

Are Your Requirements Complete?

Donald Firesmith, Software Engineering Institute, U.S.A.

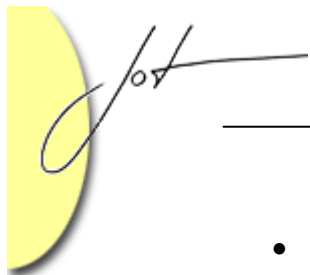
Abstract

Good requirements have several useful properties, such as being consistent, necessary, and unambiguous. Another essential characteristic that is almost always listed is that 'requirements should be complete.' But just what does completeness mean, and how should you ensure that your requirements are complete? In this column, we will begin to address these two questions by looking at (1) the importance of requirements completeness, (2) the completeness of requirements models, (3) the completeness of various types of individual requirements, , and (4) the completeness of requirements metadata. In next issue's column, we will continue by addressing (5) the completeness of requirements repositories, (6) the completeness of requirements documents derived from such repositories of requirements, (7) the completeness of sets of requirements documents, (8) the completeness of requirements baselines, and finally (9) determining how complete is complete enough when using an incremental and iterative development cycle.

1 THE IMPORTANCE OF BEING COMPLETE

The requirements engineering literature is full of books, articles, and papers that stress that good requirements share numerous characteristics such as being consistent, necessary, unambiguous, and verifiable [Firesmith 2003a] [Sommerville97]. Another of the standard characteristics is that requirements should be complete. But what exactly does it mean for requirements to be complete? What requirements work products are being referred to: individual requirements or requirements documents? Given that there is limited schedule and budget with which to perform requirements elicitation, analysis, specification, and management, just how complete should the requirements be? And given that modern development cycles mean that requirements are typically engineered in an iterative, incremental, parallel, and time-boxed manner, when should the requirements be what percentage complete? And how does completeness relate to the issue of requirements baselines? As you can see, the simple word 'complete' can have multiple meanings and raise a host of issues that should be addressed as part of a 'complete' requirements process.

Unfortunately, incomplete requirements are a very common problem for several reasons:



- It takes more effort, time, training, and will power to do the non-trivial extra work needed to make requirements complete.
- It is difficult to know just how complete the requirements should be given the limited resources (e.g., schedule and personnel) available to the requirements team.
- Subject matter experts and other sources of requirements often take certain information for granted and omit it during requirements engineering, even though it is not obvious to other stakeholders.

But before we pose some answers to these important questions, we should probably address an even more fundamental question: why is the completeness of the requirements so important?

- **Acceptance and Satisfaction.** Missing or incomplete requirements can cause customers to reject systems and users to be dissatisfied with the systems, even if they pass acceptance testing. Customers want to know that what they are buying will be adequate to meet the users' needs. This is primarily a requirements validation issue.
- **Development Cost and Schedule.** Because requirements are used to estimate system development cost and schedule, missing or incomplete requirements mean that these costs and schedules will be underestimated. One large study implicated incomplete requirements as the cause of 12.3% of cost overruns and failed projects [Standish 1994]. The primary reasons why missing or incomplete requirements cause cost and schedule overruns is that:
 - The original estimates were too low because missing requirements were not considered and
 - Costly rework was needed once these omissions were discovered.
- **Development.** Missing or incomplete requirements ensure that the architecture, design, implementation, and tests that are derived from them will be:
 - Incomplete themselves and/or
 - Incorrect because the architects, designers, implementers, or testers will make incorrect guesses or assumptions.
- **Verification.** Incomplete requirements are often ambiguous and therefore unverifiable.
- **Safety.** Perhaps the most important aspect of requirements completeness has to do with safety because over time more and more software-intensive systems have been given safety-related responsibilities. Up to 50% of all accidents are due to requirements problems and many of these accidents are due to missing or incomplete requirements [Leveson 1995]. In one analysis of 34 safety incidents, "44% had inadequate specification as their primary cause." [HSE 1995]

When speaking of the completeness of requirements, it is important to establish the scope of the term 'completeness.' There are at least seven different ways in which the phrase 'requirements completeness' could be interpreted. These include the completeness of:



