JOIN OVER 100,000 PROS! Join Log In | Member Benefits

STICKYMINDS

Agile Testing DevOps Security AI

The Trouble with Tracing: Traceability Dissected ARTICLE

By Steve Berczuk , Brad Appleton Robert Cowham | November 30, 2005

SUMMARY

Traceability! Some crave it. others cringe at the very mention of it. For hardcore configuration managers and requirements and systems engineers, it is a fundamental commandment of “responsible” software development. For many hardcore agilists and other developers, the very word evokes a strong “gag” reflex, along with feelings of pain and frustration. Traceability requires work and discipline! So how does traceability add value to our business and how can we make it easier?

Ten Commandments of Traceability

How did traceability become so revered and hated at the same time? Is it just because it’s hard? Are the people who come up with it hard-nosed? Are the folks who complain about it just hard-headed? Or are the activities performed and constraints imposed in the name of traceability simply stated too “hard and fast” with little room for flexibility.

Like it or not, many interpret traditional traceability and CM as rigidly mandating the following “ten commandments” upon the development process:

1. *Thou shalt* create and maintain linkages between every requirement to all other requirements that it impacts or decomposes into
2. *Thou shalt* create and maintain linkages between every requirement and every configuration item that participates in realizing the requirement
3. *Thou shalt* create and maintain linkages from every requirement to all design elements and interfaces that describe or model its implementation
4. *Thou shalt* create and maintain linkages from every design element and interface to every source code line and module that implements all or part of

We use cookies to ensure you get the best experience on our website.
Read More
Accept

subroutine that verifies the requirement is correctly implemented in the product

6. *Thou shalt* track and record every request for change (be it for corrective, perfective, adaptive, or preventive purposes) and not allow any changes to any artifact without first ensuring that the change was approved and developer was authorized to make it
7. *Thou shalt* create and maintain linkages from every change-request to every corresponding piece of documentation and code/test that is created or modified, along with who made the change, when, where, and why and with verifiable (via audit) evidence that the proper process and procedures were executed to ensure mandated appropriate levels of control and peer review
8. *Thou shalt* create an maintain linkages from each change request to its corresponding tasks and the corresponding changes for each task
9. *Thou shalt* create and maintain linkages from each change and change-request into the sets of artifacts, elements, codelines, builds, projects, and variants into which it is incorporated
10. *Thou shalt* keep all the above data consistent, correct, and complete at all times, and use it to generate any reports or audits necessary upon demand

Most of these are things that can and should be accomplished if traceability exists. Many of them seem much harder or more effort intensive than they really are (or need to be). In other cases, the relative ease or difficulty may be matter of the level of detail to which something should be traced. If things are done without keeping in mind the intended purpose and business use of traceability, then many complaints and inefficiencies can result.

Nine Gripes against Traceability

So what exactly are the things that so many developers, and more recently, some of the agile and lean camps don't like about traceability? Common complaints are as follows (decide for yourself whether they are invalid or justified, or if there is a better and easier way:

- It causes more artifacts, more overhead and more work to create and maintain traceability matrices that are going to get hopelessly out of date because changes are inevitable
- It encourages "analysis paralysis" and heavy "up front" planning and design
- It imposes unnecessary restrictions upon the speed and productivity of development
- It is a fool's errand that amounts to smoke and mirrors to provide an illusion about control and certainty regarding mostly redundant pieces of information already in the code and the tests
- It goes against the values in the "Agile Manifesto" because traceability

We use cookies to ensure you get the best experience on our website.

- It makes it harder to make necessary changes and encourages rigidity rather than flexibility
- It assumes a waterfall-based lifecycle with end-of-phase hand-offs to different functional groups
- It increases the amount of "software waste" for time spent seeking information and increases the duration and amount of "inventory" (partially completed, "in progress" functionality)
- It doesn't readily fit iterative development working in short-cycles with close customer collaboration

Eight Reasons for Traceability

If traceability is so much work, and if the return-on-investment is so hard for so many to see, then why do we bother? Or rather, why *should* we bother? Other than obvious contractual obligation, legal liability, or the fact that various CM and software engineering tomes declare "thou shalt have traceability," what real needs does it serve? What does it let me do that I couldn't otherwise do very easily? Several reasons commonly given are:

