

The Essential Nature of Product Traceability

and its relation to Agile Approaches

Kent Palmer

kent@palmer.name

714-633-9508

<http://kdp.me>

Copyright © 2013 by Kent Palmer. Published and used by INCOSE with permission. 20121031 Draft 02 PTS01a05.doc

Abstract. Discussion of the essential features of product traceability maps in relation to requirements, architecture, functional models, components and tests as a set of order type hierarchies and their crosslinks. The paper lays out the structure of these ideal traceability relations which define the essence of the product under development. Then the intrinsic connection of these trace relations to the representations of the product design is discussed. The importance of the trace relations to the product are made clear and then abandonment of traceability in Agile approaches is discussed. A way to transform between canonic narrative (story) representations that appear in the product backlog and the canonical form of the trace structure of the product is discussed. The fact that it is possible to transform back and forth between narrative and traditional representations of trace structures as crosslinks between hierarchically related order-type elements, and the fact that trace structures can be produced in a just in time fashion that evolves during product development shows that it these trace structures can be used in both an agile and lean fashion within the development process. Also it is shown how by doing trace structure outside the narrative representation has in a traditional way has the additional benefit of helping to determine the precedent order of development so rework can be avoided by developing components out of the sequence that their technological infrastructure and architectural expression of capability demands. Thus network trace structures using this model can be seen as an essential tool for product owners to use to help structure and prioritize the backlog in the Agile approach to software and systems development.

Keywords: Traceability, Evolving Development Information, Workitem Information Linkage Network, Application Lifecycle Management, Agile, Product Essence, Requirements, Functional Models, Distributed System Agency Models, Architecture, Test, Order Constraints, Order Types, Linking, Narratives, Stories, Usecases, Systems Development, Coherence,

Soundness, Consistency, Completeness, Verification, Mathematical Foundations of System Development, Architectural Methods

The Essence of Traceability

What Is Traceability?

Introduction. This essay lays out a theory of traceability structures and how they may be transformed by Agile Approaches. This is a question 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 does not impede agility. The forgetting of 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 increased 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 the ability to trace to tests to verify requirements; however traceability is really the product's highest level lasting structure. And this structure is mathematically necessitated. So it is really a means of grasping the essence of the product that should last as long as the product lasts. Traceability is the access we create for ourselves to the lasting essence of our product, which otherwise might be hidden even though we are its developers. We will speak of the essence of the product perduring, which means that it lasts throughout the life of the product. That essence is there whether we have access to it or not. The best thing is to build the access to the essence of the product through the trace structure as we are building the product itself. If we do not do this then we may be blind to the essence of the product we are building. So traceability is the means of making visible during development of the essence of the product. And if we do not know what it is then to that extent we have lost control of our product.

Traceability used to be between documents, and then it became links between elements in documents, and then with the advent of Requirements Tools it became possible to have separate requirements nodes which could be linked to each other, and then generate the documents. The new wave is to have Applications Lifecycle Management tools which allow links between many different types of document nodes including requirements, architectural components, tests, changesets of source code, issues, defects, usecases, narrative stories and other administrator defined information workitems stored in a single database, rather than only having a database of requirements or a separate database of defects, or changes to source code under configuration management. In this article we will call this networked traceability with links between workitem nodes of different types that give us direct access to information about the system that can evolve along with the system because developers as a team have access to

that information during development and can change it as needed to reflect the current state of the product. Here we are suggesting that these workitems do not have merely a freeform structure but there are mathematical and methodological perspectives that should determine the shape of this network of linkages between certain types of work items.

This organization of the development information about the product in linked workitems reflects has a mathematical basis which also determines the development process itself in its broadest scope, and the minimal methods we have for representing the architectural structure of the system under development. What is proposed in this paper is a particular networked representation that consists of hierarchies of different order typed nodes, and we explain that traceability is the crosslinks between the leaf nodes of this hierarchy. The links themselves are parent/child links within the hierarchies of nodes with the same anchor and the same order type, and the links between nodes in different hierarchies are directional crosslinks of specific kinds.

In this paper a specific network structure of linked nodes is suggested based on the essential structure of the systems product which includes software. The virtues of having traceability structures even in Agile approaches are made clear, and the relation to normal Agile ways of capturing this information via narrative stories is integrated with an overall approach to traceability that allows access to the product essence to be preserved, and it is suggested that this is a necessary tool in the domain of Scrum Product Owners who maintain product backlogs and should attempt to keep an overview of the product in mind as development proceeds and that networked traceability structures are a good way to do that kind of work.