1. Requirements analysis models
2. Individual requirements
3. Metadata describing individual requirements
4. Requirements repositories
5. The set of requirements documents
6. Individual requirements specification documents
7. A requirements baseline

In this column, we will begin by addressing (1) the completeness of requirements models, (2) the completeness of various types of individual requirements, and (3) the completeness of requirements metadata. In next issue's column, we will continue by addressing (4) the completeness of requirements repositories, (5) the completeness of sets of requirements documents, (6) the completeness of requirements documents derived from such repositories of requirements, and (7) the completeness of requirements baselines. We will complete this pair of columns by addressing this important issue of determining how complete is complete enough when using an incremental and iterative development cycle. Thus, this column primarily concentrates on the completeness of individual requirements, whereas the next column will primarily concentrate on the completeness of collections of requirements.

2 COMPLETE REQUIREMENTS ANALYSIS MODELS

After requirements elicitation but before requirements specification logically lies requirements analysis, which typically involves the development of various types of requirements models. Unfortunately, if these models are incomplete, then there is a high probability that the corresponding requirements and collections of requirements will also be either incomplete or missing.

Requirements models are models that document various views of the needs of the system. They can be graphical, textual, or mathematical. A **requirements model is complete** if it contains all important information needed to completely develop its associated requirements. Because different requirements models have different components, different models can have different kinds of missing or incomplete information.

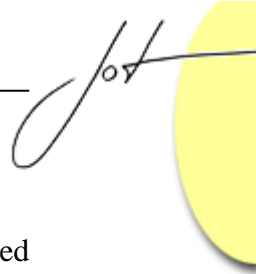
- **Context Models.** A context model documents the environment (i.e., the context) of the system including the externals that it must interact with and the necessary relationships (e.g., associations, collaborations, interactions, data flows, and control flows) between them. One or more context models can be missing if the requirements team fails to create them during requirements analysis. For example, the team may produce a top-level context model showing high-level associations but may not produce a lower-level dataflow context model even if it would be very useful and cost-effective to do so. Even if all context models are produced, individual context models can be incomplete if they are missing mandatory:
 - Externals, with which the system must interact

- Relationships between the system and its externals
- Characteristics of these relationships (e.g., dataflows not labeled with all of the mandatory data that must flow between the system and its externals)
- **Data Models.** A data model (a.k.a., information model, class model) consists of data (or classes of objects) and the important relationships between them. During requirements analysis, data models are often used as a part of domain modeling to analyze the primary (mandatory) data or classes in the application domain. As such, they provide a more powerful and detailed definition of the important concepts in the application (and nouns in the requirements) than is possible by merely using a project glossary. During requirements analysis, the requirements team may fail to create all useful and cost-effective data models at all levels of the system hierarchy. Even if an appropriate data model is produced, it can be incomplete if it is missing mandatory:
 - Data (or classes of objects)
 - Data or object types
 - Semantics (description of meaning, possibly in a data dictionary or project glossary)
 - Data or object components (e.g., attributes and aggregate parts) including their types
 - Initial or default values
 - Range of possible values
 - Units of measure¹
 - Data volume (e.g., total number of records to be stored in a database)
 - Update frequency and type (e.g., periodic, random)
 - Legitimate operations on the data (e.g., read, write, and update)
 - Relationships including:
 - Associations implying some general form of communication or dependency
 - Aggregation, membership, and containment relationships, which should not be confused with such implementation details as by reference or by pointer that are not appropriate during requirements engineering.²
 - Generalization, specialization, and/or subtyping relationships between a subtype and a supertype³ (*is a kind of* such as “Dogs are a kind of Mammal”)
 - Instantiation relationships between a mandatory object and its type (*is a* such as “Spot is a Dog”)⁴

¹ This can be a major potential source of failures and accidents such as when one part of the project uses metric units when the other part uses English units or when different parts of the project uses different metric units (e.g., meters vs. centimeters).

² Care must be taken when using UML during requirements engineering to avoid inadvertently including (and later specifying) implementation details.

³ Inheritance involves implementation details and is therefore not appropriate during requirements engineering. The preceding warning regarding the use of UML applies.



