

The Role of Requirements Traceability in System Development

by [Dean Leffingwell](#)

Software Entrepreneur
and Former Rational Software Executive

[Don Widrig](#)

Independent Technical Writer and
Consultant

In the field of software development, there is never a shortage of debates regarding the relative merits of methods, practices, and specific techniques for analysis, design, and coding activities. Not far from the top of the "most controversial" list is the topic of requirements traceability and its role in defining and implementing systems. Although the arguments may well go on for decades, and tools will assuredly evolve to support both more and less artifact traceability, there can be little doubt about one thing: Requirements traceability has been shown to be an effective technique in some development practices, particularly for those classes of systems for which software failure is not an option.

In these cases, the question becomes not if, but how, to implement traceability. For commercial practices, in which defects, or even application failures do not bring about catastrophic results, the topic is more hotly debated, and the legitimate question becomes, not how, but if. In those cases, the discussion rightfully focuses on what value -- and at what cost -- requirements traceability might bring to the project. In this article, we'll provide an overview of the whys and hows of requirements traceability, and attempt to describe a basic, generic, and practical approach that can be used in a wide variety of system development practices. For now at least, we'll leave the debates on when and how much to others!

[▶ subscribe](#)[▶ contact us](#)[▶ submit an article](#)[▶ rational.com](#)[▶ issue contents](#)[▶ archives](#)[▶ mission statement](#)[▶ editorial staff](#)

Introduction to Requirements Traceability

Experience has shown that the ability to trace requirements artifacts through the stages of specification, architecture, design, implementation, and testing is a significant factor in assuring a quality software implementation. The ability to track these relationships and analyze the impact when change occurs is common to many modern, high-assurance software processes -- particularly in the life-critical medical products area and in business- or mission-critical activities.

Historical safety data has demonstrated that projects often miss or do not address requirements and/or the impact of change, and that small changes to a system can create significant safety and reliability problems. This has caused some regulatory agencies to mandate that traceability be an integral part of the development process. For example, the latest U.S. Food and Drug Administration (FDA) guidance for traceability in medical software development activities is contained in the FDA Office of Device Evaluations (ODE) Guidance Document (1996b). In addition, the Design Controls section of the medical Current Good Manufacturing Practices (CGMP) document (FDA 1996a), Subpart C of the CGMP, defines the obligations of the system designers to be able to trace relationships between various work products within the lifecycle of the product's development.

If traceability has been mandated for your project by regulatory or internal process standards, then your team won't have any choice. If it has not been mandated, then your team will need to decide whether to apply implicit and explicit requirements traceability or not. If they decide to apply it, they will also need to determine how much traceability is required to assure the outcome you need. We'll revisit this discussion again in just a bit, but first let's look at what traceability is and how to apply it.

For a starting point, IEEE (1994) provides a compound definition of traceability:

"The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match." (IEEE 610.12-1990 §3)

"The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement it satisfies." (IEEE 610.12-1990 §3)

The Traceability Relationship

In other words, a traceability relationship is a relationship between two project elements. The first definition above is about relating one element in the process to a successive element; the second is intended to assure that there are no superfluous elements (requirements, design, etc.). In

order to relate these elements, we need to introduce a third element that relates the first two elements to each other. We call this element the "traceability relationship." There is no standard for exactly how this element is constructed, how it is represented, what rules apply to it, and what attributes it exhibits. In highly complex, high-assurance systems developed under extreme software process rigor, this relationship can take on many facets. It might be navigable via tooling, it might contain a history of its existence. There might be many different types of such relationships, such as "is fulfilled by," "is part of," "is derived from," etc., and there might be different types of relationship based upon the types of elements that are being related (for example, an explicit "tested by" relationship between A and B means that B exists to test A). If your team is operating in one of these environments, you will likely use process definitions, special tooling, and specific procedures to support traceability.