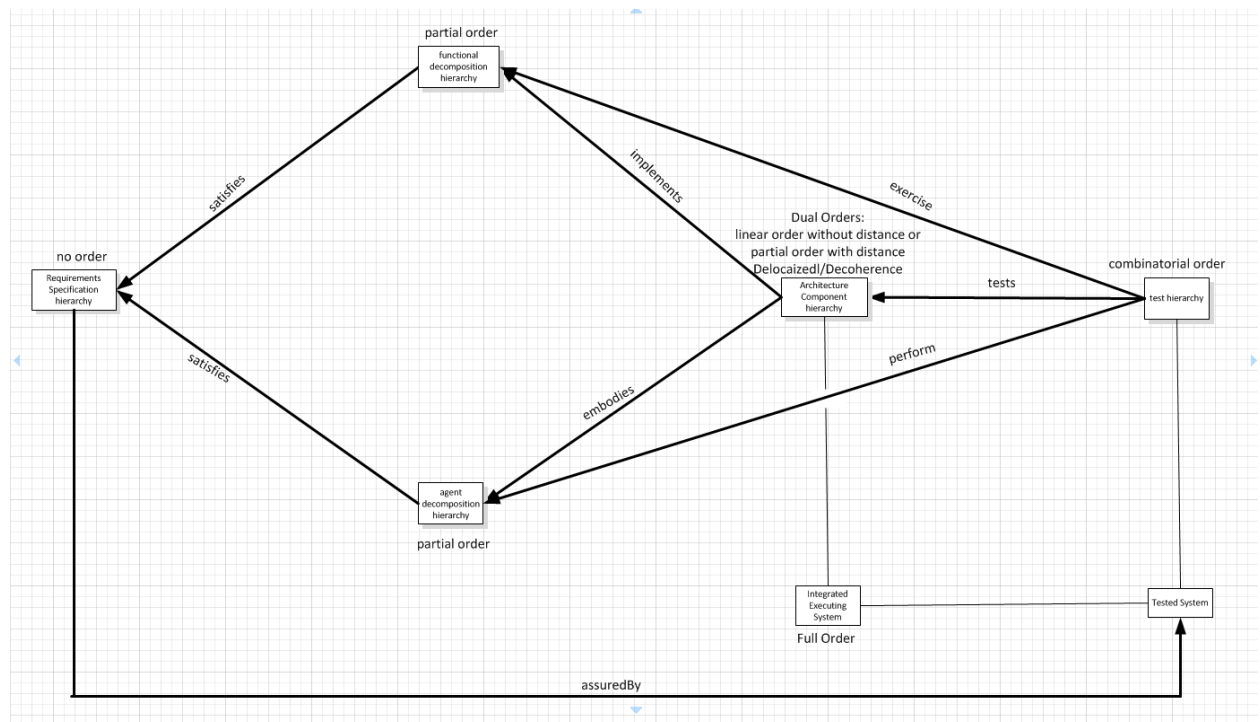
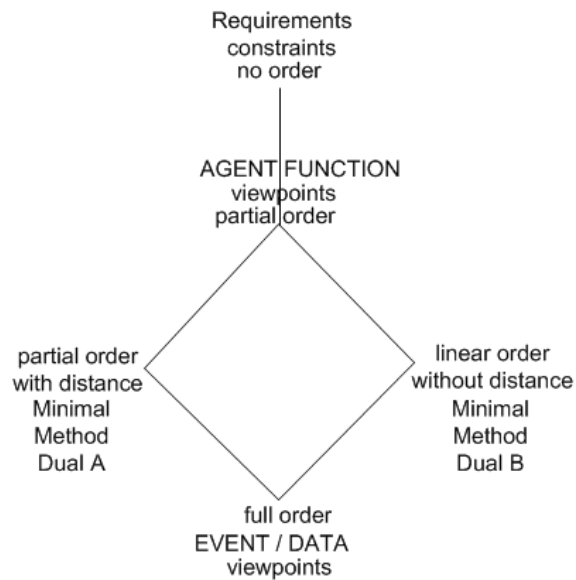


Figure 1

Trace Structure. We normally think of the trace structure as something added to the product during the development process which is something extra. But in fact the trace structure only gives us access to the essential structure of the product which is built in to the product as we are doing development. And this is necessitated by the mathematics underlying development which 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 which links Requirements, Agents, Functions, Components, 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, Components are Linearly Ordered or Partially Ordered with Distance. Executing Packages of Source Code in Modules that are Fully Ordered, and Tests are Combinatorially Ordered. These types of Orders are called by George Klir in Architecture of Systems Problem Solving “Methodological Distinctions”. 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 by us into it. Essentially the structure of the development process itself is driven by what order is necessary for us to introduce into the product each kind of order. And when the product is done at a given release it has 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 to us. So, for instance, Requirements as axiom-like statements have no order, and we have access to that 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 capabilities of the System. Functions are diffuse within the product and assigned to the agents that have the capabilities to perform the intentional actions of 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. Packages of Source Code arranged in modules are fully ordered as they execute producing a signature in spacetime. Testcases on the other hand are combinatorially ordered, a kind of order not discussed by George Klir but essential to making sure that the system works 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 the system will work when executed in its intended environment.



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 and in order for development to occur properly which means in a way that leads to a working product which we can understand. The V of the Lifecycle is driven by these kinds of orders and what is necessary to instill them sequentially in the product. So, for instance, it is canonical that we start with requirements, then develop architectural designs and decompose the product into levels designing each level until we get to the Code which we write as the embodiment of the identified components at the lowest level that nest up into higher level components of the architectural design. But once the source code is written for the implementation of the designed system then we start executing that code against module testcases, and then higher level test cases until the entire system is tested. In the back end of the V we are integrating, and then testing various levels of the integrated system which leads to verification and validation of the system. Verification makes sure that the system as tested meets the requirements, and Validation through the test environment being as close as possible to the actual intended environment of use makes sure that the system actually works in that environment for the intended purpose. The V lifecycle or any other lifecycle is constrained by the existence of the lattice of orders called the Methodological Distinctions by Klir. Software Production is the instilling of these orders in an orderly fashion. The ordering of order is called organization. We must organize ourselves and what we do in such a way as to instill order into the product in an orderly way. When we organize ourselves and execute our plans to create order in the product it just naturally occurs that the best way to do that is to lay down the strata of order in the product in the order that they occur in the lattice of Methodological Distinctions. So the strata of types of order determine not just the structure of the product but also the structure of the process by which we produce the

