structure of cross-links between order type hierarchical nodes is the essential structure of the product that remains in place when development is finished. It is the structure that gives intelligibility to the macro-scale product throughout its lifecycle. It is absolutely crucial to have access to this structure throughout the lifecycle of the product. But since this is an extrinsic structure from the point of view of realizing the source code of the system, these traces are easily lost, so, it is important to note that we lose access to them because we do not maintain and focus on the traces. If we jettison traceability all together as has become popular in Agile development, then we do not have a reliable trace structure to fall back on to rebuild the traces necessary to make the product intelligible when things have gone wrong in development, or if the development staff has changed. In that case, we have to reengineer the system from the code base and build up that intelligibility again from scratch, which is not very effective or efficient. The loss of the trace structure in development may be seen as a particularly insidious form of technical debt as it is a loss of the external intelligibility of the software product. So, the trace structure is an attribute of forethought, we build it along with the product because we want the product to remain intelligible for more than the few people who know the code base as well but rather for everyone who has to deal with the product. The trace structure is like the table of contents, or better, the outline of a book, or the index. It gives access to the relevant and significant parts of the book’s content on demand. Without the table of contents and the index, if you need to find something in the book, then you are forced to read the book again, or skim it, perhaps missing the things you are looking for! Due to its opacity, reading source code is even more onerous than reading complex technical books for which we invariably produce contents with outlines and indexes. The contents with outlines and the indexes are extrinsic to the text and we consider it an essential part of the book product if the subject matter is

technical and complex.

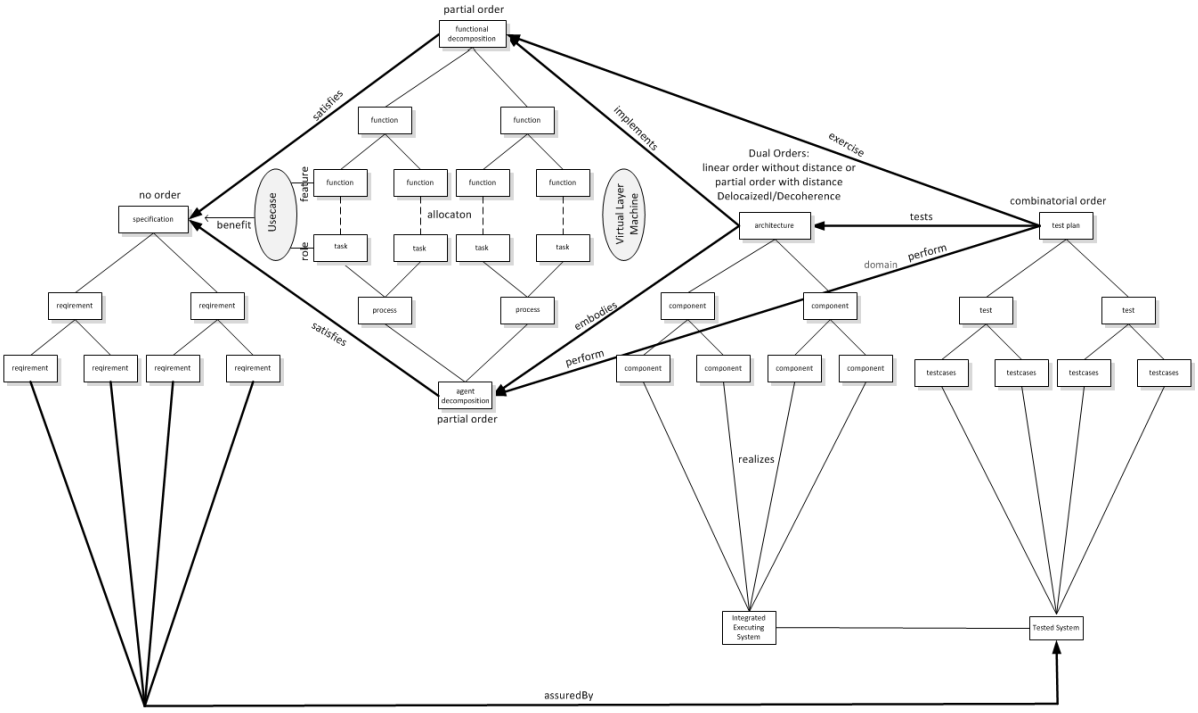


Figure 4 Usecase and Virtual Layered Machines within Hierarchy Structure

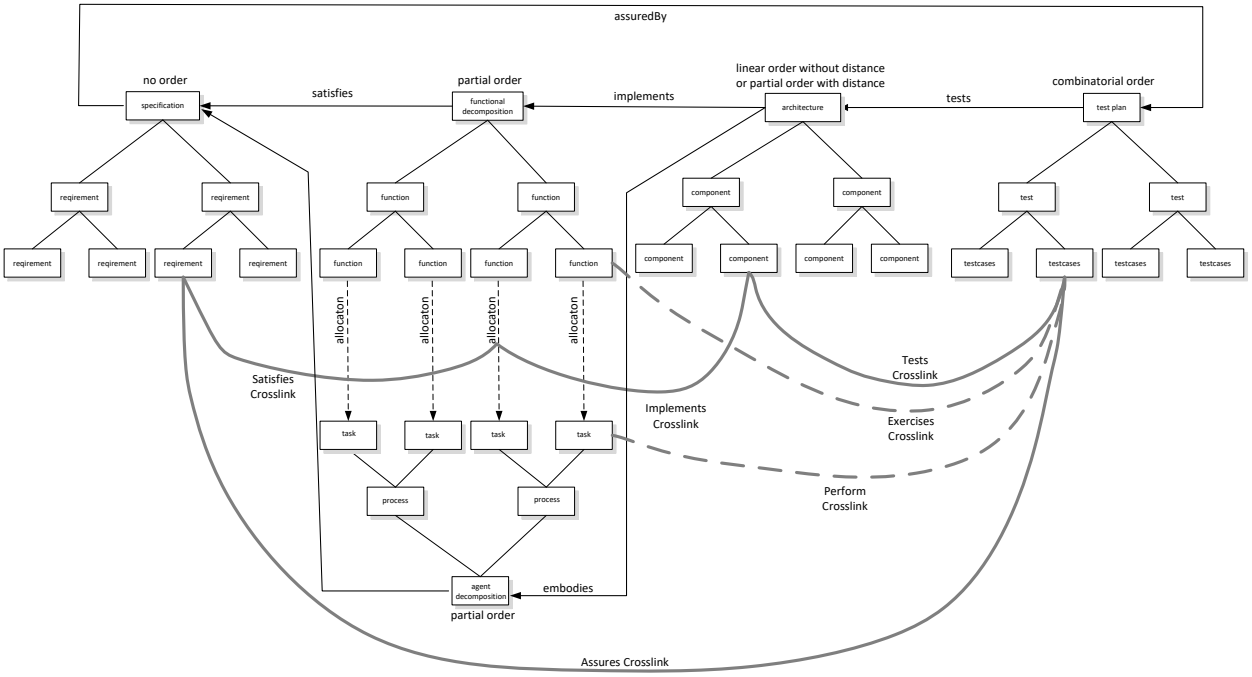


Figure 5 Closure of a single circuit gives soundness if tests passed and closure of all circuits gives coherence to product.

Kinds of Order and Minimal Method Representations. Another point of interest is that the types of order have an peculiar structure. There is a non-ordered type and two partially ordered types (agent and function). There is also one type (components) that has two orders (linear order without distance, and partial order with distance), and then again there are two types of full order (data and event) and finally one combinatorially ordered type (test). This is an interesting structure that is imposed upon us by the lattice of the orders in mathematics as articulated by the background variables by which we measure the flow with the system that articulates the fundamental viewpoints on the System, which are function, agent, data, and event. We can see that it also determines the minimal methods that we use to represent the various elements and the relationships between elements of different orders. Thus, for example, between the Agent and Function viewpoints on the design there are two minimal methods, which are the Virtual Layered Machines and Usecases. Between Agent and Function there is a single method called 'sequence diagrams' that represents worldlines and scenarios in a relativistic spacetime where there is no global clock. Between Function and Data there is the dataflow diagram or the object presented as different ways of representing the entities in the system. Between Agent and Data there is the DARTS[6] methodology for representing concurrency and parallel computation via tasks with ports (actors) and semaphores. Between Event and Data there is the State Machine and Petri Net minimal methods that give the basic computational structure to the system, which is treated in detail in my paper "Hacking The Essence of Software"[7]. Between Event and Data there is a relativistic spacetime (memory-cycles) interval. The test software is the inverse of the system. It is a testing environment or meta-system that can be represented exactly in the same way that the software subjected to testing is represented. Then, in the combinatoric order of testing, there is an inverse mirroring of the entire system represented as a meta-system with its own separate source code base, which is a scaffolding for the testing of the system. Thus, the Methodological Distinctions drives not only the structure of the product and the process of developing the product, but also our minimal

[6] Design Approach for Real-Time Systems See Gomaa, Hassan. Designing Concurrent, Distributed, and Real-Time Applications with Uml. Reading, MA: Addison-Wesley, 2000. Gomaa, Hassan. Software Design Methods for Concurrent and Real-Time Systems. Reading, Mass: Addison-Wesley, 1993.

[7] See http://kentpalmer.name

representations of it that are slices of its Turing machine or the universal Turing machine that forms the operating system in which the system under design operates.