For most project teams, however, this overcomplicates the matter. It is usually adequate to think of the relationship in terms of a simple "traced-to" and "traced-from" model. In UML terms, we would simply be looking in one direction or the other (upstream or downstream) at a *dependency* relationship (Figure 1).¹

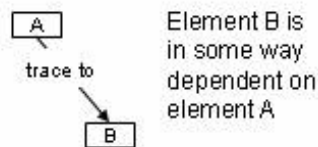


Figure 1: A Simple Dependency Relationship

A dependency relationship states that a change in one element (for example, a use case) might affect another element (such as a test case), but the reverse is not necessarily true (a change to the test case would not necessarily imply that a use case needs to be changed). In other words, what happens upstream (a boat placed in the water) affects what happens downstream (the boat goes over the waterfall), but the opposite case is not true (no matter how many boats we place in the water downstream, the upstream system behavior is unaffected).

As another example, we can envision how a specific requirement in the system is created in order to support a given feature specified in the Vision document.² Thus, we can say that a software requirement (or use case) is traced from one or more features (see Figure 2). In this manner, the traceability relationship fulfills both halves of the definition above: It both describes the predecessor-successor relationship and establishes the reason the second element exists (to implement the feature).

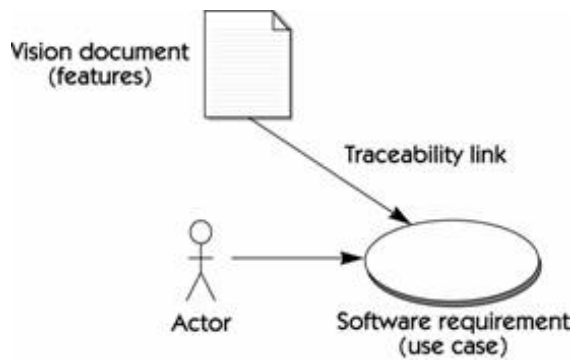


Figure 2: Traceability Link from Vision Document to Software Requirement

Without complicating matters much further, we can infer additional meaning by looking at the context of the types of requirements that are being related. For example, even without a specific "tested by" relationship type, a software requirement that is *traced to* a test case would suggest that the software requirement is "tested by" the test case that it is "traced to." A use-case realization that is *traced from* a use case would imply that the requirement is "implemented by" the referenced collaboration.

A Generalized Traceability Model

As we have learned, different projects drive different types of requirements artifacts and different ways of organizing them. These decisions, in turn, drive differing needs as to the number and nature of the artifacts to be traced. However, building on what we've learned, we can see that there is a static structure, or model, for a traceability strategy that is common to most projects; this hierarchical model goes from higher level requirements through more detailed requirements, and then on to implementation and testing. Therefore, no matter what type of project your team is executing, your traceability model will likely appear like the one in Figure 3.