product in the widest possible sense. This is why products have structure, and why the processes by which those products are brought into existence have the structures that they do have by necessity supported by different kinds of work because it is written into the structure of mathematics that these are the possible orders and they are ordered by the lattice of Methodological Distinctions, and 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 being primarily determined by mathematics in its essential nature which exhibits the qualities of Hyper Being¹. Any order 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, if we want to be agile (effective) and lean (efficient) or what is both, i.e., efficacious, then we will respect the this necessary ordering of order within our development processes and our products. When we say we self organize as a team to produce the product we are organizing around this order which is predetermined by mathematics to apply to everything we create but especially software because it has few other constraints beyond the mathematical constructs, the Turing machine and the hardware architecture on which it executes. But just because the product is forced to take on its order in this fashion does not mean we have access to those strata of ordering within the product that become affective at different stages of production. To have access to this essential structure of the product we must build along with its traceability structures.

Traceability structures are a series of hierarchies that reflect the different types of elements with different kinds of order. So for instance we have a series of hierarchies for requirements, functions, agents, components, modules and tests. Each of these hierarchies has its own anchors and exist as having nodes of the given type that are linked to each other by parent child relations all the way down to the leaf nodes. All the nodes in a given hierarchy are of a designated type. There may also be other performance hierarchies 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. But the key point is that it is the cross-links between these hierarchies that are the traceability structure of the product. The hierarchies themselves merely represent the elements of a given type of ordering. What gives the unique signature of the product is the set of nodes in those hierarchies and the way that they are crosslinked together.

¹ See Ontology of Software article in Wild Software Meta-systems at http://works.bepress.com/kent_palmer. Wild Being was defined by Merleau-Ponty in The Visible and the Invisible which is also called difference by Derrida and Being crossed out by Heidegger. Software is the only known kind of Being that exemplifies the third meta-level of Being called by Plato the Third Kind of Being in the Timaeus.

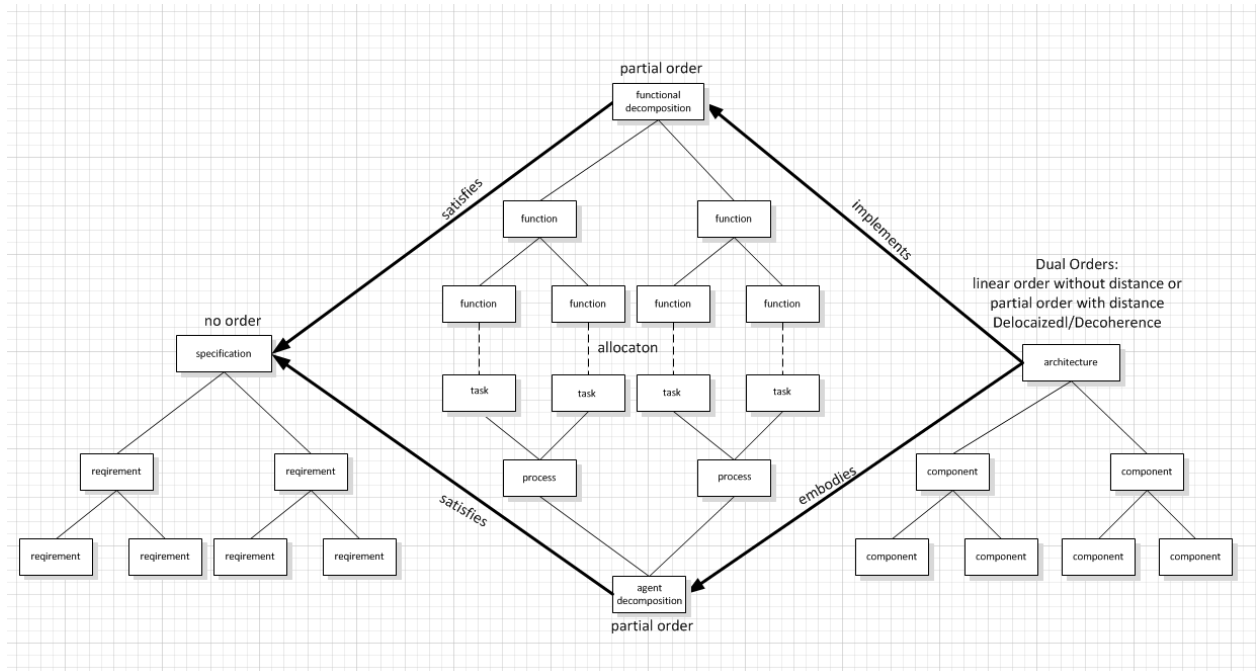


Figure 3

So, for instance, given a requirements hierarchy when the functional hierarchy is known we can crosslink them so that we say that a given function **satisfies** a requirements. But if it is a non-functional requirement then it would instead be linked to a performance hierarchy like the agent hierarchy and so a given architecture of the functions would then **satisfy** a non-functional requirement. Agents and Functions are both partially ordered and together they may model a domain². The fact that both agents and functions are partially ordered is taken advantage of by Agile approaches which say that these can be put into a backlog and done in the priority that gives the most value to the customer first. But this fact that the 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 order which if violated will cause rework because some things must be done before others as dictated by the architectural dependencies and also 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 necessarily contain elements either partially ordered with distance or linearly ordered without distance. This is to say that when we write the code then the design components become delocalized and decoherent within the source code. That is to say in the source code there is always some spreading out of the elements necessary to realize a given component. This is made easier to deal with through object oriented designs, but then this property of the system merely shifts 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

² These partial orders may be related to mathematical domain theory which is a theory of computation.
http://en.wikipedia.org/wiki/Domain_theory

for performance reasons, and they **implement** certain functions in the functional hierarchy. Once all the components have been defined then it is possible to go on to write the source code that **realizes** those 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 could have 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 testcases. These testcases when they exist at each level **exercise** the functions, **perform** the agents duties, and **test** the components. Finally when we complete the circle and connect the testcases back to the requirements we say that the tests **assure** the requirements. Fully closed circuits among the hierarchies of types of order makes the structure of the product sound³.