- Validate we built what the customer actually agreed to pay us to build, that we built the "right thing" and that we "did things right" when building it
- Verify that we delivered every file and feature/fix that we said we would, and nothing that we said we wouldn't
- Identify and isolate a set of changes that were made, and be able to either back-out the change from an existing version, and/or reintroduce the change to other versions
- Assess the impact upon the design, code, tests and other aspects of the system for a proposed change-request (this in turn helps us determine effort, risk, cost, and the other affected groups)
- Maintain an audit trail about who did what, when, where, why and how—so we can prove what we should and shouldn't be held accountable for when the customer, or management or some other formal authority demands to know, or claims that we didn't something we shouldn't have, or for invalid reasons
- Capture everything necessary to identify exactly what was delivered and repeat the steps necessary to reproduce it again
- Report status of our activities to the customer and to management for the features and fixes that we are currently working on, so they can evaluate the progress and risks in delivering the project
- Retrace our steps regarding something we did poorly, or that we simply didn't know to pay attention to before, so we can identify root causes and learn how to prevent and/or correct it in the future

Those are all some pretty good reasons. Does that really justify why some of us have to

We use cookies to ensure you get the best experience on our website.

straightforward when one is doing **test-driven development (TDD)**. When it comes to tracing requirements through to design and code, images of manually maintained traceability matrices that are hopelessly effort-intensive and never quite up-to-date seem to spring to mind.

The Seven Functions of SCM
To revisit why any of those reasons are valid, it may help to revisit just exactly what Software Configuration Management is and what functions it fulfills. Most standard definitions of CM will cite the basic four elements of hardware and software CM as defined by IEEE and military/government standards:

- Identification: identifying and defining the structure of and relationships between the items or components that we must build
- Control: controlling the release of a product and changes to it throughout the lifecycle
- Status Accounting: recording and reporting the status of components and component changes
- Audit and Review: verifying the correctness and completeness of a product delivered and the consistency of its components

More recently, **SWEBOK** adds process management, and build & release management. Susan Dart gives what is probably the **definitive description of the functions of Software CM and what it adds to traditional CM**. She cites the basic four functions of CM from early standards, and then goes on to add:

- Manufacturing: managing the construction and integration/assembly of product builds and releases for one or more architecture/technology platforms, production environments, test environments, and market/customer variants
- Process Management: ensuring the correct execution of the organization's procedures, policies, and life-cycle model
- Team Work: controlling the work and interactions between multiple developers/engineers and multiple teams collaborating and coordinate their efforts to develop, test and deploy the product

That tells us about what services SCM should be providing, but what does it tell us about who is served by it, and how?**The Six Facets of Traceability**
So what are the main stakeholder needs and views that traceability supposedly serves in performing CM and delivering value to the customers and the business? Based on sources like **CMMI**, **SWEBOK**, and several others, I think the difference facets of traceability are as follows:

- **Change Dependency Analysis**: assess the impact and risk of a proposed change to facilitate communication, coordination and estimation *{show the "know-how" to "know-what" to change, "know-where" to change it, and "know-who" will change it}*
- **Product Conformance**: assure that necessary and sufficient requirements were implemented, and ensure the implementation of each requirement was verified/validated *{prove we "did the right things" and "did the things right"}*

We use cookies to ensure you get the best experience on our website.

and ensure they were executed satisfactorily *{prove we "walk the walk" and not just "talk the talk"}*
- **Project Accountability**: assure that each change of each artifact was authorized and ensure that they correspond to requested functionality/business-value *{safeguard against "gold plating"}*
- **Baseline Reproducibility**: assure that the elements necessary to reproduce each baseline have been captured and ensure that the baselines can be reproduced *{so "all the king's horses and all the king's men" can put your fallen "humpty-dumpty" build back together again}*
- **Organizational Learning**: assure that the elements necessary to rediscover the knowledge of the system have been captured and ensure that the rationale behind critical decisions can be reproduced—e.g., for root-cause analysis, or to transfer system knowledge to a deployment/support team. *{"know-why" you did what you did when you did it}*

I limited myself to "six" because I view these six as indicative of the canonical set of stakeholder views/dimensions of CM: product, process, project, organization, environment, and evolution (change).