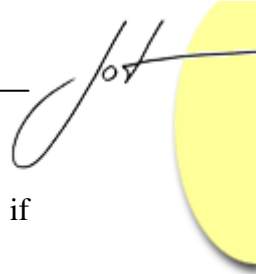
- Invariant conditions that hold among the data
- **Decision Models.** A decision model consists of a cohesive set of interrelated decisions and their consequences. Decision models are typically analyzed and documented using decision trees or decision tables that show the different consequences that result from making different sequences of decisions. Decision models are often used during requirements analysis to model mandatory business rules and algorithms that can be put into the form of series of if/then statements. During requirements analysis, the requirements team may fail to create all useful and cost-effective decision models at all levels of the system hierarchy. Even if all appropriate decision models (e.g., decision trees) are produced, these decision models can be incomplete if they are missing mandatory:
 - Decisions (i.e., nodes in a decision tree)
 - Consequences of a decision (i.e., relationships between consecutive decisions or branches between nodes)
- **Event Models.** An event model consists of a cohesive set of events, their causes, and/or their consequences. Event models are useful during requirements analysis to determine how a system must react to events. Although they are occasionally documented in the forms of tables or matrices, event models are typically documented in the form of event trees (*forward*-search decision trees that document the sequences of events that can result from specific accidents or failures of a system or its components) or fault trees (*backwards*-search decision trees that document the sequences of events that can lead to a specific failure of a system or one of its components). Event modeling can be used for the analysis of functional requirements, but it is most often used for the analysis of reliability (failure modeling) and safety requirements (hazard modeling). During requirements analysis, the requirements team may fail to create all useful and cost-effective event models at all levels of the system hierarchy. Even if all appropriate event models are produced, these event models can be incomplete if they are missing mandatory:
 - Events
 - Causes of the events (e.g., preconditions and preceding events)
 - Consequences of the events (e.g., postconditions and following events)
 - Estimates of the probabilities that the events will occur (not always practical)
 - Estimates of the consequences of the events, should they occur
 - Estimates of the associated risks, typically calculated as probability times consequences
- **Formal Models.** A formal model is a collection of statements about a system's functionality or other properties stated in a language (e.g., Z, Petri Nets) with sufficient formality to allow the automated mathematical proof (e.g., via forward chaining and backward chaining) of useful properties such as consistency and the correctness of a translation of requirements into design. Ignoring the many

⁴ Beware of the common mistake of using “is a” when “is a kind of” is actually intended; instantiation is very different than subtyping.

difficulties and controversies surrounding the creation of formal models of the system (e.g., difficulty and expense of production and validation, potential defects in the model), such models can be used to unambiguously specify required system properties and behavior. During requirements analysis, the requirements team may fail to create useful and cost-effective formal models of all appropriate parts of the system's requirements because of missing statements of mandatory system properties or functions.

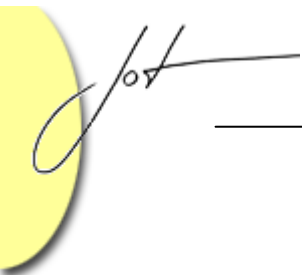
- **Performance Models.** A performance model is a model that can be used to analyze performance requirements and the allocation of these requirements to the steps of associated scenarios and the system components that will implement these steps. During requirements analysis, the requirements team⁵ may fail to create performance models of all appropriate parts of the system's requirements (e.g., functional requirements having associated performance requirements) at all levels of the system hierarchy. However, even if performance models are produced, they can be incomplete if they are missing mandatory:
 - Scenarios (normal case, exceptional case, start-up, shut-down, degraded mode, installation, etc.)
 - Scenario steps
 - Overall and allocated performance budgets
 - Evaluation and verification results showing that the performance budgets can be met
- **Process Model.** A process model is a model that documents system processes in terms of functions and the important relationships between them. Process models are used by requirements teams to identify and analyze functional requirements. During requirements analysis, requirements teams may fail to create all useful and cost-effective process models at all levels of the system hierarchy. Even if all appropriate process models are produced, these process models can be incomplete if they are missing mandatory:
 - Functions
 - Relationships:
 - Functional decomposition
 - Control flow
 - Data flow
- **Safety Models.** A safety model is a model that documents important aspects of system safety and is used by the requirements/safety teams to identify and analyze safety requirements. During requirements analysis, the requirements teams may fail to create all useful and cost-effective safety models at all levels of the system