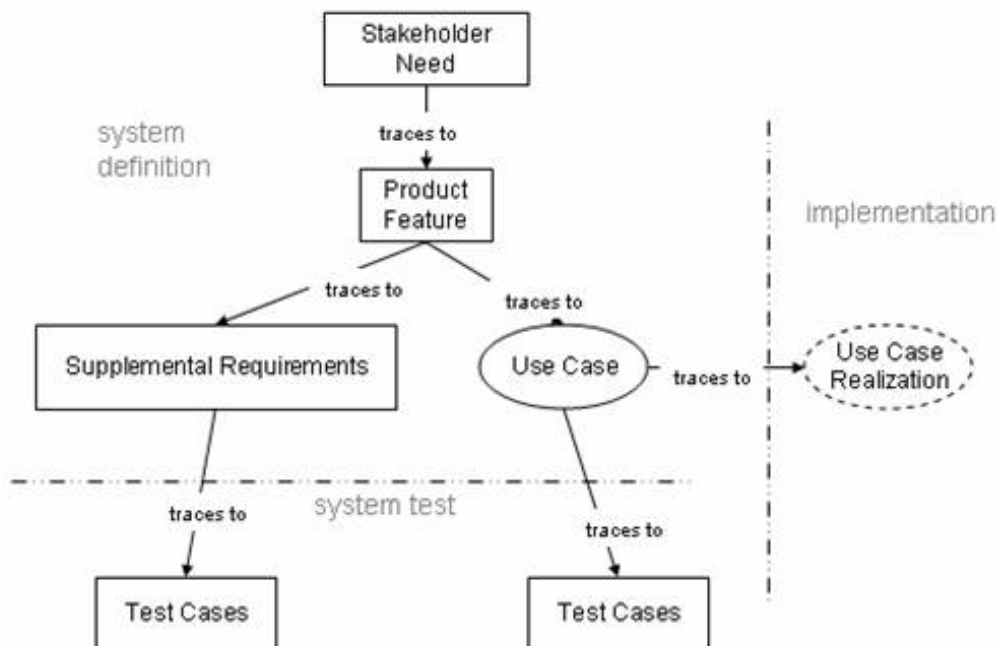


Figure 3: Generalized Traceability Hierarchy

This model shows that we are tracing requirements both within a domain, as in the system definition (or requirements) domain, and from there, into the implementation and test domains. Although there are many additional "things" that you could trace (requirements to glossary items, etc.), experience has shown that these basic types of traces usually cover most needs. Of course, your conditions might differ, and it might be necessary to add or subtract from the core traceability list.

In the following sections, we'll look at examples based on this model and explain why you would want to do such tracing.

Tracing Requirements in the System Definition Domain

Let's look first at tracing requirements within the system, or product, definition domain. We'll call this "requirement to requirement" traceability because it relates one type of requirement (e.g., feature) to another (e.g., use case).

Tracing User Needs to Product Features

Generally, the point of developing a system of any kind is to satisfy some set of user and other stakeholder needs. Therefore, the time spent understanding user needs is some of the most valuable time you can spend on the project. Defining the *features* of a system that meets those needs is the next step in the process. While performing this step, it can be helpful to continually relate the features of your proposed solution back to user needs. You can do so via a simple table, or *traceability matrix*, similar to the one shown in Table 1.

Table 1: Traceability Matrix -- User Needs to System Features

	Feature 1	Feature 2	...	Feature n
Need #1	X			
Need #2		X		X
Need ...		X	X	
Need #m				X

In Table 1, we've listed all the user needs that we have identified down the left column. In the row across the top, we've listed all the system or application features that were defined to satisfy the stated needs.

Once the rows (needs) and columns (features that have been defined to address those needs) are defined, we simply put an "X" in the cell(s) to record the fact that a specific feature has been defined for the sole purpose of supporting one or more user needs. We also note that, typically, this is a "one to many" mapping, as there are typically far fewer needs identified, and they are specified at higher levels of abstraction,

than the number of system features that are defined to implement those needs.

After you've recorded all known need-feature relationships, examining the traceability matrix for potential indications of error can be an instructive activity:

1. If there are no Xs in a *row*, it's possible that no feature is yet defined to respond to a user need. This might be acceptable if, for example, the feature is not fulfilled by software ("The case shall be of non-breakable plastic"). Nevertheless, these are potential red flags and should be checked carefully. Modern requirements management tools will have a facility to automate this type of inspection.
2. If there are no Xs in a *column*, it's possible that a feature has been included for which there is no known product need. This might indicate a gratuitous feature or a misunderstanding of the feature's role. Or it might indicate a dead feature that is still in the system but whose reason to exist has disappeared, or at least is no longer clear. Again, modern requirements management tools should facilitate this type of review, and in any case, you are not dealing with a great deal of data at this level.

In addition, the dependency relationship inherent in the traceability relationship allows you to see what specific needs would have to be reconsidered if a user requirement should change during the implementation period. Hopefully, this *impact assessment process* will be supported by the automated change detection capabilities of your requirements tool.

Once you've mapped the need-feature relationships and have determined that the needs and features are correctly accounted for and understood, it's time to consider the next level of the hierarchy: relationships between the features and the use cases.

Tracing Features to Use Cases

Just as important as tracing user needs to system features is ensuring that the system features can be related to the system's use cases. After all, it's the use cases that illuminate the proposed implementation of the system from a user's perspective, and our job is to ensure that we have a fully responsive design.