The Five Orders of Traceability

Thus far, we've covered the basics of traceability regarding what it does, why it's useful, why it's painful, what functions it serves, and which stakeholders it serves. Now it's time to delve deeper into what is really happening with traceability, and how much and how deep we really should be attempting to trace.

Phil Armour, in his magnificent book **The Laws of Software Process**, maintains that software is not a "product" in the usual production-oriented sense of the word, but that software is really a medium for capturing executable knowledge. He then uses this to derive that software is therefore not a product-producing activity but rather a knowledge creating and knowledge acquiring activity.

Couched in these terms, traceability is supposed to help us track and link related pieces of knowledge as we progress through the software development lifecycle. The software development lifecycle is devoted to creating knowledge that transforms fuzzy functional concepts into tangible working results. Software development then becomes the process of transforming theoretical concepts into executable reality.

Armour goes on to describe **"The Five Orders of Ignorance"** and how, if software development is a process of knowledge-acquisition and knowledge-creation, then it is also ultimately a process of "ignorance reduction" whereby we progressively reduce our ignorance of what the system is, what it needs to do, how it needs to do it, and how we need to do it and manage it. These five orders of ignorance are:

- **0th Order Ignorance (0OI)** – Lack of Ignorance: I know something
- **1st Order Ignorance (1OI)** – Lack of Knowledge: I don't know something
- **2nd Order Ignorance (2OI)** – Lack of Awareness: I don't know that I don't know something

We use cookies to ensure you get the best experience on our website.

- **4th Order Ignorance (40I)** – Meta Ignorance: I don't know about the Five Orders of Ignorance

Perhaps it should follow then, as a corollary to all of this, that *the goal of traceability* is to connect all the dots between these various pieces and "orders" of knowledge to help those of us auditing or reviewing them to learn the lessons of how and why that knowledge came to be created in its current form.

Traceability attempts to reveal knowledge that is otherwise opaque or translucent, and make it transparent so that it is visible to all stakeholders concerned.

To that end, I hereby propose **The Five Orders of Traceability**:

- **0th Order Traceability** – Existence: Tracking Knowledge Content
- **1st Order Traceability** – Structure: Tracking Knowledge Composition
- **2nd Order Traceability** – History: Tracking Knowledge Context
- **3rd Order Traceability** – Transformation: Tracking Knowledge Creation (causal connection in the knowledge creation/derivation process)
- **4th Order Traceability** – Meta-Traceability: Tracking Knowledge of the Five Orders of Traceability.

It's not entirely clear to me whether I have the 2nd and 3rd items mixed-up in the above (perhaps structure should come after context). When deriving the ordering and identification of the above, I basically took my cue from Armour's 5 orders of ignorance, and made the assumption that it is the intent of a particular "order" of traceability to eliminate or address the corresponding "order" of ignorance! Hence 3rd order traceability should help resolve 3rd order ignorance, 2nd order traceability should help resolved 2nd order ignorance, and so on.

With that in mind, I'll elaborate further upon each of the five orders of traceability I've "coined" above ….

0th Order Traceability is merely the existence of knowledge content. There are no additional linkages or associations to navigate through that content. There is no explicit traceability; the content is simply there.

1st Order Traceability is an attempt to structurally organize the knowledge content and provide links that navigate the decomposition from one level of organization/detail to another. This would be like an outline structure and cross-referencing/indexing capabilities. A number of tools give us a means of organizing a particular portion of system knowledge:

We use cookies to ensure you get the best experience on our website.

- Modeling tools provide a way of organizing and viewing problem-domain and solution-domain abstractions
- Many interactive development environments give a logical (e.g., object-based) view of the code and/or a physical (e.g., file-based) view of the file and folder structure of the codebase.
- Many of these tools even provide a way to link from a section of a document to a model entity (e.g., a UML object or package) and/or a code-construct (e.g., a class, method, or module)

2nd Order Traceability goes beyond mere content and structure to provide contextual awareness. Not only are there links, but there is also contextual information (typically in the form of metadata) giving a clue as to the events that transpired to create it: who authored the content, when they did it, where they did it (physically or virtually)

This type of historical/log information assists in auditing and recovery, and is typically most-suited for automatic capture/recording by the application used to create/modify and store the information (perhaps using a mechanism like **dynamic event-based traceability**). The information might also include application integration data (for example, recording the identifier of a requirement or a change-request and associating it with the corresponding portion of the design, code, or tests when it is created or updated).