⁵ Because performance models are typically used to verify support of an architecture for its associated performance requirements, performance models are most often produced by architecture teams once an initial architecture exists rather than by requirements teams prior to architecture development. However, requirements and architectures are often developed in parallel during an iterative and incremental development cycle, especially during the top-down hierarchical development of a multilevel system. Performance models can not only verify the feasibility of performance requirements but also help produce *derived* performance requirements at lower levels in the system hierarchy.



hierarchy. The following safety models may be missing or may be incomplete if they are missing any of the listed information:

- **Asset Models**, which model the valuable assets that must protected from accidental harm. Asset models include asset types and their values to stakeholders as well as the potential types and severities of harm that can occur to the valuable assets.
- **Accident Models**, which model credible potential accidents, whereby accidents are sequences of one or more events that cause unplanned and unintended harm to valuable assets. Accident models include accident types, resulting harm, asset harmed, and probability of accident occurrence.
- **Hazard Models**, which model credible potential hazards, whereby hazards are collections of conditions in the system and its environment that are prerequisites to the occurrence of accidents. Hazard models include hazards, their component conditions, their causes, their consequences (including accidents), and their probability of occurrence.
- **Safety Risk Models**, which model credible potential safety risks, whereby safety risks are risks to valuable assets due to accidents/hazards. Safety risk models include harm severities, accident/hazard probabilities, and associated hazard indices, safety trust levels of safety-related requirements, and safety integrity levels of system components that implement the safety-related requirements.
- **Security Models**. A security model is a model that documents important aspects of system security and is used by the requirements/security teams to identify and analyze security requirements. During requirements analysis, the requirements teams may fail to create all useful and cost-effective security models at all levels of the system hierarchy. The following security models may be missing or may be incomplete if they are missing any of the listed information:
 - **Asset Models**, which model the valuable assets that must protected from malicious harm. Asset models include asset types and values to stakeholders as well as the potential types and severities of harm that can occur to the valuable assets.
 - **Attacker Models**, which model credible potential attackers including attacker types and profiles (e.g., goals, abilities, and resources).
 - **Attack Models**, which model credible potential attacks, whereby attacks are sequences of one or more events that cause unauthorized and intended (by attackers) harm to valuable assets (including types, resulting harm, asset harmed, and probability of occurrence)
 - **Threat Models**, which model credible potential threats, whereby threats are collections of conditions in the system and its environment that are prerequisites to the occurrence of attacks. Threat models include threats, their component conditions, their causes (attackers and potential vulnerabilities), their consequences (including attacks), and their probability of occurrence.

- 
- **Security Risk Models**, which model credible potential security risks, whereby security risks are risks to valuable assets due to attacks/threats. Security risk models include harm severities, attack/threat probabilities, and associated threat indices and security integrity levels.
 - **State Models**. A state model consists of a cohesive set of states and the transitions between them. Often, the required behavior of a system (e.g., its functional requirements) depends on the mode or state of (1) the system itself, (2) one or more of its mandatory⁶ hardware or software components, (3) one or more of the mandatory external entities that interface with the system, and/or (4) the mandatory data that flows through the system. If state models are used as requirements models to analyze and determine functional requirements, then we need to determine if these state models are complete. During requirements analysis, the requirements teams may fail to create all useful and cost-effective state models at all levels of the system hierarchy. Even if all appropriate state models are produced, these state models can be incomplete if they are missing mandatory:
 - States (including initial and final states)
 - Substates
 - State transitions
 - Events that trigger transitions
 - Guard conditions on transitions
 - **Use Case Models**. A use case model analyzes the mandatory behavior of the system in terms of the systems external actors and the use cases that the system performs on their behalf. During requirements analysis, the requirements team may fail to create all useful and cost-effective use case models at all levels of the system hierarchy. Even if all appropriate use case models are produced, then the use case model can be incomplete if it is missing the following kinds of information (if representing needs or requirements):
 - Use case name
 - Use case goal / description
 - Primary Actor benefiting from the performance of the use case
 - Secondary actors involved in the use case
 - Use case preconditions
 - Use case paths:
 - Path name
 - Path-specific preconditions
 - Path interactions / steps
 - Path-specific postconditions
 - Use case postconditions
 - Use case invariants
 - Use case relationships:

⁶ Note that mandatory components tend to be architectural constraints rather than pure requirements.