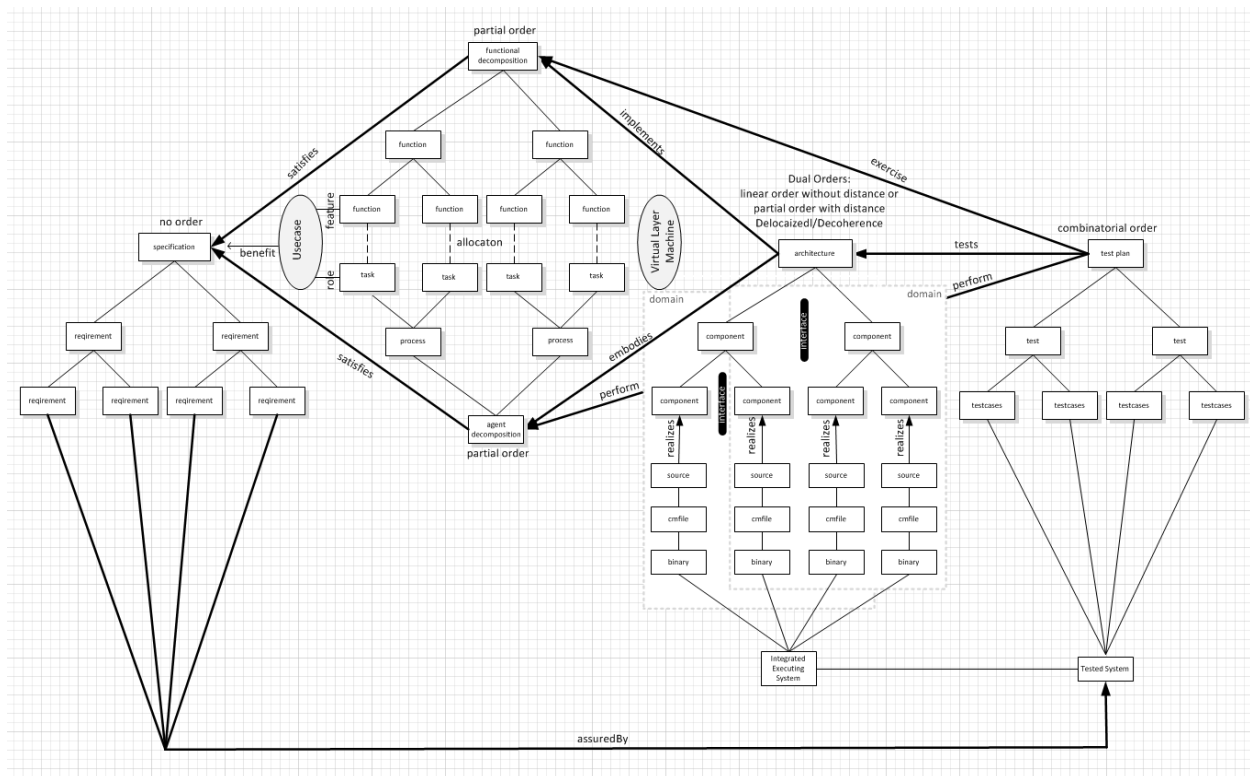


Figure 4

This structure of crosslinks between order type hierarchical nodes is the essential structure of the product which 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 the realizing source code of the system, these traces are easily lost, so that we lose access to them because we do not keep up the traces. And if we

³ <http://en.wikipedia.org/wiki/Soundness> Here we take soundness to mean that because circuits between the hierarchies of ordered elements are completely connected and the test results are affirmed then all the premises are take to be true, and this is because broken links are false premises as well as failed tests.

jettison traceability all together as has become popular in Agile development then we do not even have an old out of date trace structure to fall back on to start to build the traces necessary to make the product intelligible when things have gone wrong in development or 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. So the trace structure is an attribute of forethought, we build it along with the product because we want the product to remain intelligible not just to a few people who know the codebase but by everyone who has to deal with the product. The trace structure is like the contents of a book or the index, it gives access to the relevant and significant parts of the books content on demand. Without the 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.