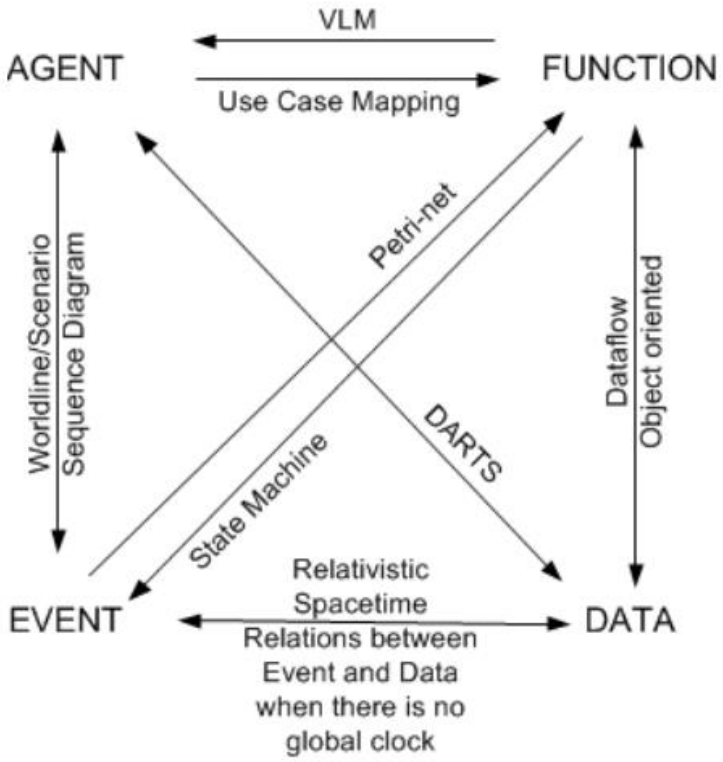


Figure 6

This entire structure is represented in Figure 9 where we see the elements in the lattice of methodological distinctions, the minimal methods that connect them, and the trace relations between them as crosslinks. This is the most basic architectural representation of the essence of software systems and it is also driven by the mathematics of the kinds of order (as well as the differentiation of the fundamental background variables necessary to see change within the system) that we are seeking to instill in our product. Order is, itself, ordered by the lattice of methodological distinctions. When e think about the Architecture of the System we use these minimal methods to conceive of how different slices of the Turing machine will work in its parts and together as a whole. We must organize ourselves to produce artifacts with these orders that are layered in the strata of products we build in an orderly way self-organization of the team is organized around this fundamental order of the System itself as we

conceive of it working together. It is significant that this orderly way of conceiving the product we build connects hierarchies of requirements, with functions and agents and then with components that are realized in source code modules that are then integrated and tested by other modules of test code that exercise in execution the combinatorial order of the possible ways that the software can be executed to the greatest extent possible to assure that the requirements are met, and that the software works as advertised.

Set and Mass Representations of the Product. Something not often noted is that there is a transition from 'set like' designs to 'mass like' executables when the software is compiled and built. When the software becomes an executable it is very difficult to see inside it, even with debuggers to know exactly what is happening within the mass of the software executable. In our culture we emphasize sets over masses, and really only have two kinds of mathematics that are mass-like. These are geometry and topology as well as the real number line that is folded back into itself by the rules of algebras. The rest of mathematics is 'set like' and we are used to using syllogistic logic to think about those sets that inform our designs. However, masses have their own logic called Pervasion logic. The best example of which is the Boundary logic developed by G. Spencer Brown[8] or Bricken[9], or Hellerstein[10]. Thus, there is a way to think logically about executables through pervasion logic so that we do not have to be lost when we move across the line from 'design sets' to 'mass executables' in the manner that we are today. Rather, we can apply Boundary Logic to understand what is going on in the mixtures of masses that are created within our executables. We can use logics that work like Venn diagrams defining the emergent properties that pervade the masses of the executables. These mass pervasions are represented as functions in the mathematical orders. Functions ultimately express our intention for the system to do something. In building a system we are projecting our intention into it and that is expressed by the pervasion of the system by functions. Functions, as Robert Rosen said in Life Itself, were first discovered by taking animals and crippling them in some way and seeing what capabilities and activities were lost or changed. In this way functions are *localized* to the *parts* of the

[8] Spencer-Brown, G. Laws of Form. London: Allen & Unwin, 1969.
[9] http://www.boundarymath.org/ See also http://www.boundaryinstitute.org/bi/
[10] Hellerstein, N S. Diamond: A Paradox Logic. New Jersey: World Scientific, 2010. Hellerstein, N S. Delta, a Paradox Logic. Singapore: World Scientific, 1997

system. We reverse this process when we create a functional model and then assign functions to components or agents within the system because we made the functions pervade certain parts that implement them. This is important for traceability because there is a chasm represented by the full ordering of event and data in spacetime during execution that is relativistic if there is no global clock for the system. When we enter that arena where we live within the system (in the context of integration and testing) and are exercising it and allowing the agents within it to perform their duties, then we lose track of the trace structure and we only pick it up again as we map from the tests to the functions, or to the components, agents, and functions. In other words, we lose track of the intelligibility of the system when it transforms into a mass that is executing and when we do view it, it is usually through very narrow windows given to us by the debugger. The trace structure provides a framework for a bridge over this mass-like chasm of the executing code because it maintains the intelligibility around that unintelligible moment of the execution of the software en-masse. One of the main differences between users and developers is that the users just see the executing software en-masse, while the developers can peer into it with debuggers attempting to ferret out defects in the executing code not envisaged during development. The discipline and the challenge of building a source code for a system is to make sure that each action of the system only appears once in the code giving it a 'set like' structure. But, we may fail to do that, and during execution there may be many agents executing the same actions within the code in different orders, with different lag times, etc., so that it is very difficult to catch the mechanic errors within the software machine embedded in the code as defects. It is this transformation between set and mass states of the product that demands that a framework that preserves the intelligibility of the software be built around the product. And that framework is the trace structure only one part of which is related to requirements and their relations to tests, but rather this trace structure is related to each type of mathematical entity of the system that appear in all the developmental strata of the system. The trace structure ties the system together, and the minimal methods give us an abstract representation for each element of the system that we can use for applying reason to the processes of the system. For example, associated with data is the Entity-Relationship-Attribute diagrams and related to the Events there is temporal logic. The heart of the system is in its representation of state machines with stacks or tapes

and the protocols between them that can be modeled with petri nets. DARTS[11] gives us a way to represent the parallel and concurrent actors meant to execute in unison on different tasks performing different types of functions within the system, and we can follow the interchange between these agents through the sequence diagram that gives insight into the interaction between different worldlines under varying scenarios. At the top level between agents and functions there are two minimal methods, one is Usecase and the other is the Virtual Layered Machine that can be seen as an abstraction that encompasses the entire system, which can be represented by the Gurevich Abstract State Machine (ASM) structure[12]. We can use the Wisse Metapattern method[13] to understand the objects within the system that are interacting causally through the rules of the Gurevich ASM. All this takes place under the umbrella of the unordered requirements constraints of the system that reflect user needs. All the processes that the system tests respond to are within this umbrella of requirements constraints and show that they are assured by the tests and satisfied by the agents and functions that are distributed into components within the system. Components **implement** functions and **embody** the agents. Code is written that **realizes** the components, then it is compiled into an actual mass of executables and the system is built and executed against its test cases, hopefully mostly automated. The tests **exercise** the functions, **perform** the duties of the agent, and **test** the components. When the tests are passed, then they **assure** that the requirements are met for the system.

[11] Cf. H. Gomaa Design and Analysis of Real-Time Systems

[12] Gurevich, Yuri. Abstract State Machines: Theory and Applications : International Workshop, Asm 2000, Monte Verità, Switzerland, March 2000 : Proceedings. Berlin: Springer, 2000

[13] Wisse, Pieter. Metapattern: Context and Time in Information Models. Boston: Addison-Wesley, 2001. See also http://www.informationdynamics.nl/knitbits/htm/primer.htm http://www.informationdynamics.nl/pwisse/

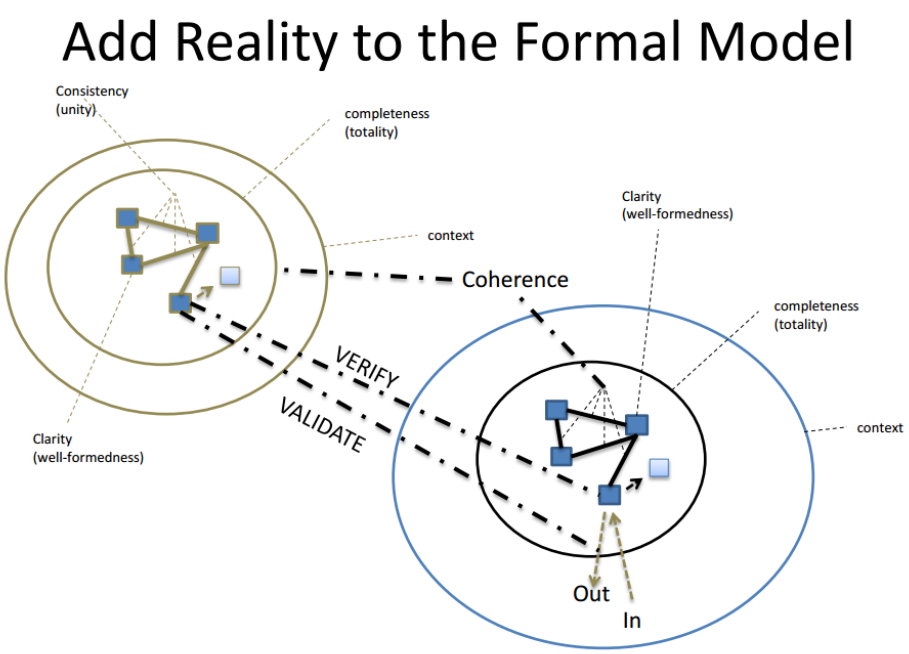


Figure 7

Traceability and Coherence.