- Includes
- Extends
- Generalization / Specialization / Subtypes
- Precedes

Recommendations

The following recommendations have proven useful with regard to evaluating the completeness of requirements models:

- Determine if all types of requirements models have been considered and evaluated for usefulness and cost-effectiveness for all appropriate system components at all levels of the system hierarchy (e.g., system of system, system, subsystem, assembly, subassembly, configuration item, etc.). It is by using multiple requirements models that we ensure the adequate completeness of the requirements. Because a single model provides only one view of the system being specified, using multiple models helps ensure that all of the requirements for the system are engineered.
- For each kind of requirements model produced, determine if all of its useful, cost-effective, and mandatory components have been identified, analyzed, and documented.
- When assessing model completeness, use a checklist of requirements model types and model components based on the preceding lists of this section.
- Establish project-specific model completeness guidelines and/or standards. Also establish practical waiver and/or deviation procedures for allowing requirements engineers to tailor model development to match the needs of the part of the system being analyzed.
- Ensure that the diversity and experience of the members of the requirements team match the complexity of the system being engineered.

3 COMPLETE INDIVIDUAL REQUIREMENTS

Definition

An **individual requirement is complete** if it contains all necessary information to avoid ambiguity and need no amplification to enable proper implementation and verification. To avoid ambiguity, a requirement must express the entire need and state all conditions and constraints under which it applies [Young 2004].

This section deals with the completeness of individual requirements. It does not deal with requirements that are entirely missing, which is an aspect of the completeness of the requirements repositories and specification documents.

Different kinds of requirements are specified differently. Therefore the following different kinds of requirements may be incomplete because different component parts of them are missing:

- Functional Requirements
- Data Requirements
- Interface Requirements
- Quality Requirements
- Constraints

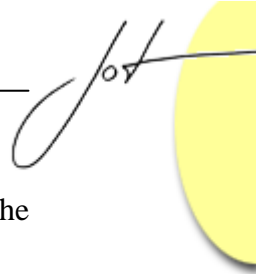
Functional Requirements

Functional requirements specify what the system must do (i.e., its functionality or the functions it must perform). Functional requirements specify a necessary behavior, typically in terms of how the black-box system shall respond (e.g., outputs and changes in state) to stimuli (e.g., inputs, passage to time, and changes in state).

With regards to functional requirements, requirements engineers often merely specify that the system shall perform ‘function X’. This begs the following questions:

1. What triggers the system to do X?
2. In what system state(s) shall the system do (or not do) X?
3. What are the performance requirements related to the system when doing X? Specifically, are there minimum/maximum limits on throughput, response time, jitter, etc. when the system does X?
4. In what state shall the system be left when it is done with doing X?
5. What are the data and interface requirements associated with doing X? Specifically, what data shall flow in and out of the system when the system is doing X? What data must the system store and what stored data must the system use when doing X? What interfaces shall the system use when doing X?
6. What capacity requirements are associated with doing X? Specifically, how shall the system’s performance when doing X change as system load nears the limits of its specified capacity?
7. Is doing X a critical function that must be preserved when the system goes into degraded mode? In other words, is performing X related to the system’s survivability requirements?
8. What is the safety trust level (STL) of the system doing X? What are the safety integrity levels (SILs) of the system components involved in doing X?⁷
9. Does the system doing X have related security requirements? Specifically, are only certain external entities (e.g., individual people, roles people play, groups of

⁷ Safety trust levels are a way to categorize the safety risks associated with individual system requirements, whereas safety integrity levels are a corresponding way to categorize the safety risks associated with individual system architectural components. If a system requirement has a given safety trust level (level of safety risk), then the system components that collaborate together to implement that requirement must have correspondingly high safety integrity levels in order to ensure that the requirement’s safety risk is properly mitigated.



- people, external systems) authorized to request the system to do X? Must the system ensure the security of private data or messages when doing X?
10. What shall the system do if it cannot do X? For example, too often requirements engineers only specify the normal case (“sunny day”) paths of use cases. What about the “rainy day” paths? Does the alternative exceptional behavior to performing X vary depending on system state?