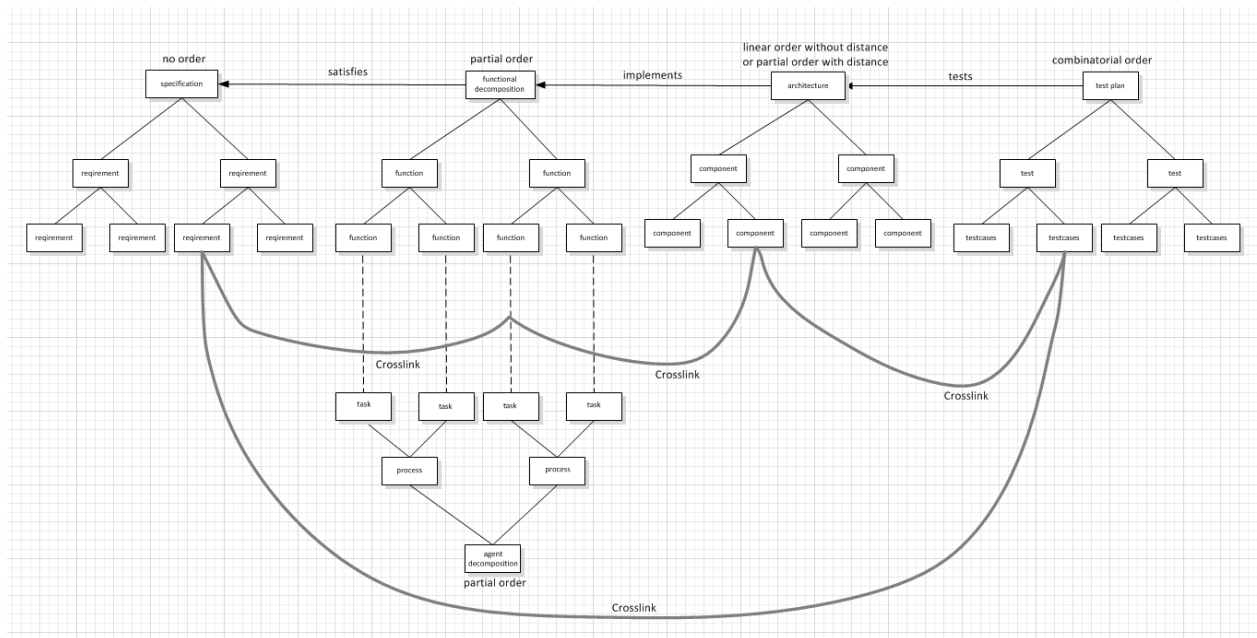
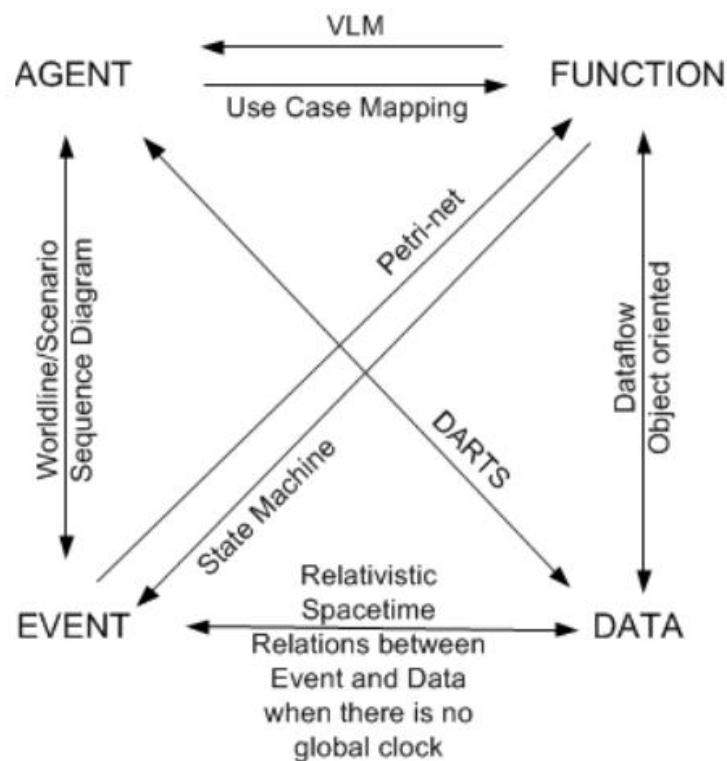


Figure 5 Closure of a single circuit gives soundness if tests passed and closure of all circuits gives coherence to product.

Kinds of Order and Minimal Method Representations. Another point of interest is that the types of order have an interesting structure in as much as there is one non-ordered type, two partially ordered types (agent and function), but then there is 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 relation between elements of different orders. Thus for instance between Agent and Function viewpoints on the design there are two minimal methods

which are 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 as different ways of representing the entities in the system. Between Agent and Data there is the DARTS⁴ 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 further in my paper “Hacking The Essence of Software”⁵. Between Event and Data there is a relativistic spacetime interval. The test software is the inverse of the system, as a testing environment or meta-system which can be represented exactly in the same way that the software itself under test is represented. There is then in the combinatoric order of testing an inverse mirroring of the entire system 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 just the structure of the product, and the process of developing the product but also our minimal representations of it that are slices of its Turing machine or universal Turing machine that forms the operating system in which the system under design operates.



⁴ 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.

⁵ See <http://kentpalmer.name>

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 representation of the essence of software systems which is 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. We must organize ourselves to produce artifacts with these orders layered in strata of the products we build in an orderly way. And that orderly way 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 exercises in execution the combinatorial order of the possible ways that the software can be executed to the greatest extent possible in order 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 executable 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 kind of mathematics that are mass-like which is geometry and topology as well as the real number line which is folded back into itself by 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 Brown⁶ or Bricken⁷, or Hellerstein⁸. Thus there is a way to think logically about executables through pervasion logic, so we do not have to be lost when we move across the line from design sets to mass executables the way we are today. Rather we can apply Boundary Logic to understand what is going on in the mixtures of masses created within our executables with 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 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 which implement them. This is important to traceability because there is a chasm

⁶ Spencer-Brown, G. Laws of Form. London: Allen & Unwin, 1969.