3rd Order Traceability is the "nirvana" of traceability for many. Not only do we have structure and context, but we have additional associations and attributes (e.g., metalinks between [meta]data) that capture the causal connection between related pieces of knowledge in the value chain. Some call this "**rich traceability**" and there are some high-end requirements management tools capable of doing this.

Still, tracing all the way through to design models, and code remains very effort-intensive unless all knowledge artifacts are captured in the same repository (requirements, models, code, tests, project-activities, change-requests) where these advanced "rich tracing" capabilities exist.

With 3rd order traceability, we are effectively capturing important decisions, criteria, constraints, and rationale at the various points in the knowledge creation lifecycle where one form of knowledge (e.g., prose, model, code) is being either transformed into another form of knowledge, or is being elaborated to another level of decomposition within the same form of knowledge.

4th Order Traceability is meta-traceability, or tracking of knowledge about the five orders of traceability within or across systems. (Sorry—I couldn't come up with anything better that is analogous to Armour's 5th order of ignorance— a.k.a. meta-ignorance. If you have

We use cookies to ensure you get the best experience on our website.

These five “orders” gives some more fundamental insight into the depth and breadth of knowledge that needs to be traceably structured and visibly organized to transparently communicate to the system’s stakeholders. What about the depth and breadth of the stakeholders we need to communicate with and the communication structures into which they are organized?

The Four Rings of Stakeholder Visibility

In the Feb 2005 ObjectWatch Newsletter, Roger Sessions describes the *Rings of the Enterprise* for service-oriented enterprise architectures as:

- **Ring 0:** Individual Applications (and Projects/Teams)
- **Ring 1:** Enterprise Systems (and IT Programs/Portfolios)
- **Ring 2:** Collaborative Partners
- **Ring 3:** Everybody Else

Sessions is author of the book **Software Fortresses: Modeling Enterprise Architectures**. In the software fortresses model:

Enterprise architecture is viewed as a series of self-contained, mutually suspicious, marginally cooperating software fortresses (each with their own “guards” and “walls”) interacting with each other through “drawbridges” carefully crafted and meticulously managed via treaty or “trust” relationships.

Each one of these rings of enterprise-scope corresponds to a boundary between stakeholders that must be bridged with *communication* and **trust**. *Every development team and project manager must build bridges across these fortresses by successfully managing the expectations and interfaces between them*! And traceability is often used as a means of enabling transparent communications through these rings, across the drawbridges and into the fortresses.

Great! Now we know more about the 4 levels of scale/diversity of our stakeholders, the 5 orders of information to track, the set of 6 stakeholder views and concerns to target, the 7 functions of SCM to serve, 8 good reasons for, 9 common complaints against, and 10 commandments of traceability. That still doesn’t tell us what makes it so darn painful!

The Three Driving Forces for Traceability

If trust is the motivating goal, transparency is the means to achieve it, and traceability is a mechanism to provide transparency, what is it that makes traceability a necessary mechanism rather than simpler alternatives that might yield transparency without all the pain of traceability?

If everything could just be simple, we’d need only one team, have only one target audience, immediate visibility or status/results, and easily understood information in small batches. The reality is that things often aren’t that simple. Issues of scale and diversity rear their ugly head along all these different dimensions (SCM functions, stakeholder facets, communication rings, etc.). As projects, systems, and organizations get larger and more diverse, the following issues always seem to rear their ugly head:

We use cookies to ensure you get the best experience on our website.