As before, we can consider a simple matrix similar to the one shown in Table 2:

Table 2: System Features versus Use Cases

	Use case 1	Use case 2	...	Use case n
Feature #1	X			X
Feature #2		X		X
Feature ...			X	
Feature #m		X		X

In Table 2, we've listed all the system features down the left column. In the row across the top, we've listed the use cases that were derived to implement the stated features.

Once the rows (use cases) and columns (features) are defined, we indicate a traceability relationship with an (X) in the cell(s) representing a use case that supports one or more features. Note that this is also likely to be a set of "many to many" relationships; although both features and use cases describe system behaviors, they do so via different means and at differing levels of detail. A single feature will often be supported or implemented by multiple use cases. And, although less likely, it might also be true that a single use case implements more than one feature.

After you've established all known feature-to-use-case relationships, you should once again examine the traceability matrix for potential indications of error:

1. If there are no Xs in a *row*, it's possible that no use case is yet defined to respond to a feature. As before, these are potential red flags and should be checked carefully
2. If there are no Xs in a *column*, it's possible a use case has been included that no known feature requires. This might indicate a gratuitous use case, a misunderstanding of the use case's role, or a use case that exists solely to support other use cases. It might also indicate a dead or obsolete use case.

In any case, reviewing and analyzing the data will improve your understanding of the implementation, help you find the obvious errors, and increase the level of surety in the design and implementation.

Once you've mapped the feature-use-case relationships and have determined that the features and use cases are correctly accounted for, you need to apply similar thinking to the matter of nonfunctional requirements and their specification.

Tracing Features to Supplementary Requirements

Although the use cases carry the majority of the definition of the system's functional behavior, we must keep in mind that the supplementary requirements³ also hold valuable system behavioral requirements. These often include the nonfunctional requirements of the system such as usability, reliability, supportability, and so on. Although there might be fewer of these requirements (or more of them), their criticality can be as great or greater (example: results must be within an accuracy of $\pm 1\%$)

than that of the use cases themselves. In addition, certain functional requirements (e.g., those that are algorithmic or scientific in nature such as a language parsing program) will likely not be expressed in use-case form.

Again, we look to the Vision document for these features, or additional high-level requirements, that drive supplementary specifications and trace them from there into the supplementary requirements that capture this important information. This can be captured in a matrix similar to Table 3.

Table 3: System Features versus Supplementary Requirements

	Supplementary Req 1	Supplementary Req 2	...	Supplementary Req n
Feature or System Requirement #1	X			X
Feature or System Requirement #2		X		X
Feature or System Requirement #m		X		X

Tracing Requirements to Implementation

Having described the type of requirements tracing typical in the system definition domain (requirement-to-requirement traceability), we are now prepared to move from the requirements domain into the implementation and test domains. Although the principles are the same -- we use the dependency traceability relationship to navigate across this chasm -- the information content on the other side is remarkably different. Let's look first at crossing from requirements to implementation.

Tracing Use Cases to Use-Case Realizations

In making this transition, we move to relating one form of artifact -- the use case, which is a requirement form -- to another artifact: the *use-case realization*⁴ in the design model. In so doing, we use these two specific elements to bridge the gap between requirements and design, as shown in Figure 4:

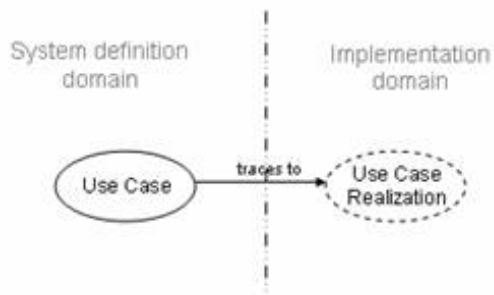


Figure 4: Tracing from Requirements to Implementation

This simplifies the traceability problem immensely *because there is a one-to-one name space correspondence between a use case and its realization.* Therein we meet both traceability requirements: The relationship between the entities is expressed directly by their name sharing, and the reason for the existence of the subordinate or "traced to" entity, the use-case realization, is implicit in its very nature. That is, the use-case realization exists for only one purpose: to implement the use case by the same name. Therefore, there is no matrix that requires analysis, as the design practice we employed yielded inherent traceability by default!