A good way to ensure the completeness of normal functional requirements is to specify which:

- Events **trigger** the system to perform the function including any input data, requests received, or exceptions being handled
- **Preconditions** must hold for the system to be able to successfully perform the function, including system mode and state, the state of system externals, and the values of any system data
- **Actions** the system must perform when receiving the triggers when the preconditions hold
- **Postconditions** must hold once the system successfully performs its function

Many requirements specifications are incomplete in that they only specify what the system shall do under ‘sunny day’ (normal) conditions and thereby fail to specify what the system shall do under ‘rainy day’ (exceptional) conditions. This incompleteness forces the developers to guess at the stakeholders’ intentions or else they let these requirements fall through the cracks completely. Exceptional path functional requirements address these situations and differ from the preceding paths in that they specify the:

- Failed **preconditions** (i.e., the preconditions that caused the failure to occur)
- Resulting failure postconditions

Data Requirements

Data requirements specify what the mandatory data that the system must be able to input, manipulate, store, and output. As implied by the previously described data model, individual data requirements may be incomplete if they do not specify the following information:

- Data or object types
- Semantics (description of meaning, possibly in a data dictionary or project glossary)
- Data or object components (e.g., attributes, aggregate parts) including their types
- Initial or default values
- Range of possible values
- Units of measure
- Data volume
- Update frequency and type (e.g., periodic, random)
- Legitimate operations on the data

- Relationships including:
 - Associations
 - Aggregation, Membership, and Containment relationships
 - Generalization, Specialization, and/or Subtyping relationships between a subtype and a supertype
 - Instantiation relationships between a mandatory object and its type
 - Invariant relationships among the data (where appropriate)

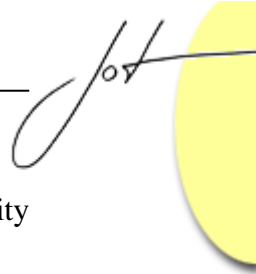
Interface Requirements

An interface requirement specifies a mandatory interface between the system and one or more externals. Interface requirements may be incomplete if they do not specify the following information:

- The name of the interface
- The definition (a brief description) of the interface
- The direction of the interface (i.e., inbound, outbound, or bidirectional)
- The service requests including:
 - Syntax (a.k.a., signature) including:
 - Parameters with type and direction
 - Return type
 - Semantics including:
 - Meaning
 - Protocol in terms of:
 - Assertions (e.g., preconditions, postconditions, and invariants)
 - State models
 - Concurrency semantics
 - Sequential vs. Concurrent
 - Synchronous vs. Asynchronous
 - Need for Reservation of Resources
 - Critical Regions
 - Exceptions Raised:
 - Syntax
 - Semantics
 - Interface-specific Data with Data Types (of Objects with Classes)
 - Quality Attributes (e.g., performance, reliability, safety, and security)

Quality Requirements

Quality requirements are based on the components of some associated quality model [Firesmith 2003b]. A quality requirement specifies a minimum acceptable amount of some quality factor (e.g., correctness, interoperability, performance, portability, reliability, safety, security, and usability) by specifying a minimum amount of some quality subfactor of the quality factor. For example, performance requirements might



specify a minimum acceptable amount of throughput or response time. Quality requirements typically consist of a:

- System-specific quality criterion that provides evidence of the existence of a given quality factor or quality subfactor under specific conditions.
- Quality measure threshold that provides a minimum or maximum acceptable amount of the quality factor or quality subfactor.

Therefore, a quality requirement may be incomplete if any of the following are missing:

- The system-specify quality criterion (or any of its parts)
- Any conditions under which the quality criterion is to exist
- The unit of measure to be used to measure the level of existence of the quality criterion
- The threshold amount of the quality measure.

An example would be that a system in state X shall respond to a specific kind of request Y (the system-specific quality criterion) within a maximum of 30 milliseconds (the quality measure threshold).

Constraints

There are a great many kinds of constraints, including architectural, design, implementation, testing, and business constraints. Because they vary so greatly, there is little that can generally be said about the manner in which they can be incomplete. The best one can do is to check each one against its source and subject matter experts to ensure that it expresses a complete need, is adequate to avoid ambiguity, and requires no amplification to be understood by all stakeholders.