"goings on" of the project, the less likely I am to trust it. People often do not readily trust what they cannot readily see
- **Feedback Cycle-time**—the more time that passes in between the time that I ask for something (or from when I know the work starts) and the time to when I receive it or hear about its status, the less likely I am to trust it
- **Knowledge Complexity**—the more complex something is, the less confidence I have that it does what I expect

All of these impede the ability to efficiently and accurately communicate and respond to change. And they reduce the visibility, transparency, and frequency of available information. It becomes harder to see what we care about, and harder to locate those concerns among what limited information we can see.**Two Overarching Objectives—Transparency & Identification**

That's what traceability is really trying to accomplish in a nutshell:

- *Transparency:* the ability to readily view *all* the information we are concerned with, and then …
- *Identification*: the ability to identify our concerns so we can separate independent sets of concerns and cohesively associate the related ones

Traceability aims to provide transparency by tracing the linkages through the complex quagmire of requirements through all intermediate artifacts and finally running and tested code. Transparency is what ultimately helps us resolve the issue of visibility issue, and the information structure is what organizes and identifies the set of concerns to track and navigate:

- Architectural (structural) transparency helps us identify and analyze change-impact
- Functional (behavioral) transparency helps us identify and analyze product conformance
- Process (procedural) transparency helps us identify and analyze process compliance
- Project (managerial) transparency helps us identify and analyze project accountability
- Build/Baseline (physical) transparency helps us identify and analyze reproducibility
- Decision-making (logical) transparency facilitates organizational learning and root-cause analysis

The Role of Tools

Executing traceability reports surfaces the invisible, opaque and translucent and presents them to us while letting us select, sort, and filter on the entities vital to our role or perspective: requirements, designs, code, tests, requests, changes, projects, configurations, etc.

Tools play a crucial role in achieving traceability and transparency. Tools help us both create and maintain the drudgery of complex information, structure, and linkages, and

We use cookies to ensure you get the best experience on our website.

One Ultimate Mission—Fostering a Community of Trust

Economists talk about measuring the level of trust in a society. It's been proven that societies where there is greater trust experience faster economic growth—"My word is my bond." The same is true in software. To have trust, you must not be afraid that someone is hiding something from you. Transparency enhances trust. All of the things we have discussed so far distill down to this: they help convey to us (the stakeholders) a greater confidence in our knowledge of what is going on in the project, and whether or not it will successfully deliver what we need, when we need it. This let's us trust the people managing and performing the work.

All of the traceability questions we asked and the reasons we gave all boil down to questions of trust, and communication:

- Do I trust the analysts/architects to do what I said and say what they did?
- Do I trust the product or its producers to correctly realize the correct requirements?
- Do I trust the process or the engineers following it?
- Do I trust the project or the people managing it?
- Do I trust the environment in which it is built and tested?
- Do I trust the organization to be able to remember what they learned and learn from their mistakes?

Traceability, Transparency, and Trust

Whether or not traceability efforts successfully achieve any of these goals is another matter entirely. Often, traceability efforts achieve at most one of these (conformance), and sometimes not even that. Many question whether the amount of effort required actually adds more value than what it subtracts in terms of effort (particularly when traceability efforts themselves may create additional artifacts and activities that require tracing).

Traceability is often desired because it's presumed to somehow alleviate fears and give us "TRUST-ability." I suspect that's really an illusion. Traceability is useful to the extent that it facilitates more effective communication & interaction between individuals, teams, and stakeholders. And traceability can be useful to the extent that its helps us successfully convey, codify and validate system knowledge. So to the extent that traceability helps us more quickly identify the right people to interact with and the right information to initiate that interaction, it can be extremely useful. It is the people and the quality of their interactions that provide the desired "trustability." Beyond that, unless unobtrusively automated, it can quickly turn into "busy work" with very high "friction" on development of new functionality realized as "working software. "Trustability ultimately has to come from "people." And while transparency engenders trust, traceability is

We use cookies to ensure you get the best experience on our website.

We need to move the industry to a place where software development is not an opaque cloud where magic happens (maybe). Management can believe that all reports are lies or hiding the truth or they can know that reports are automatically generated based on transparent traceability data. This leads them to trust the information in the reports and to trust the teams generating them. It is this principle that was at the heart of FDD knowledge base systems and is a major goal for Microsoft's new Visual Studio Team System. Communication is increased by trust, but also generates other requirements. Managers usually can't understand code or other technical artifacts. They rely on documentation and on the results of complete system testing. To non-engineers the complexity of adding non-code artifacts increases transparency whereas only having working code looks quite opaque.