⁷ <http://www.boundarymath.org/> See also <http://www.boundaryinstitute.org/bi/>

⁸ Hellerstein, N S. Diamond: A Paradox Logic. New Jersey: World Scientific, 2010. Hellerstein, N S. Delta, a Paradox Logic. Singapore: World Scientific, 1997

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 with the system in integration and test and are exercising it and allowing the agents within it to perform their duties then we lose track of the trace structure only picking 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 except as we view it 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 which 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 while it is being built with set like regularity of the source files. The discipline and the challenge of building 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 itself as defects. So 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 that product. And that frame work 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 a canonical representation for each element of the system which we can use to reason about the workings of the system. For instance 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 as state machines with stacks or tapes and the protocols between them can be modeled with petri nets. DARTS⁹ gives us a way to represent the parallel and concurrent actors meant to execute in unison on different tasks performing different functionality 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 level machine that can be seen as an abstraction that encompasses the entire system and which can be represented by the Gurevich Abstract State Machine (ASM) structure¹⁰. We can use the Wisse Metapattern method¹¹ to understand the objects within the system that are interacting causally through the

⁹ Cf H. Gomaa Design and Analysis of Real-Time Systems

¹⁰ Gurevich, Yuri. Abstract State Machines: Theory and Applications : International Workshop, Asm 2000, Monte Verità, Switzerland, March 2000 : Proceedings. Berlin: Springer, 2000

¹¹ 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/>

rules of the Gurevich ASM. All this takes place under the umbrella of the unordered requirements of the system that reflect user needs. All the system tests must answer back to this umbrella of requirements 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 then it is 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 the requirements are met for the system.

Add Reality to the Formal Model

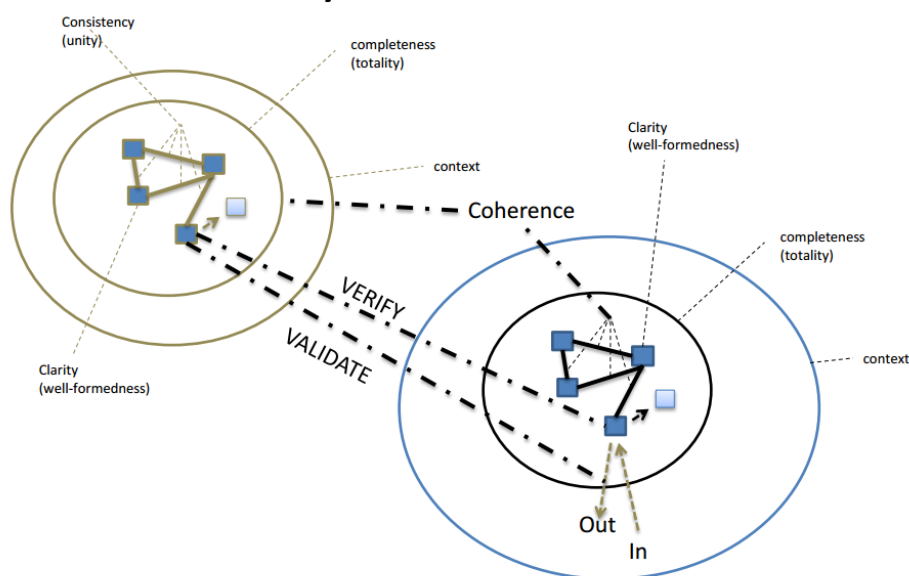


Figure 7

Traceability and Coherence. This cycle of the crosslinks 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 which is based on the aspects of Being truth, presence and identity, with the properties between these aspects of clarity (wellformedness), completeness and consistency. But when the formal system is confronted by the aspect of reality in testing then three other properties arise which are verifiability, validity, and coherence. Coherence is the relation between reality and identity. We run tests not just to verify and validate the software, but the software is also integrated and the limit of the integration of the software is its coherence, we are proving through our testing also that the software is coherent which is added by the architecture that was pictured, planed

and modeled during implementation¹². The link back of Test results to Requirements is through verification, and the link to the environment in which the system will be used is through validation, but it's link to itself through itself is via the property of coherence. And it is the completed circular paths of the crosslinks 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 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 making sure that the crosslinks between the order type hierarchies are circular and which when closed and assured become sound and and when all circuits at the leaf node level are closed then trace is complete indicating coherence of the system. Each hierarchy of order types has an anchor or root node which represent its unity, and it has leaf nodes that represent its totality. When all the order types are both unified and totalized, and crosslinks are established between them at the leaf nodes and those are complete then the circularity between the order types assures coherence. Coherence of the product with in itself is an essential characteristic. Building that coherence in is enhanced greatly by having a complete and consistent as well as clear trace structure. Abandoning the Trace structure not only decreases dramatically the intelligibility of the system but it also makes coherence of the end product much harder to achieve when the circular traces are not sound.

