

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.

© 2014 The Authors. Published by Elsevier B.V.
Selection and peer-review under responsibility of the University of Southern California.

Keywords: Traceability, Agile, Requirements, Architecture, Test, Coherence, Soundness

* Corresponding author. Tel.: +1-714-633-9508; http://kdp.me; http://kentpalmer.name (see site for original paper)
E-mail address: kent@palmer.name
1. The Essence of Traceability

1.1. 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.

1.2. 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.

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

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 install 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*. 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†, 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.

Figure 2. Allocation relations between Agent and Function partial orders

---

* 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

† See ‘Hacking the Essence of Software’ by the author at http://kentpalmer.name
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[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 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 as 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 3. Cross-links are the traceability structure

1.4. 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.

2. Agile Traceability

2.1. 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, and coherence. Notice that with Agile two properties drop out, which are clarity and validity. 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[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[4]. 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.

2.2. 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[5], Stories[6]) 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 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 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.
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) [7].

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

References