This cycle of the cross-links of the trace structure from requirements through agent and function, through components to test, and back to requirements is what gives the system its coherence. The design is a formal system that is based on the aspects of Being: truth, presence, and identity, with properties between the aspects of clarity (wellformedness), completeness, and consistency. But, when the formal system is confronted by the aspect of reality during testing, then three other properties arise. These are verifiability, validity, and coherence. Coherence is the relationship between reality and identity. We run tests to not only verify and validate the software, but to check if the software is properly integrated. The limit of the integration of software is its coherence and we are proving through our testing that the software is coherent, not just correct, which is added by the architecture that was pictured, planned, and modeled during implementation. The link back from Test results to Requirements is through verification, and the link to the environment (context where the system will be used) is through validation, and the link 'to itself through itself' is via the property of coherence. Note that it is these completed circular paths of the cross-links that assure coherence of the product. Any given circuit that is closed makes the system more sound, but the full closure of all the circuits at the leaf node level makes the system tend toward the limit of full system coherence. So, the superstructure of traceability that 'seems' extrinsic is actually the basis for the most intrinsic property of the system, which is its internal coherence. If we want our products to be coherent, then we must make sure that they maintain their integrity by determining that the cross-links

between the order type hierarchies are circular and, when closed and assured, become sound. When all circuits at the leaf node level are closed, then the trace is complete and this indicates coherence of the system. Each hierarchy of order types has an anchor node that represents its unity and leaf nodes that represent its totality. When all the order types are both unified and totalized, and cross-links are established between them at the leaf nodes and completed, then the circularity between the order types assures coherence. Coherence of the product within itself is an essential characteristic. Building in that coherence is enhanced greatly by having a complete and consistent, as well as clear, trace structure. Abandoning the trace structure not only dramatically decreases the intelligibility of the system, but it also makes coherence of the end product much harder to achieve because the circular traces are not sound. The traceability structure is an external representation of traces that exist as discontinuities within the product itself. But within the product these traces are invisible for the most part because they are differences between things of different order types that do not show up explicitly within the representations of the code itself, but in the discontinuous differences that are hidden within the code and must be read between the lines of the code.

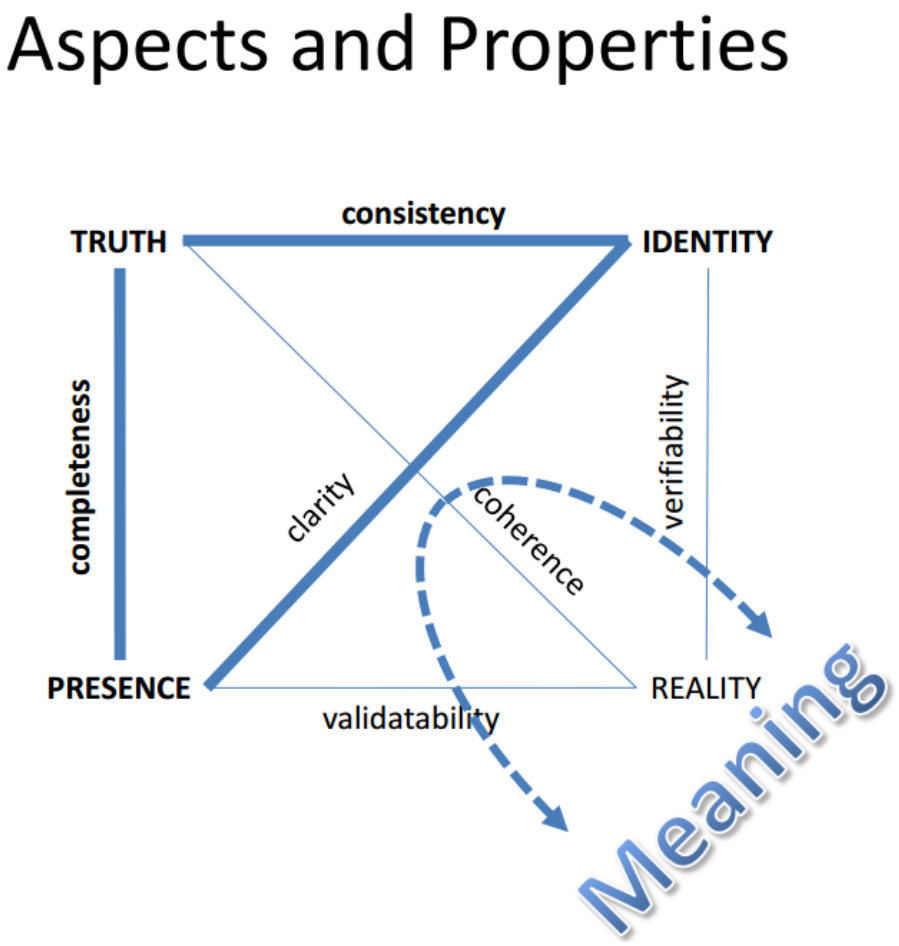


Figure 8 Formal System in relation to Reality

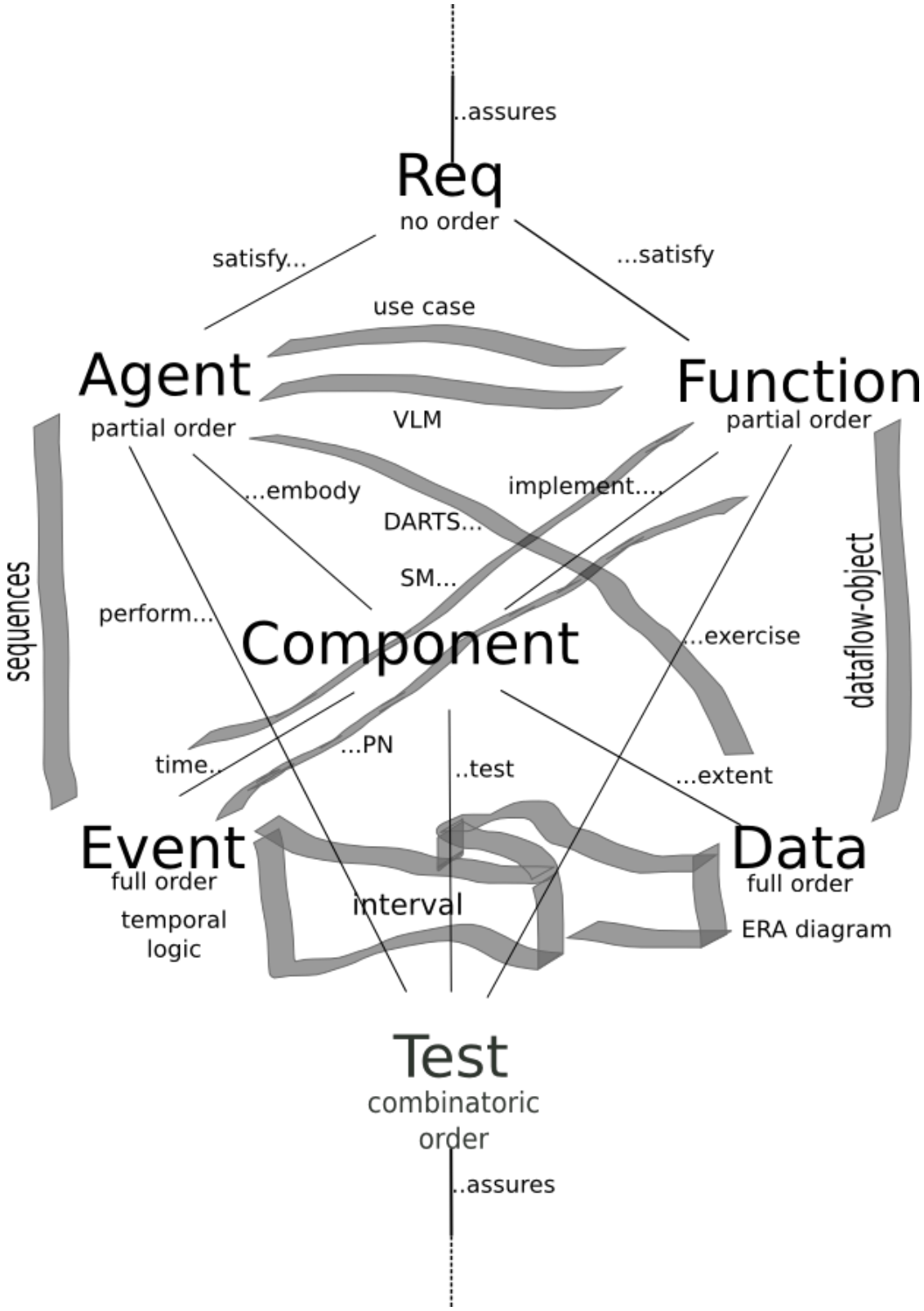


Figure 9. The Essence of the Software Product with Traceability relations and Minimal Methods