Aspects and Properties

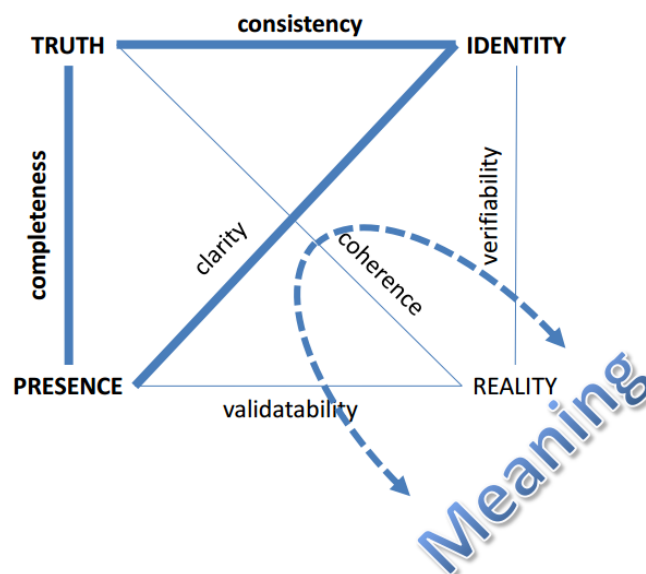


Figure 8

¹² See author's dissertation Emergent Design at <http://about.me/emergentdesign>

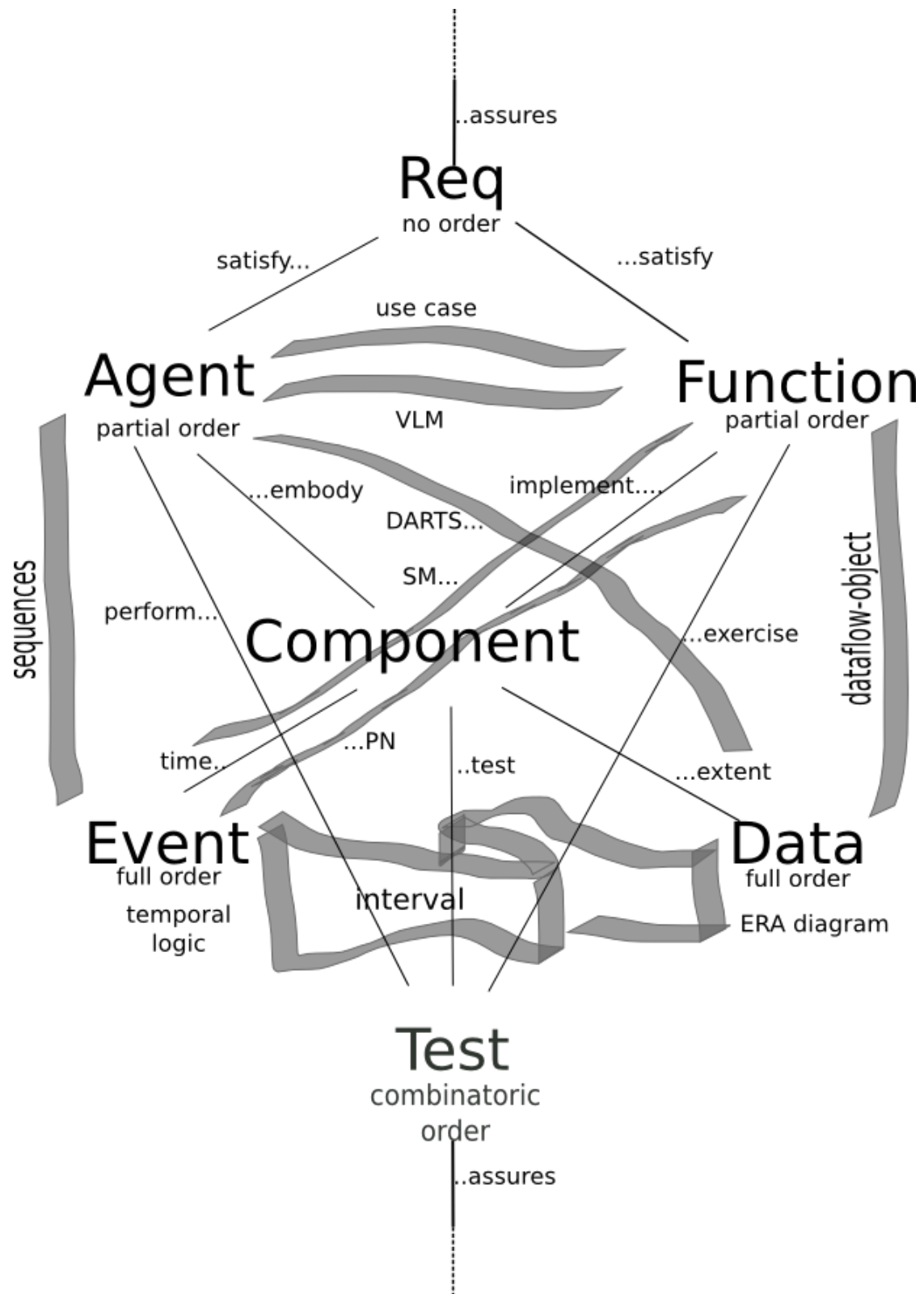


Figure 9. The Essence of the Software Product with Traceability network crosslink relations and Minimal Methods

Traceability and Agility. It is given this importance of traceability for maintaining the intelligibility, soundness and coherence of the product throughout the lifecycle hard to understand why Agile approaches 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, and coherence. Notice two properties drop out which are clarity and validity. Agility is short sighted in this respect in as much as the disasters that forged the idea of developing requirements, functional models, 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. The point is that there is more than just efficiency and effectiveness, there is also the kernel of the system to be built that 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 to then render it more effective or more efficient. The question is 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 to be built to 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? What are the tests under which we 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 is extrinsic to the source code that becomes the product as the executable that is working software, but it is not extrinsic to the what of that which is being built, it is in fact the only way to know the what. For instance 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 at 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 of 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 testcase structure then you do not know whether it does what it is supposed to do and its performance and other Quality of Service, i.e. non-functional, characteristics. So there is a big problem that needs to be addressed which 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 is

intrinsically hard due to the characteristics identified by F. Brooks¹³ which are complexity, conformity, changeability, invisibility. Essence is defined by a series of constraints. Constraints come from the outside to constrain something. And that is why the set of trace hierarchies are extrinsic to the system because 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 get some software source code up and running as soon as possible so it can be evolved from prototype into 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 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.