Recommendations

The following recommendations have proven useful with regard to evaluating the completeness of individual requirements:

- Determine if all types of requirements have been considered and evaluated for usefulness and cost-effectiveness for all appropriate system components at all levels of the system hierarchy (e.g., system of system, system, subsystem, assembly, subassembly, configuration item, etc.).
- For each kind of requirement produced, determine if all of its useful, cost-effective, and mandatory components have been identified, analyzed, and documented.
- When assessing individual requirement completeness, use a checklist of requirements types and associated components based on the preceding lists of this section.
- Establish project-specific requirement completeness guidelines and/or standards. Establish practical waiver and/or deviation procedures for allowing requirements engineers to tailor requirements specifications to match the needs of the requirements being engineered.

4 COMPLETE REQUIREMENTS METADATA

Metadata refers to data *about* data. Thus, a requirement's metadata is data about that requirement rather than data listed in the requirement. For example, each requirement may have an associated project-unique identifier, priority, and status. Most requirements engineering processes include standards, procedures, and/or guidelines that mandate or recommend that each requirement have associated metadata that describes or characterizes the requirement. Thus, a 'complete' individual requirement by itself may still be incomplete in the sense that it is missing mandatory or appropriate metadata. Listed in alphabetical order, the following are typical types of requirements metadata that may be found to be missing when requirements metadata are evaluated for completeness:

- Categorization (e.g., data requirement, functional requirement, interface requirement, quality requirement, constraint)
- Criticality to Customer (e.g., critical, high, medium, low)
- Criticality to Users (e.g., critical, high, medium, low)
- Estimated Cost Range (e.g., high, medium, low)
- Frequency of Execution (e.g., high, medium, low - if functional requirement)
- Implementation Status (e.g., not implemented, implemented)
- Owners (e.g., owning producer for implementation)
- Prioritization (e.g., current release, next release, eventually)
- Probability of Defects (in the implementation)
- Project-unique identifier (PUID) for requirements identification and traceability
- Rationale
- Risk (associated with implementation) (e.g., high, medium, low)
- Source (i.e., requirements trace to document, goal, stakeholder)
- Status (e.g., Proposed, Analyzed, Specified, Evaluated, Approved, Baseline, Frozen, Retired)
- Stakeholders (i.e., requirements trace to stakeholder)
- Verification Method (e.g., analysis, demonstration, inspection, or testing)
- Verification Status (e.g., not verified, verified)
- Validation Status (e.g., not validated, validated)
- Volatility (e.g., high, medium, low)

Recommendations

The following recommendations have proven useful with regard to evaluating the completeness of requirements metadata:

- For each project, determine what metadata can, should, and shall be associated with individual requirements.
- Document the project requirements metadata policy in the appropriate organizational or project standard, procedure, or guideline document.

- Communicate the metadata to both requirements engineers specifying requirements and stakeholders evaluating requirements.
- If specific types of requirements metadata are not explicitly specified in such a standard, procedure, or guideline, then requirements engineers and requirements evaluators should use a list such as the previous one to help them avoid having an appropriate type of requirement metadata fall through the cracks.
- Store requirements metadata with the associated requirements in the project requirements repository.
- As part of the evaluation of each individual requirement, also evaluate the requirement's metadata for completeness and correctness.
- Use a requirements tool to rapidly identify and report missing metadata.
- In terms of benefit to the project, the most important metadata to capture and those metadata most likely to be mandatory include:
 - Project-Unique Identifier (PUID)
 - Prioritization
 - Rationale [Hooks and Farry 2001]
 - Source
 - Status (may include more than one kind of status)
 - Verification Method

5 CONCLUSION

Today, it is reasonably well known that requirements will never be *totally* complete, finished, and finalized as long as a system is in service and must evolve to meet the changing needs of its customers and users. This is especially true when the system is being produced in an iterative and incremental fashion whereby the requirements are being developed in parallel with the architecture and design. However, the preceding sections of this column should make it clear that requirements engineering teams can do much more to ensure that their requirements are adequately complete at any given point in time. Because in practice, many requirements engineers do not realize just how many ways their requirements models and associated requirements can be incomplete, their resulting models and requirements are unfortunately far too incomplete, ambiguous, and unverifiable. It is the intent of this column to help them generate checklists that will enable them to produce better-quality requirements and requirements models.