Mathematical Foundations of Traceability

Groupoids. There is a mathematical basis for traceability[14]. The fundamental basis of traceability is rooted in Groupoids[15]. Groupoids[16] (**GRPD**) are a generalization of the Category Group[17] (**GRP**). Group is a single Categorical Object with morphisms that send that object to it-self which represents basic symmetry relations of Groups. A Groupoid is like a group with multiple objects with attendant arrow morphisms rather than just one. In a Groupoid states need to have a product. A groupoid is basically a Category where all the morphism arrows are invertible[18]. A groupoid is like a path in which at each station on the path different operations are available[19]. Where a group has only one spot and there is no path, and all operations are available at that spot. Thus you can see that groupoids are more complex that groups in as much as different operations are available at different spots within a groupoid and thus it is a way to signify partial operations within a group-like setting, or if there is no operations that involve other spots then there are separate groups that are isolated in the groupoid. So the groupoid has two extremes in which there are separate isolated groups with no interaction on the one hand or there is only one group at the other extreme. But if there are separate identity elements within the groupoid then the operations can interconnect the various elements of the groupoid or it can allow a path to be traversed from one identity element to the other. The key point is that a groupoid can reflect equivalences. Thus when there are different things that are equivalent they form a groupoid structure. In our trace paths between the various hierarchies with their leaf nodes there is a set of equivalences being created I which a requirement can be equivalent to a

[14] This came to the attention of the author through the study of the discovery of Vladimir Voevodsky of the Univalent foundations of Mathematics which has implications for Type Theory in Computer Science. See Homotopy Type Theory: Univalent Foundations of Mathematics. Princeton, N.J.: Univalent Foundations program, 2013. http://homotopytypetheory.org/

[15] http://en.wikipedia.org/wiki/Groupoid

[16] Higgins, Philip J. Categories and Groupoids. London: Queen Mary College, 1960

[17] http://en.wikipedia.org/wiki/Group_(mathematics)

[18] Baez, J C. "An Introduction to N-Categories." Lecture Notes in Computer Science. (1997):

[19] Brown, Ronald. Topology and Groupoids. Deganwy: Groupoids, 2006. Brown, Ronald, Philip J. Higgins, and Rafael Sivera. Nonabelian Algebraic Topology: Filtered Spaces, Crossed Complexes, Cubical Homotopy Groupoids. Zürich: European Mathematical Society, 2011.

set of functions, and when those functions are allocated then those requirements are equivalent to the agents that are assigned to those functions, and when the agents are functions are assigned to components then those are seen as equivalent to the requirements. And when the tests are done on those components we see those tests as being equivalent to the requirements and when we close the loop then we have considered that set of equivalences as sound. Thus the path of the trace structure that sets up the equivalences between the various hierarchies are mathematically a groupoid. A groupoid is a directed graph structure which is precisely what we have created when we map from the leaf nodes of one hierarchy to the leaf nodes of the next hierarchy until we have connected all associated leaf nodes of all the hierarchies related to a given requirement. We would like to stipulate that given any requirement this loop structure that goes from hierarchy to hierarchy should be disentanglable from the groupoids related to other requirements. This prevents dependencies for the verification of requirements. However, dependencies between other leaf nodes of the other hierarchies may be impossible to partition so as not to have circular dependencies but this should be avoided at all costs if possible.

	A	B	C	D	E	F	G	H	I	J	K	L
1		a	b	c	d	e	f	g	h	i	j	k
2	a	a	a							a	a	a
3	b	b	b		b	b						
4	c			c								
5	d		d		d			d	d			
6	e		e			e	e					
7	f					f	f					f
8	g				g			g		g		
9	h				h				h		h	
10	i	i						i		i		
11	j	j							j		j	
12	k	k					k					k

Figure 10 Groupoid Table of Equivalences for one closed path

One aspect of the groupoid structure is that there are 2-groupoids[20] that are networks rather than paths. In other words when we group various paths together they naturally lead to

[20] Noohi, Behrang. "Notes on 2-Groupoids, 2-Groups and Crossed Modules." Homology, Homotopy and Applications. 9.1 (2007): 75-106.

networks of 2-groupoids and there is a natural composition of 2-groupoids into a whole network. This composition into a network of 2-groupoids is what we mentioned gives coherence to the product. The closure of groupoid paths gives soundness while the closure of the entire network of the 2-groupods gives us coherence. Groupoids naturally compose into these higher level syntheses in an additive fashion. There are not many operations in mathematics that give synthesis but the addition of groupoid paths into a 2-groupoid network is one of those composition operations. And also under composition we can see that via category theory there may be compositions of paths that are equal to each other. So there is a homeomorphism between Category Theory, Groupoid Theory and Directed Graph Theory that allows us to look at the traceability structure of a product mathematically from at least three different points of view.

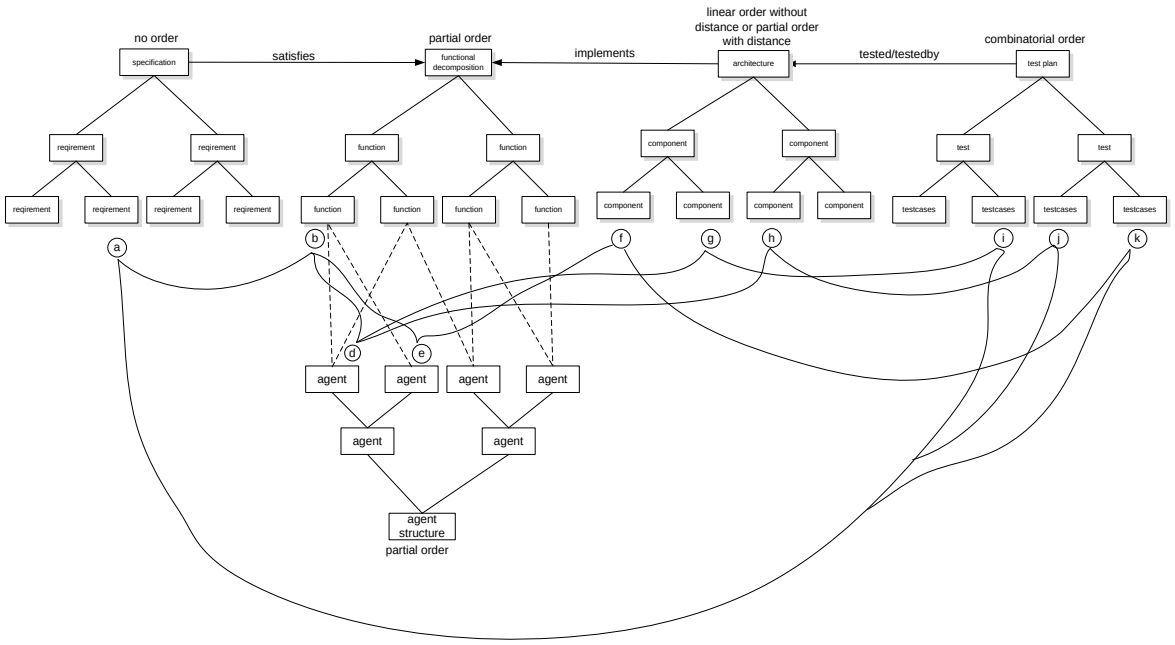


Figure 11 Groupoid Path Cross-link Trace Structure

This groupoid composition and synthesis operation can be extended to the 3-groupoid level for instance if we have product lines and we want to see how the features of different products within the product line are part of the traceability structure. We can take groupoid networks for

one product and relate them to the groupoid network of another product under the rubric of a 3-groupoid and thus see how there are interdependencies between different products in a product line in terms of the different hierarchies.

The set of Hierarches that make up the essence of Software are not arbitrary but relate to the Foundational Mathematical Categories (FMC). The FMC are the various different foundations that are possible for mathematics[21]. We have defined these as Singularity[22], Site/Event[23], Multiple[24], Set[25], Mass[26], Whole (mereology)[27], Holon[28], Holoidal[29], Singular[30]. These possible foundations for mathematics move from the central ones which are set and mass out toward the extremes. For instance wholes have parts but holons are parts that are themselves wholes, and holoidal is when all the parts interpenetrate in the whole like a hologram. Singular is when you cannot distinguish the parts from the whole any longer. Going in the other direct sets are based on the assumptions that there can be whole particulars but what is prior to the arising of the one that also defines plurality, that is what Badiou calls the Multiple. Prior to the multiple must be localization of space and time which this heterological multitude needs to exist. Prior to that is the collapse of local spacetime into a singularity where the laws of physics and the differentiations of mathematics no longer apply. It turns out that hints of these possible foundations can be found in Euclid's Elements and Datum. They come from a critique of A. Badiou's Being and Event in which he attempts to make all Ontology dependent on Set theory when there are actually multiple competing foundations for mathematics beyond set theory.