Many have had to create "design documents" after the code was in production (or at least had passed QA) because management considered the code incomplete or untrustworthy without documentation. Sometimes this was driven by the legitimate concern for maintainability, but sometimes managers want documents to show that code that is already in production actually does what it is supposed to do. Thus a cause of the concern for traceability goes to the differences between managers who want documents that they understand and engineers who want code that works. Perhaps a similar difference can be found with sales and accounting departments. A salesman knows that new sales are what keep the company going, while the accounting department has to provide the traceability that the IRS and investors demand.

To have transparency, you must drive out fear. Many people believe that data hiding is required because figures can be misinterpreted. Hence, they hide information from people whom they believe will draw the wrong conclusions. It would seem preferable to educate people to draw the right conclusions, making information sharing (or transparency) an enabler to team power.

Conclusions

So there you have it: **traceability provides transparency, and transparency engenders trust!**

When we do regular builds and other regular development activities and publish build-status reports and burndown-charts and cumulative flow diagrams in a visible fashion, we are giving agile stakeholders a form of RFID for each task, feature, build, iteration and release.

This type of frequent and regular feedback gives transparency of an agile development team's activities to groups like CM, V&V/QA, Systems Engineering, and Program

We use cookies to ensure you get the best experience on our website.

When we revisit this subject in a future column, we will discuss whether there might be a better and simpler way than the manual matrices many mundanely maintain. We'll consider how we can apply simplicity to improve transparency while creating less to trace, and how basic design principles to minimize dependencies and separate concerns can remove complexity and reduce the traceability burden.

Acknowledgements

David J. Anderson, Ron Jeffries, Michael Beedle, Scott Ambler, Steven Gordon, Dean W. Schulze, Paul Oldfield, Pete C. Ruth, Paul Tiseo and Mark Graybill for their contributions to the lively and informative discussion about traceability on numerous agile-related forums over the past two years.

Topics: configuration management, requirements, requirements management, test techniques,

We use cookies to ensure you get the best experience on our website.

ABOUT THE AUTHOR

Steve Berczuk

Steve Berczuk is a Principal Software Engineer with experience as a manager, Scrum Master and technical lead in Boston, MA. The author of *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*, he is a recognized expert in software configuration management and agile software development. Steve is passionate about helping teams work effectively to produce quality software. He has an M.S. in operations research from Stanford University and an S.B. in Electrical Engineering from MIT, and is a Certified ScrumMaster. Contact Steve at steve@berczuk.com or visit berczuk.com and follow his blog at blog.berczuk.com.

Brad Appleton

Brad Appleton is an Enterprise Agile+DevOps leader, coach & manager, and seasoned DevOps/ALM/CM solution architect at a large Fortune 100 company. Currently he helps organizations and teams scale, adopt and apply lean/agile development methods and DevOps/ALM/CM practices and tools. He is co-author of *Software Configuration Management Patterns*, the Streamed-Lines branching patterns, and a former section editor for *The C++ Report*. You can read Brad's blog at blog.bradapp.net.

Robert Cowham

Robert Cowham has long been interested in software configuration management while retaining the attitude of a generalist with experience and skills in many aspects of software development. A regular presenter at conferences, he authored the Agile SCM column within the CM Journal together with Brad Appleton and Steve Berczuk. His day job is as Services Director for Square Mile Systems whose main focus is on skills and techniques for infrastructure configuration management and DCIM (Data Center Infrastructure Management) - applying configuration management principles to hardware documentation and implementation as well as mapping ITIL services to the underlying layers.

UPCOMING WEBINAR

We use cookies to ensure you get the best experience on our website.

LETS HANG!

FEATURED RESOURCES

SECURITY
Shifting Security Left: DevSecOps eGuide
White Paper | StickyMinds

Log In or Register to post comments

USER COMMENTS

0 comments

YOU MAY ALSO LIKE

SECURITY
Agentic AI Opens Up New Security Flaws
Nahla Davies | August 27, 2025

We use cookies to ensure you get the best experience on our website.

SECURITY

Countering the Threat of Ransomware-as-a-Service (RaaS)

Nahla Davies | August 22, 2025

SECURITY

Building Trust in Blockchain: The Role of Smart Contract Testing

Ekaterina Egorova | November 18, 2024

Join the crew for FREE! Access exclusive content, contribute, and more.

See All Member Benefits

We use cookies to ensure you get the best experience on our website.