In this column, we began by providing a brief overview of the reasons why requirements completeness is important. We then addressed the completeness of requirements models in terms of the different types of requirements models and their potentially missing component parts. Requirements models were followed by a look at the completeness of various types of individual requirements, once again in terms of the potentially missing component parts that could make them incomplete. Finally, we

presented a brief look at the different kinds of metadata about requirements that could potentially be missing.

As you can see, evaluating the completeness of requirements models, requirements, their metadata, and the repositories in which they are stored is not a trivial exercise. It is certainly worthy of more than a single checkbox on a simple requirements evaluation checklist. Anyone responsible for evaluating these requirements work products would be well advised to use a more complete collection of potentially missing items. The contents of this column would be a good starting point for producing such a project or organizational set of information to look for.

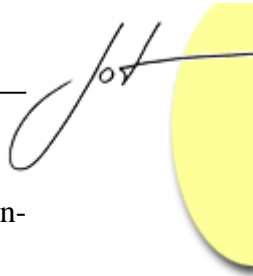
Whereas incomplete individual requirements are a major problem, incomplete sets of requirements are an even greater problem. Although this column has provided much that can prevent the specification of incomplete individual requirements, we are not done with this topic. In next issue's column, we continue with our discussion of requirements completeness by addressing the completeness of the requirements repositories that are used to store both requirements and the requirements models from which they are derived, the completeness of requirements documents derived from such repositories of requirements, the completeness of sets of requirements documents, the completeness of requirements baselines, and determining how complete is complete enough when using an incremental and iterative development cycle.

ACKNOWLEDGEMENTS

Many thanks go to my colleagues Peter Capell, Nancy Mead, and Tim Morrow, who reviewed this column and provided helpful observations and recommendations.

REFERENCES

- [Hooks and Farry 2001] Ivy F. Hooks and Kristin A. Farry: *Customer-Centered Products: Creating Successful Products through Smart Requirements Management*, American Management Association, 2001.
- [Firesmith 2003a] Donald G. Firesmith: "Specifying Good Requirements", in *Journal of Object Technology*, vol. 2, no. 4, July-August 2003, pp. 77-87. http://www.jot.fm/issues/issue_2003_07/column7
- [Firesmith 2003b] Donald Firesmith: "Using Quality Models to Engineer Quality Requirements", in *Journal of Object Technology*, vol. 2, no. 5, September-October 2003, pp. 67-75. http://www.jot.fm/issues/issue_2003_09/column6
- [HSE 1995] *Out of Control: Why Control Systems Ho Wrong and How to Prevent Failure* (2nd Edition), Health and Safety Executive (HSE), 2003



[Leveson 1995] Nancy G. Leveson: *Safeware, System Safety and Computers*, Addison-Wesley, 1995

[Standish 1994] The Standish Group: *The CHAOS Report (1994)*, The Standish Group, 1994.

https://secure.standishgroup.com/sample_research/chaos_1994_1.php

[Sommerville97] Ian Sommerville and Pete Sawyer: *Requirements Engineering: A Good Practices Guide*, John Wiley & Sons, 1997.

[Young 2004] Ralph R. Young: *The Requirements Engineering Handbook*, Artech House, 2004.

Disclaimers

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

The views and conclusions contained in this column are solely those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Software Engineering Institute, Carnegie Mellon University, the U.S. Air Force, the U.S. Department of Defense, or the U.S. Government.

About the author



Donald Firesmith is a senior member of the technical staff at the Software Engineering Institute (SEI), where he helps the US Government acquire large, complex, software-intensive systems. Working in industrial software development since 1979, he has worked primarily with object technology since 1984 and has written 5 books on the subject. During the last four years, he has developed the world's largest (1,100+ webpage), free, and open source informational website of reusable process engineering components. Based on the OPEN Process Framework (OPF), it is located at <http://www.donald-firesmith.com>. Currently writing a book on the engineering of safety requirements, he can be reached at dgf@sei.cmu.edu.