---

[21] These are defined in the author's dissertation Emergent Design at http://emergentdesign.net
[22] Collapse of laws applicable to space-time phenomena. http://en.wikipedia.org/wiki/Mathematical_singularity
[23] Localization of Spacetime understood topologically, See Badiou's idea of the Event in Being and Event
[24] From Badiou Being and Event utter heterological order prior to the identification of Oneness.
[25] Set Theory http://en.wikipedia.org/wiki/Set_(mathematics)
[26] Mass non-count dual of set seen in Geometry
[27] http://en.wikipedia.org/wiki/Mereology http://plato.stanford.edu/entries/mereology/
[28] Due to Koestler part is whole and whole is part Koestler, Arthur. Janus: A Summing Up. New York: Random House, 1978. See http://en.wikipedia.org/wiki/Holon_(philosophy)
[29] Leonard, George. The Silent Pulse: A Search for the Perfect Rhythm That Exists in Each of Us. New York: Dutton, 1978. P. 78 Neologism for things that relate to holograms, with holoid being an entity that reflects other entities in the hologram and are thus holonomic. A term for interpenetration.
[30] Singular means that there is only one like spacetime, or the world, or a particular individual

What we find interesting is how the set of hierarchies that appear in the essence of Software lattice seem to be related to these FMCs. For instance Requirements have no order and thus they are similar to the Multiple which is unordered. Function is definitely related to mass because it pervades the software product giving it unity and totality and identifying sub-functions within the product. These functions are allocated to Agents which are then like particulars of sets in as much as the agents are each separate with their own characteristics. Then the component hierarchy is related to mereology. It is the Mass envelope plus the mereology of the parts that give rise to the concept of the whole. Components are tested which is an sort of power set together with a sampling of the fields from which parameters are drawn. This is like the Holon which is where the whole is part and the part is whole. In other words through combinatoric order the integrity of the components in taken as a whole can be tested and we can see the degree of integration of the components into the whole, in which the components themselves are seen as wholes interacting within the greater whole of the system. Finally the test scenarios are drawn from the heterarchy of contexts and contexts themselves are related to each other and for the system to fit within the environment it must mirror that environment to the greatest degree possible as this is called holoidal which is a model of interpenetration. Of course all of this takes place against the background of spacetime and that is the site/event FMC. The actual compiled code is also a hierarchy of calls during execution that leaves a signature trace in spacetime. Once we realize that the hierarchies that make up the software essence are related to the FMCs then we see how these are not accidental distinctions but are themselves foundational distinctions because they are based on the possible foundations of mathematics itself. The traceability structure is spanning the discontinuities between these various possible foundations for mathematics. This suggests that the hierarchies are themselves incommensurable with each other and that the act of spanning of them to produce a sound and coherent system is showing us how are not just different types of order by different foundations

of order are knit together to for the fabric of the system. The fabric of the system as a network spans the various foundational tectonic rifts within mathematics itself. Of interest here is how Mass normally comes after set in the series but it is coming before Set in this series when set is related to Agency. We assume that this is because the emphasis in software is on achieving wholeness which is signified by the intention of functionality giving unity and totality to the software product. This functional completeness is the dual of the combinatorial testing which gives another type of wholeness which is that of all the various possibilities of interaction. Between these two bookends of wholeness come the agent hierarchy that determines performance and the component hierarchy that determines the parts of the system which will be built. The assignment of functions to agents may change or the assignment of components to agents might change in order to explore the space of possible performance based on different architectures. On either side of the two types of wholeness are the requirements and the contexts which themselves are not rigorously ordered and the whole thing takes place in the supra-context of spacetime. Thus there is a certain logic to the relation of the Foundational Mathematical Categories as they are exemplified in the differentiation of the Hierarchies that are the basis of traceability. Trace structures are crosslinks between these incommensurate realms and they produce equivalences via groupoid structures that can be composed into higher groupoid structures thus producing syntheses against the backdrop of the incommensurability of the FMCs with each other.

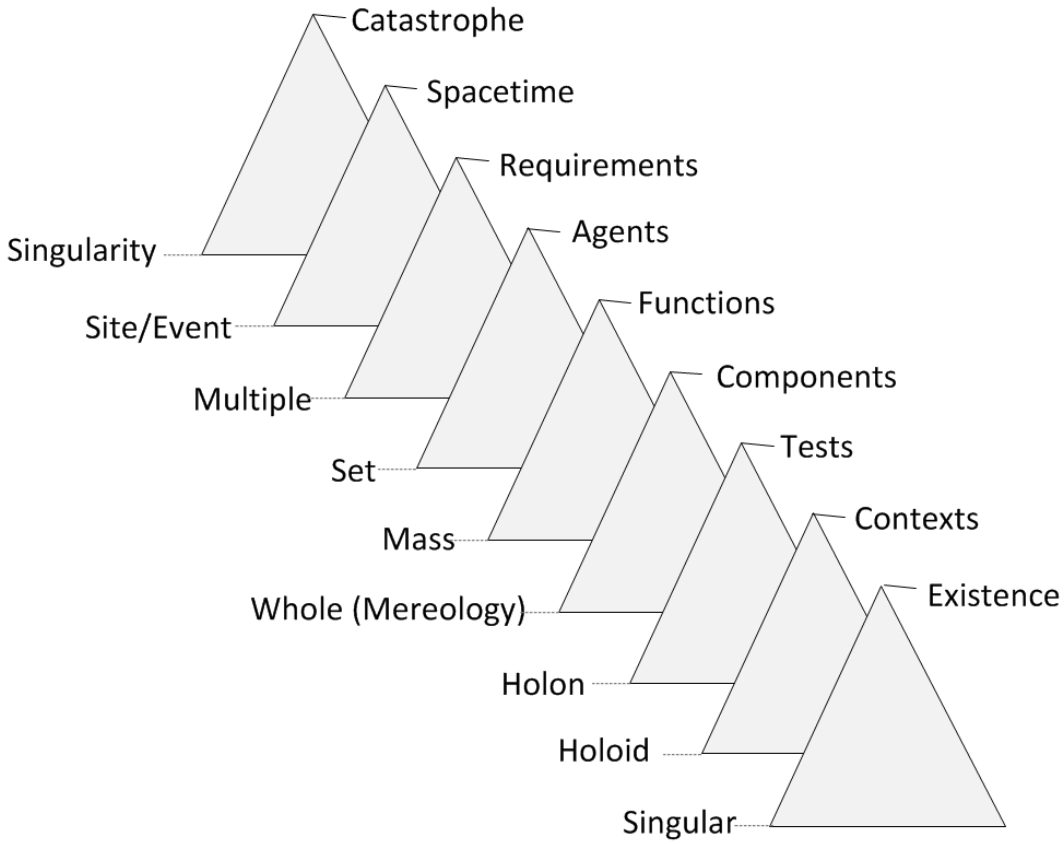


Figure 12 Foundational Mathematical Categories and Product Essence Hierarchies

It is something for us to consider why it is that the essence of software might be defined as equivalence relations across incommensurable continuities between the various FMCs. Because the structure of software is more or less unconstrained except by mathematics, the homotopy of type structures, the Turing machine formalism and the hardware infrastructure we are pushing the limits of the possibilities of order itself which comes out in the way that the types of order organize the software product directly. This suggests that we are so close to the origin of those orders that the various FMCs are visible to us as they articulate the possibilities of the interpretation of those orders. The whole product traverses those orders based on the FMCs in order to establish its own wholeness against the backdrop of multiple kinds of order and multiple kinds of FMC. To do that we need something from math which has the capability

of providing a basis for synthesis and that is the groupoids, which not only can form cycles of equivalence relations between very different realm but also can weave together to form higher level syntheses that take us from soundness to coherence within the software product. By not developing these crosslinks that give us our traceability as external access to the extrinsic essence of software we are leaving it up to chance whether we will be able to bridge these incommensurable divides between the FMCs.

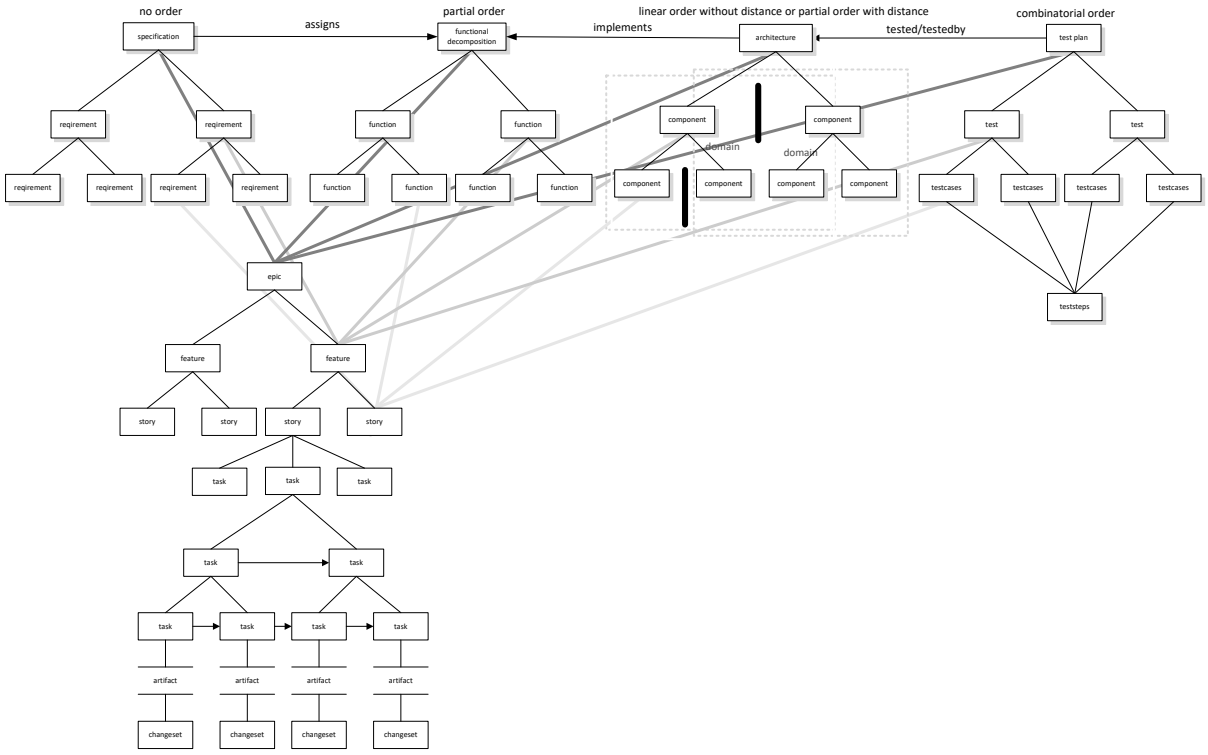


Figure 13 Relation of Narrative Hierarchy to the Product Essence Hierarchies

Another use of the groupoid structure is to relate the narrative hierarchy (epic, feature, story, task) to the groupoid paths and networks among the traceability hierarchy. It is clear that when we create epics and order them we are creating a directed graph, and when we take epics and decompose them into features and order them there is another lower level directed graph. Likewise when we take features and decompose them into stories and then order them we have created an even lower level directed graphs. So the narrative backlogs at the various levels that reflect agile product strategy are all seen in terms of various levels of abstraction of paths by