Tracing from the Use-Case Realization into Implementation

For those who require a higher degree of assurance, or when traceability to code is mandated, it might not be adequate to stop at the logical construct of the use-case realization. In such a case, the traceability relationship must be followed from the use-case realization to its component parts, which are the classes (code) that implement the collaboration (see Figure 5).

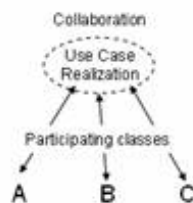


Figure 5: Tracing from the Use-Case Realization to Classes

How you will accomplish this mechanically will depend on the types of tools that you employ in your requirements, analysis, and design efforts. Without adequate tooling, the problem will quickly become intractable, as you will likely be dealing with hundreds of use cases and thousands of classes.

Tracing Supplementary Requirements into Implementation

Of course, many requirements will not be expressed in use-case form, and for anyone who must drive to uncompromised degrees of quality and safety, it will be necessary to trace from supplementary requirements into implementation as well. In this case, we trace individual requirements, or groups of requirements, to a collaboration in the implementation. After all, the use-case realization wasn't really so special; it was just a type of

collaboration all along. But in this case, we'll have to name the collaboration and keep track of the links via some special means because they don't come pre-named and conveniently collected in a use-case package. However, the principle is the same, as Figure 6 illustrates.

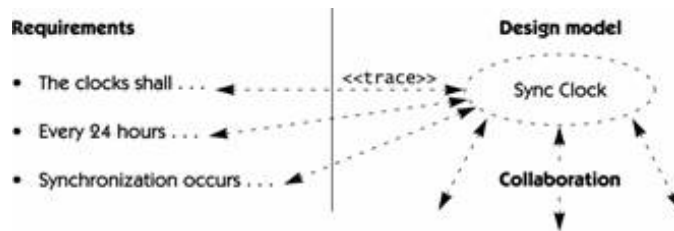


Figure 6: Tracing from Supplementary Requirements to Implementation

From here, we can trace to the specific code contained in the classes that realize the collaboration. Again, the mechanics of this will be determined by the types of tooling we choose to employ.

Tracing from Requirements to Test

And finally, we approach the last system boundary that we must bridge to implement a complete traceability strategy: the bridge from *the requirements domain to the testing domain*.

Tracing from Use Case to Test Case

As Jim Heumann described in his May 2001 *Rational Edge* article,⁵ one specific approach to comprehensive testing is to ensure that every use case is "tested by" one or more test cases. This was already reflected in a portion of our generalized traceability model, highlighted in Figure 7.

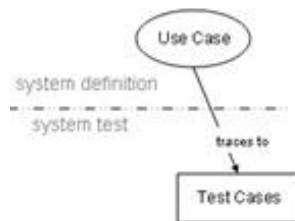


Figure 7: Tracing Use Cases to Test Cases

However, this simple diagram understates the complexity of the case somewhat, for it is not a trivial 1-for-1 transition that is described.

First, we have to identify all the scenarios described in the use case itself. This is a one-to-many relationship, as an elaborated use case will typically have a variety of possible scenarios that can be tested. From a traceability viewpoint, each use case "traces to" each scenario of the use case as shown in Figure 8.

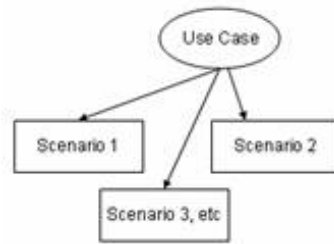


Figure 8: Tracing Use Cases to Test Case Scenarios

In matrix form, one row represents each scenario of a specific use case, as Table 4 illustrates.