Agile Traceability

Transforming Narrative structures. Agile Software Development approaches especially Scrum concentrated on the production by the self-organizing team of an adequate product backlog that contains narratives (Themes, Epics, Features, Stories) which (if they are so called ‘user stories’¹⁴ 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, Architectural and Functional Hierarchies, but through a transformation where Roles are part of the Architecture, and can be Software Agents as well as People, Features are part of the Functional hierarchy as seen from the point of view of the User, and Benefits are the side effects of realized Requirements. But these elements are fused together rather than 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, and requirements are unordered, and so that means that the stories that fuse them together can be developed in almost any order not prohibited by precedence of intrinsic dependencies, and this allows high value items in the eyes of the customer to be developed first, so the most valuable part of the system can be seen to manifest as working software that can be demonstrated out of development as soon as possible. Because this canonical formula is a slice of a Usecase with a known benefit then it is part of an 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 vision of the sequence of the stories, and then they can be developed in almost any order because the requirements, functions and agency of the system is aligned in that local story. Thus in many cases teams just start off with stories,

¹³ Brooks, Frederick P. The Mythical Man-Month: Essays on Software Engineering. Reading, Mass: Addison-Wesley Pub. Co, 1995. http://en.wikipedia.org/wiki/No_Silver_Bullet

¹⁴ http://en.wikipedia.org/wiki/User_story

hopefully written in this canonical form, hoping for the best. And with small systems this probably works well to come up with prototypes quickly.

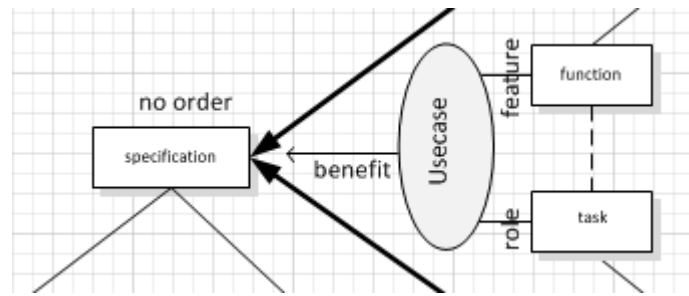


Figure 10. close up of Figure 4

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 develop the order type hierarchies further that are needed to give a picture of the wholeness of the system. Here the picture of the system to be built is contained in the narrative hierarchy which is different from the order type hierarchies and is in the stories 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 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 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 and the super-synthesis from which it must emerge that it is necessary to have the 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 but rather must be pictured, planned and modeled in order to produce the images to coordinate the efforts of the team of the whole system to be produced that 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 instance, it produces both the source code for the system, and also that for 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 before they are built. 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 was built which are normally prototypes. We need to have plans that address all of the elements within the software and those are determined by the types of order that it must have, and thus is based on the trace structure that gives planning all the elements and their relations so that detailed plans can be made and executed. And 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 which 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 that is 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 the unknown necessary precedent order between component. By constructing the hierarchies we find out about what is missing in our synthesized conceptions of the system confined to the narratives. With this done 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, it is the only way to get straight 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 it is complementarity because it gives insight into the What which gives intelligibility to the product and serves to make its properties clear among which is its coherence, as well as completeness, consistency and verifiability.

But once we understand what traceability does for us, and the fact that we are going to have to decide on the precedence order for development anyway, we should recognize that it is

traceability that allows that precedence order to be discovered and made explicit. And 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 they occur to us. It also helps to have a view of the end to end causality within the system to be developed which we can get by articulating the virtual layered machine as a Gurevich ASM.

Therefore, we should think again about what is the most efficacious way to proceed in our development. Because since 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 then the design space. We can leave high level those that we do not know yet, and drill down to flesh out those nodes in these hierarchies that we do know. And 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”. The hierarchies would show at each stage how complete the conceptualization of the system is in its realization during development. They would allow the precedence order to be discovered and explicitly represented because user priority is not the only kind of consideration in building the system. Features are balanced by capabilities that make the features possible and both are merely different types of functionality of the system some of which is the scaffolding on which the features must rest to work.

Thus we see why Traceability is important in System development, which is because it 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 effectiveness and efficiency of development, but is necessary for us to know the intrinsic *whatness* of the product being built. And also this structure should be what lasts across multiple lifecycles of projects that add features to a product. But just because it is extrinsic does not mean it is not significant and relevant in as much as it gives intelligibility, soundness and coherence to the product if the cycles of crosslinks 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 a spike, or as an activity of the Product Owner who is responsible for keeping the entire product in his sights, and allowing him to derive narratives systematically, and determine the precedence order of the development of various components which should influence the priority which 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

Systems Engineer and Software Engineer with 30 years of experience in Aerospace specializing in Requirements Engineering and Architectural Design as well as Process Engineering and Software Development Technology and more recently involved in Agile and Lean organizational transformations. See <http://scaleagile.com>, <http://agiletheory.com> <http://onticity.com> <http://kentpalmer.name>