which the product is to be composed. A given epic, feature, or story is a synthesis of the different elements that would appear in the hierarchies that give a specific benefit and has a specific priority in the backlog. This narrative set of paths is related to time within the production process and the sequence of synthetic events that should happen in a certain order according to the product owners. On the other hand the traceability structure is not related to time but instead to the layers of order within the product. But order is implicit in the order layers because we expect order to be laid down in the produce such that the simplest order is instilled first before more complex orders. However, there is really no way of relating the layers of order in the product directly to time. So we can consider these two structures related to order and related to time sequence as orthogonal to each other. It is just that you would not want the time sequence order to insist on a more complex order be instilled in to the product before a simpler order if at all possible because that may well be a violation of the precedence order that is necessary to develop the product capabilities prior to features. But the paths through the narrative order can be considered as a groupoid and be represented by directed graphs. And at certain times these various paths may be parallel if different teams and different individuals are performing the different path steps in parallel. But for the most part the greater portion of the narrative paths is not being executed at any given time and so they are mostly waiting in the backlog to be executed. On the other hand once a traceability link is created in the product it is effective and continues to be effective until it is changed but normally it will last through the life of the product unless it is changed. Any given execution of development is establishing a given link within the overall traceability structure just a few at a time and according to a sequence set in the backlog. Once the narratives are executed then their ultimate result should be the establishment of a traceability link of some kind within the overall product as it is constructing the product essence. The key point is that the result of performing syntheses that are in epics that are broken into features that are broken into stories and then tasks for

individual team members is ultimately some part of the overall traceability structure at some level of abstraction and establishing the equivalences between the various hierarchies. But what we notice that is very significant is that the paths of synthesis construction that is referenced by the backlog items in their priority order does not amount to static structure within the product nor are the paths additive into networks of paths that represent the synthesis of the product. Rather synthesizing actions of the team produce products that ultimately underwrite the soundness and coherence of the software product without the paths of development themselves forming a network except perhaps in some managers planning tool. This is one way to see why planning tools like schedules that produce Gant charts are not really effective. Because each element of the product is a static datum of some kind when we connect them in a path and close the path we have a static repository of equivalences that can be revisited to allow verification and validation. But with regard to tasks and stories the parallel execution of them or their order of execution is ephemeral and not necessarily causally related to each other. Equivalences when established have a kind of casual necessity to them which is established by the verification and validation process that leads to properties of soundness for specific requirements and coherence of the product as a whole. The same thing cannot be said for the path of development steps that we see in terms of syntheses that we product as the result of stories and tasks. In any given synthesis many different kinds of work may be called upon to produce the desired result in the product traces all of these kinds of work are ephemeral and do not necessarily additively produce the results hoped for in the product. So we take this difference between narrative structure and product trace structures as part of our proof why narrative structures will not be able to stand in for traceability structures. Narrative structures are ephemeral and once they are executed then our epics, features, stories and tasks go into a done pile and are never referred to again except for auditing purposes, and they do not compose into anything that produces a static structure that can be checked for correctness like the trace

structure. On the other hand each part of the traceability structure once it is established is based on specific data that should be kept as the output of the development process, and this traceability data that are linked by traceability cross-links then establish a lasting picture of what is connected to what in the software product across the many kinds of different things that exist in the different hierarchies. And these paths not only close to produce soundness for a given requirement but they also form a network as a 2-groupoid that composes into an overall synthesis. On the other hand the finished tasks of the development process become just a heap of used synthetic descriptions that have no use after they are performed, except to make sure something happened during the development process that was supposed to happen.

Thus not only do we supply a mathematical underpinning for traceability that leads to composition and synthesis of the product structure using groupoids but we also see quite clearly why even though the development narratives can be represented as groupoid paths as well, that these do not compose into syntheses, but rather instead represent partial syntheses that are opportunistically connected with each other but have no necessary structure other than what goes together in the mind of the produce owner and other stakeholders. On the other hand the paths of the trace structure of crosslinks and the network of 2-groupoid synthesis in fact has a precedent order that is necessary due to its ordering according to types of order and because of that the groupoid paths can be seen as causal, because what is flowing down them in an inferential fashion is the increasing complexity of the ordering within the product and we posit that the best practice is to instill simpler orders prior to more difficult orders, so the type of causality flowing in the groupoid paths is the complexification of the system which as it spreads into the groupoid network becomes a means of accessing the various levels of complexity separately and for finding particular elements within the overall structure of the product. It also allows for various kinds of implicit inferencing about the structure of the product by providing a map as to what effects what within the overall structure of the product.

This implicit inference capacity is called systematicity[31] and it is what allows us to infer different things based on our experience from the way that things are connected in the product traceability structure. If we only have code to work from and no traceability paths then this inferential structure needs to be rebuilt every time we try to understand the code from studying directly the code base.

What do you do in the case where you have products with no, or limited traceability structures that were produced during development? What one should do is first make sure that test cases are linked to requirements. If there are no requirements for test cases then there is no way to tell if the tests were successful because there is no goal that is articulated for the tests. But then once you have that link created then it should be fairly straight forward to discover the way in which the tests like to the components which are tested in any given test. Thus the next set of linkages should be created to the component hierarchy. What may be missing all together is the functional model, however sometime tests are organized by function. But if the tests are not organized already by function the a functional analysis of the system needs to be undertaken. With an existing system this functional analysis is fairly straight forward, one merely creates a hierarch of all the functions of the system as a whole. It is then fairly easy to assign requirements to functions and functions to components. The last step is to produce an agent hierarchy for the performance hierarchy. It is this hierarchy that is going to embody and execute the components as separate strands of computation. Thus this can often be abstracted from the component architecture. Once the agent hierarchy exists and has been mapped to the component hierarchy then the last connection to be made is the allocation of the functions to the component hierarchy. This is the way to build up the hierarchies with their crosslinks when they do not already exist for a product that has been built which has no trace structure. One thing that needs to be mentioned is that although to establish soundness the crosslinks should

[31] Categorial Compositionality: a Category Theory Explanation for the Systematicity of Human Cognition. Public Library of Science, 2010.

form a closed loop of equivalences using the groupoid formalism between all the hierarchies. But if there is one break in the closed loop even though soundness has not been achieved it is still possible to do the mapping by following the mapping backwards and forwards and all the other links can then be established implicitly in that manner. So for instance it is possible to leave the architecture to function link broken and the trace structure will still function to give one traceability even if it does not give soundness for any given link and coherence for all the links for which closed loops are necessary.

Agile Traceability

Traceability and Agility. Given the importance of *traceability* for maintaining the intelligibility, soundness, and coherence of the product throughout the lifecycle, it is hard to understand why Agile advocates believe that they can do without it and in many cases have abandoned it. Interestingly, it is the focus on the effectiveness of producing working software that calls for everything extrinsic to be dropped from the lifecycle that does not directly contribute to the working code. And it is true that the traceability scaffolding does not directly contribute to the working software running. However, it contributes to its intelligibility and many of its other properties like completeness, consistency, verifiability, clarity, validity and coherence. Agility is short sighted in this respect if one considers the disasters that historically forged the idea of developing requirements and functional models. Unfortunately, in the guise of efficiency and expediency these attributes and architectural models are forgotten in the pressure to produce working code as soon as feasibly possible. Lean does not help either because it tends to see the extrinsic framework of traceability as something that might be waste, and something that, if needed, should be done just in time. In the *absence* of traceability we rely on our ability to directly read the source code and we try to see the big picture that tells us how the code realizes the product essence, but without the supports that would make the product (as a whole) intelligible. There is a need for more than just efficiency and effectiveness, the kernel of the system to be built must be taken into account. If we do not have a grasp on that kernel, then certainly the development process cannot be brought under control. The question is: Are we effective and efficient at doing *what*? The ‘what’ is answered by the traceability structure itself. What are the requirements that answer to customer needs? What are the functions of the System that give the wholeness that will satisfy those requirements? What is the performance structure of the system that will allow it to deliver those capabilities in a timely fashion via the partition of agency? What are the components of this system that articulate the architecture? What are the tests that will determine if the system meets the requirements, and will work in the operational environment, and is coherent within itself? Once we know **what** we are building, i.e., its essence, which is a series of constraints related to different types of order, then we can consider doing it effectively and efficiently, i.e., being efficacious in the development process.