Table 4: Traceability Matrix for Use Cases to Scenarios

Use Case	Scenario Number	Originating Flow	Alternate Flow	Next Alternate	Next Alternate
Use Case Name #1	1	Basic Flow			
	2	Basic Flow	Alternate Flow 1		
	3	Basic Flow	Alternate Flow 1	Alternate Flow 2	
	4	Basic Flow	Alternate Flow 3		
	5	Basic Flow	Alternate Flow 3	Alternate Flow 1	
	6	Basic Flow	Alternate Flow 3	Alternate Flow 1	Alternate Flow 2
	7	Basic Flow	Alternate Flow 4		
	8	Basic Flow	Alternate Flow 3	Alternate Flow 4	
Use Case Name #2	1	Basic Flow			

However, we are still not done, because each scenario can drive one or more specific test cases, as illustrated in Figure 9.

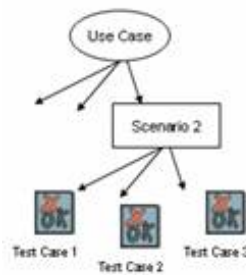


Figure 9: Tracing Scenarios to Test Cases

However, in matrix form, this simply adds one more column, as illustrated in Table 5 (we've deleted the scenario descriptions in this version).

Table 5: Traceability Matrix for Use Cases to Test Cases

Use Case	Scenario Number	...	Test Case Id	
Use Case #1	1		1.1	
	2		2.1	
	3		3.1	
	4		4.1	
	4		4.2	
	4		4.3	
	5		5.1	
	6		6.1	
	7		7.1	
	7		7.2	
	8		8.1	
	Use Case #2	1		1.1

In this way, a traceability matrix of one-to-many (use case to scenario) and an additional one-to-many (scenario to test case) relationship can fully describe the relationship among these elements. In a manner similar to the other matrices, automated tooling can help you build this matrix as well as perform certain automated inspections and quality tests. In addition, some tools will provide immediate impact assessment by indicating which "traced to" elements (e.g., test cases) might be affected, when a "traced from" element (scenario) is changed.

Tracing from Supplementary Requirements to Test Cases

For those requirements that are not expressed in use-case form, the process is similar to the requirements-to-implementation process described above. More specifically, requirements are either traced individually to scenarios and test cases, or they can be grouped into "requirements packages" that operate in the same logical fashion as a use case. The matrices we describe above are unchanged, except that the

column on the far left contains the specific requirement, or requirements package, that is being traced into implementation.

Using Traceability Tools

Powerful software development tools offer a simple user-guided procedure to "point and click" through the relationships that might exist between two elements of the lifecycle. The RequisitePro requirements management tool offered by Rational Software is an example of such a tool.

Using these tools provides you a more efficient way to take on larger projects than using the manual matrix methods discussed earlier. For example, after you have defined the relationships between the features and the software requirements on a project, you can use a tool to display a matrix version of those relationships *automatically*.

These types of tools will allow you to build the larger matrices required for more sophisticated projects and to examine the data automatically for many of the types of potential red flags we discussed earlier.

In addition, a tool can also relieve much of the aggravation of *maintaining* the matrices as changes are made.

Summary

Tracing is an important technique your team can apply in the struggle to ensure that you are designing and implementing the right system. The trick is to implement "just the right amount" of traceability in "just the right way" so that the risk-to-reward ratio benefit fits your project's circumstances. Otherwise, you may find that

- Your project suffers from excessive overhead without commensurate quality improvement.
- Or worse, you fail to deliver the requisite quality and reliability demanded by your project circumstances.

These are both in the class of "very bad things that can happen to your project," so it behooves the team to define and implement the right traceability strategy from the beginning.

References

Dean Leffingwell and Don Widrig, *Managing Software Requirements: A Unified Approach*. Addison-Wesley, 1999.

Rational Unified Process 2001. Rational Software Corporation, 2001.

Jim Heumann, "Using Use Cases to Create Test Cases." *The Rational Edge*, June 2001.

Notes

¹ In UML, the arrow points the other way (i.e., from the dependent element to the independent element), and the line is dashed, not solid. However, most requirements tools use the arrow as indicated, so we'll do the same.

² See RUP 2001.

³ See RUP 2001.

⁴ In UML terms, a use-case realization is a stereotype (type of) of a collaboration

⁵ Jim Heumann, "Using Use Cases to Create Test Cases." *The Rational Edge*, June 2001



For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided. Thank you!