So, the traceability structure that is extrinsic to the source code is what becomes the product, which is executable as working software. For example, if you don't know the requirements, then it is not possible to have the criteria that would allow you to say you have passed the tests that are done on the system in the end when it is integrated. If you don't know the functional structure of the system, then you have no idea whether it is unified and a totality, which together define wholeness. If you don't know the architecture that allows for the performance characteristics to be met, then you do not know if the system will work in its intended environment. If you don't know the components from which it is constructed and how they fit together, then you have no idea how the different parts of the software will interact, nor how it will interface with its environment. If you don't know the test-case structure, then you will not know how it will perform and other Quality of Service, i.e., non-functional, characteristics. So, there is a big problem that needs to be addressed that no amount of efficiency and effectiveness will solve, which is What we are building, i.e., the essence of the software system. And we know that the essence of software development is intrinsically hard due to the software characteristics identified by F. Brooks[32][3], which are: complexity, conformity, changeability, and invisibility. Essence is defined by a series of constraints. Constraints come from the outside to constrain something, which is why the set of trace hierarchies are extrinsic to the system. They carry the constraints that make the essence visible. Agility applies to the intrinsic quality of the software as a executing mass, it attempts to effectively get some software source code up and running as soon as possible so it can be evolved from a prototype into a final product as quickly as possible. Lean works on the human organization that produces the code and tries to make that efficient as possible by eliminating waste. But efficaciousness that encompasses both efficiency and effectiveness is also related to the What of the software that is being developed. *No matter how effective and efficient you are, if the What is wrong then the production process cannot be efficacious*[33]. The What of the software essence can only be known via the extrinsic structures and this is because the essence of the product is the embodiment of external constraints on the software product.

Transforming Narrative Structures. Agile Software Development approaches, especially Scrum, concentrate on the self-organizing production of an adequate product backlog that contains narratives (Themes, Epics, Features[34], Stories[35]) which (if they are called 'user stories' derived from Usecases) have a canonical form of . . .

Role *R* wants Feature *F* because of Benefit *B*

Now, what we notice about this is that it mirrors the Requirements hierarchy, Architectural hierarchy, and Functional hierarchy, but through a transformation where Roles are differentiations in the Architecture (which can be Software Agents as well as People). Features

[32] Brooks, Frederick P. The Mythical Man-Month: Essays on Software Engineering, 2012.
[33] Ries, Eric. The Lean Startup. New York: Crown Business, 2011.
[34] Leffingwell, Dean. Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise. Upper Saddle River, NJ: Addison-Wesley, 2011.
[35] Cohn, Mike. User Stories Applied: For Agile Software Development. Boston: Addison-Wesley, 2004.

are expressions of Functionality as seen from the point of view of the User, and Benefits are the side effects of realized Requirements. These elements are dispersed and fused together rather than being in separate hierarchies within the narrative. The reason that this can be done in this way is that Agents and Functions are partially ordered as a domain, while requirements are unordered. That means that the stories that fuse them together can be developed in almost any order that is not prohibited by the precedence of intrinsic dependencies, and this allows high value items in the eyes of the customer to be developed first so that the most valuable part of the system can be seen to manifest as working software that can be demonstrated as soon as possible. Because this canonical formula is a slice of a Usecase with a known benefit, then it can be part of the overall vision of the way that agents and functions interact within the system. So, through the Usecase the stories can be given initial coherence and a vision of the sequence of the stories. As a result, they can be developed in almost any order because the requirements, functions, and agency of the system are aligned in that local story. Thus, in many cases teams just start off with stories, perhaps written in this canonical form, hoping for the best. Now, with small systems, this probably works well for creating prototypes quickly. However, when we are dealing with *larger* systems, or when we get lost in our development, it may be necessary to fall back in a spike (a reverse sprint that adds necessary definition to the system so that the product backlog can be fleshed out and made coherent) and to further develop the order type hierarchies that are needed to give a picture of the wholeness of the system.

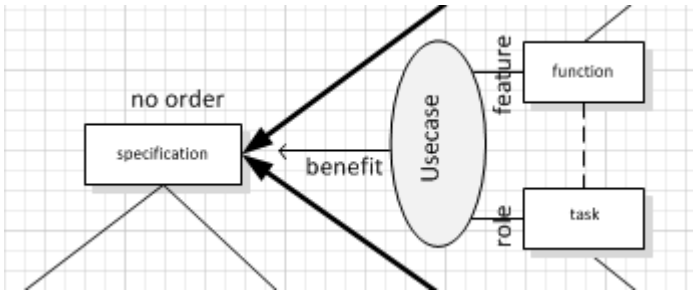


Figure 14. close up of Figure 4

Here, the picture of the system that is to be built is contained in the narrative hierarchy which is different from the order type hierarchies, and it is in these stories (in the narrative hierarchy) that contain descriptions of syntheses that are needed within the development process. The plan for developing the system refers to its essence contained in the crosslinks between the order type hierarchies and a model that can be constructed based on that superstructure using minimal method representations. In this way we identify the Picture, the Plan, and the Model that are necessary to produce the Whole Schema of the system. Picture is in the narrative syntheses that appear in the Product Backlog. Plan is based on the superstructure of the order type hierarchies that encompass the wholeness of the system as it evolves.

Model is built from minimal methods that can be used to represent the systems parts in a way that can be analyzed and synthesized using UML, SysML, or Domain Specific Languages that capture the Design of the System. The System is part of a Super-system that includes the integration and testing environment (or the meta-system) for the System. The System plus its picture, plan, and model equals the Super-system. To get the System we must separate it out from the Meta-systemic operating and test environment on the one hand, and the Super-synthesis of the Super-system on the other hand, and break it off from its picture, plan, and model. One of the major problems is that the Picture, Plan, and Model do not equal the Whole system. You can only realize the whole system by producing the super-system as a super-synthesis and then backing out of that the whole system. It is because of this problem that we cannot produce the whole system directly but only indirectly via the meta-system. The super-synthesis from which it must emerge must have a plan based on the vision of the whole that comes from the order type hierarchies and their crosslinks, as well as models based on minimal system representations (state machine, petri net, objects, functional flows, virtual layered machines, Usecases, DARTS, etc. such as found in UML and SysML). The meta-system and the super-system synthesis are inverses of each other. The meta-system is the operating environment for the system that occurs within the super-system. That super-system is a super-synthesis made up of many partial syntheses that appear in narratives in the backlog as fusions of requirements, architecture, and functions (as represented via Roles, Features, or Capabilities, and Benefits). The partial syntheses need to be organized so that they create a synthesis of the system. But, the synthesis of the system cannot be produced directly, rather it must be pictured, planned, and modeled in order to produce the images to coordinate the efforts of the team of the whole system that is to be produced that will guide development. The team, in its self-organizing work, produces the meta-system within the super-system that contains the entire work environment out of which the system is produced. For example, it produces both the source code for the system, as well as the testing of the system. The system must eventually be broken out as a Turing machine from this universal Turing machine of the testing environment. The system must eventually be broken off from the pictures, plans, and models that were used to envision how it would work before it actually existed in an executable form in which all the parts worked together properly. If the entire super-system with its

meta-system is not projected by the self-organization of the team, then there is no environment in which the System can take form. Look at any building that is built, there is a building site around it, where materials are staged and the parts of the building are organized and worked on before the building is assembled. The deconstructed meta-system of the building site is necessary to be built first before the system we are building can be put together. Pictures, Plans, and Models are what Architects use to plan the structure of buildings. We need to do the same thing when we build software. We need to have pictures of what the system would look like when it is built, which are normally referred to as prototypes. We need to have plans that address all of the elements within the software. Plans are determined by the types of order that are needed, and thus are based on the trace structure that gives planning all the elements and their relationships and it is this trace structure that allows actionable plans to be created and executed. Furthermore, there must be models that rise above the delocalization and decoherence of the source code base to give abstract diagrams of how the parts of the system relate to each other and how they would work dynamically. This is mostly captured by state machines and petri nets that are used to approximate Turing machines, which we see in the slices of our design representations.

If we have canonical narratives then the transformation from the narratives to the trace hierarchies is made possible by translating the benefits into requirements, the roles into agents, and the features into functions. It should be possible to construct these hierarchies based on the given narratives in canonical form and thus figure out what is missing, or find the unknown necessary precedent order between the components. By constructing the hierarchies we find out what is missing in our synthesized conceptions of the system that is confined to the narratives. If this can be accomplished, then it should be possible to fix the narratives in the backlog and to begin the production again with the new insights that come from the broad overview of the product that the trace structure represents. Thus, there is no reason that this cannot be done *just in time* and as needed even though the trace structure is extrinsic. This is the only way to correctly envision the structure of the essence, the What that is being built. Thus, there is no intrinsic conflict between Traceability and Agility or Lean development, in fact, there is complementarity because it gives insight into the What, and this gives intelligibility to the product and serves to give its elements coherence, as well as completeness, consistency, and verifiability.

Once we understand what traceability does for us, and the fact that we are going to have to decide on the precedent order for development anyway, we should recognize that it is traceability that allows precedent order to be discovered and made explicit. In fact, since

narratives are just fusions of requirements, agents, and functions, it would be better to create those fusions from the separate hierarchies, via the Usecases, rather than just making up what we can think of at a given time and adding it to the backlog as it occurs to us. It also helps to have a view of the 'end to end' causality within the system that is being developed. This can be achieved by articulating the virtual layered machine as a Gurevich Abstract State Machine (GASM)[36] .

Therefore, we should think again about what is the most efficacious way to proceed in our development. Because they are hierarchies of nodes, they do not have to be created all at once as we might do in a waterfall or spiral model. Rather, we can create the hierarchies as we explore the requirements and the design space. We can leave high level stubs for those nodes that we do not know yet, and drill down to flesh out those nodes in hierarchies that we do know. Then we can create Usecases that link agents, functions, and requirements through the interface of roles, features, and benefits. Narratives, as slices of Usecases, are syntheses that balance the analysis of the requirements, agent, and functional decompositions. Those syntheses should be aimed at producing components of the system and integrating them together. Thus, the components represent the realization of the synthesis described in the narrative as 'working software'. At each stage the hierarchies would demonstrate how complete the conceptualization of the system is in its realization during development. They would allow the precedent order to be discovered and explicitly represented because user priority is not the only kind of consideration when building a system. Features are balanced by capabilities that make the features possible. Both are merely different types of functionality within the system, some of which are the scaffolding on which the features must rest in order to work. This does not preclude test driven development. We can easily move from the fusion of the first three hierarchies that appear in narrative to the test case prior to developing the code that implements the components of the system. Having explicit hierarchies with specific equivalence relations between them makes it easier to have more flexible software development processes.

Now we see why Traceability is important in System Development. It is because traceability is the extrinsic structure that carries the constraints that define the essence of what is being built. We can see that this structure is indeed extrinsic to both the effectiveness and efficiency of development, but it is necessary for us to know the intrinsic whatness of the product being built. Also, this structure should be what lasts across multiple lifecycles of projects that add features to a product. Note that just because this trace structure is extrinsic, it does not mean that it is not significant or relevant since it does give intelligibility, soundness, and coherence to the product if the cycles of cross-links are closed at the leaf nodes. It can be developed based on canonic narratives that are used in Agile Scum models of development, and it can be developed in a 'just in time' manner just when the project loses sight of its technical goals, as in a spike, or as a result of the work of the Product Owner who is responsible for keeping the entire product in his sights. Traceability will allow him to derive narratives systematically and determine the

[36] Börger, E, and Robert F. Stärk. Abstract State Machines: A Method for High-Level System Design and Analysis. Berlin: Springer, 2003.

precedent order of the development of various components, which should influence the priority that they are given in the product backlog.

References

Börger, E, and Robert F. Stärk. Abstract State Machines: A Method for High-Level System Design and Analysis. Berlin: Springer, 2003.

Cleland-Huang, Jane, Orlena Gotel, and Andrea Zisman. Software and Systems Traceability. London: Springer, 2012.

Cohn, Mike. Agile Estimating and Planning. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, 2006.

Cohn, Mike. Succeeding with Agile: Software Development Using Scrum. Upper Saddle River, NJ: Addison-Wesley, 2010.

Cohn, Mike. User Stories Applied: For Agile Software Development. Boston: Addison-Wesley, 2004.

Coplien, James O, and Gertrud Bjørnvig. Lean Architecture for Agile Software Development. Chichester, West Sussex, UK: Wiley, 2010.

Hood, Colin. Requirements Management: The Interface between Requirements Development and All Other Systems Engineering Processes. Berlin: Springer, 2008.

Hull, Elizabeth, Ken Jackson, and Jeremy Dick. Requirements Engineering. London: Springer, 2005

Klir, George J. Architecture of Systems Problem Solving. New York: Plenum Press, 1985.

Kotonya, Gerald, and Ian Sommerville. Requirements Engineering: Processes and Techniques. Chichester: J. Wiley, 1998.

Leffingwell, Dean, and Don Widrig. Managing Software Requirements: A Use Case Approach. Boston: Addison-Wesley, 2003.

Leffingwell, Dean. Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise. Upper Saddle River, NJ: Addison-Wesley, 2011.

Proceedings of the 1st Workshop on Agile Requirements Engineering. New York, NY: ACM, 2011.

Rinzler, Ben. Telling Stories: A Short Path to Writing Better Software Requirements. Indianapolis, IN: Wiley Pub, 2009.

Robertson, Suzanne, and James Robertson. Mastering the Requirements Process. Harlow: Addison-Wesley, 1999.

Rubin, Kenneth S. Essential Scrum: A Practical Guide to the Most Popular Agile Process. Upper Saddle River, NJ: Addison-Wesley, 2012.

Sommerville, Ian, and Pete Sawyer. Requirements Engineering: A Good Practice Guide. Chichester: Wiley, 1997.

Wisse, Pieter. Metapattern: Context and Time in Information Models. Boston: Addison-Wesley, 2001.

Biography

Kent Palmer is a Real-time Software Engineer, and Systems Engineer. He has a Ph.D. in Sociology and Philosophy of Science from the London School of Economics with a dissertation titled The Structure of Theoretical Systems in Relation to Emergence. He has worked in Aerospace Industry for 30 years recently transitioning to Agile at Scale consulting in commercial firms. He also has a Ph.D. in Systems Engineering with a dissertation titled Emergent Design. His CV and several monographs related to Special Systems theory are available at http://works.bepress.com/kent_palmer. His homepage for various works on Systems Theory, Systems Engineering, Sociological Theory and other subjects is at http://archonic.net. His most recent dissertation is at http://about.me/emergentdesign. See also http://scaleagile.com, http://agiletheory.com, and http://onticity.com. His resume and recent papers are at http://kentpalmer.name.

Reviewers Comments:

Referee(s)' Comments to Author:

Reviewing: 1

Comments to the Author

I found this paper very interested and I agree with the general thesis put forward. However,m I found that it was too long and in some sections not focussed enough (I felt that there were digressiosn which woudl be acceptable in a thesis chapter but are too diverting for such a paper). I also believe that a concrete example needs ot be included followed by a discussion of this exmaple (e.g. pros and cons).

Reviewing: 2

Comments to the Author

Although I found some very interesting ideas and trains of thought in this paper, albeit at quite an abstract level, it seems to be an early draft rather than a mature paper for final review.

The paper proposes an abstract model which is visualised in Figure 9 'The Essence of the Software Product with Traceability relations and Minimal Methods' and makes the point how important traceability and traceability control are throughout the lifecycle of a product.

Also, the paper emphasises the importance of not throwing overboard traceability control when implementing agile (or alternatively lean) Software development approaches.

The paper needs significant rework in order to do justice to the very interesting concepts that are discussed in parts of the paper.

I suggest the following improvements be made:

1. The paper should be clearly structured with a logical sequence of chapters that are numbered and have a clear title. This structure should be in line with the purpose of the paper and include for example a 'real' introduction, a chapter that explains in detail the abstract concepts that are proposed, a chapter on the importance of traceability and its control throughout the entire life cycle of a software product, a chapter on agile and lean

approaches to software development (including evidence if any that traceability is neglected in these approaches), a discussion/analysis chapter, followed by a conclusions chapter.

I have not made these changes. I believe that the structure of the paper works the way it is. It is possible to rewrite the paper completely as suggested but I am not sure that there would be any gain from this rewriting. The current paper has been substantially rewritten and reworked and hopefully the structural problems have been solved for the most part in the latest version.

2. The scope seems to be primarily Software Engineering. This should be made clearer.

This paper applies to both Software and Systems Engineering both need traceability. The emphasis is on software for examples.

3. In the paper there is a mix of very informal familiar language and highly complex abstract language with technical expressions that are not easy to understand, not even by experts in the field. I suggest that only necessary technical terms are used and properly explained, when they are first used. The readership of the SE journal is global, which should be kept in mind.

Explaining all the terms in this way would cause the paper to be too long. There are references to my dissertation in the paper which has a glossary that explains many of the terms.

4. The readability of Figures 1, 3, 4 and 5 needs to be improved.

Figures have been redrawn in some cases to make them clearer.

5 Figure 8 is not easy to understand and should either be improved and well explained in the text or removed.

Not sure what figure is being referred to now. If it is the diagram now 9 then it is explained in the text.

6. All Figures should be mentioned in the text before the Figure and explained in the vicinity of the Figure (not several pages before or after the Figure).

The attempt is made to make the diagrams balanced within the text so that they are equall spaced out on the pages. That is why they are positioned as they are.

7. The term 'test' or 'testing' is used. It should be made clear that in the Software Engineering context this has a different meaning than would be the case in a typical Systems Engineering context, where only a small part of the design verification or product verification activities against the product or system requirements would consist of (formal) tests. Rather these activities may often be inspections, reviews or demonstrations that are less formal and therefore less costly.

It is assumed that the audience in a Systems Engineering Journal knows that and this does not need to be spelled out as it is in the Systems Body of Knowledge.

8. Both Agile and Lean approaches are criticised for neglecting traceability related concerns in light of the importance of having and maintaining traceability control. The paper should show clear evidence that this is actually the case (if it is actually the case). If there is no evidence, then maybe the paper should only make the point that there is a danger of neglecting it, if agile or lean approaches are applied without the appropriate care.

Lean sees it as waste. Agile sees it as not linked to executing code.

I think that the paper has some good potential, but serious rework is needed to make it well structured and readable for the journal's global readers.