Reducing Requirements Defect Density by Using Mentoring to Supplement Training

John Terzakis
Intel Corporation, USA
john.terzakis@intel.com

***Abstract*—In a previous short paper [1], data demonstrating that using mentoring to supplement training had a significant impact on reducing requirement defect density levels from initial to final versions of requirements specifications for a software product was presented. This paper provides additional details of the initial training, mentoring and review methods delivered by the requirements Subject Matter Expert (SME). In addition, data from a third generation of the requirements specification is now available, which supplements the existing defect data from the earlier two generations. Requirements authors typically receive little formal university training in writing requirements. Yet, they are expected to write requirements that will become the foundation for most future product development. Defects introduced during the requirements phase of a project impact multiple downstream work products and, ultimately, product defect and quality levels. Many companies, including Intel Corporation, have recognized this skills gap and have created requirements training classes to address this issue. While effective in providing the fundamentals of good requirements writing, much of this knowledge can be misapplied or lost without proper mentoring from an experienced requirements SME. Our experience over the last decade at Intel has found that adding SME peer mentoring improves both the rate and depth of proper application of the training, and improves requirements defect density more than training alone. The data from case studies across three generations of a software product will expose the issues with training alone and the benefits of combining training with SME mentoring in order to reduce requirements defect density levels. All three generations of requirements specifications achieved at least a 90% reduction in requirements defect density from initial to final releases.**

Keywords-requirements specification; requirements defects; requirements defect density; training; mentoring; multi-generational software products.

I. INTRODUCTION

This paper is an extension of a previous short paper [1] that presented data demonstrating the benefits of using mentoring by an experienced requirements SME to supplement requirements training. Details of the training materials, additional mentoring and review examples and additional defect density data from a third generation of the software requirements specification will be presented.

While bachelor's degrees exist for a variety of Engineering disciplines, degrees and even undergraduate courses in Requirements Engineering are scarce. Primary requirements authors (those whose primary role is to elicit and write requirements) may have some training. However, secondary authors (those whose primary role is architecture, development, testing, etc.) may have little or no requirements training. As Berenbach, et al, state "Requirements analysts typically need significant training, both classroom and on the job, before they can create high-quality specifications." [2] To close this skills gap, many companies have created in-house requirements courses or contracted third-party trainers to teach the basics of well-written requirements. Many are based on the IEEE 830 standard, [3], or the good, practical books published in the field over the past fifteen years [4][5]. At Intel, in-house requirements courses have been taught to over 15,000 employees since 1999. While useful for providing an initial understanding of the issues and challenges of requirements authoring, the knowledge gained through these courses can be misapplied or lost due to the inexperience of authors in writing effective requirements. By pairing them with an experienced requirements SME, the authors can be provided with early feedback on the deficiencies of their requirements.

This paper is organized into six sections. Section I is the introduction. Section II reviews the requirements training materials in detail. Section III discusses the backgrounds of the authors and the review process for the requirements. Section IV provides information on trust, early requirements samples and mentoring. Section V presents the requirements defect densities for all three requirements specifications. Section VI analyzes the data and derives conclusions based on the data.

II. REQUIREMENTS TRAINING MATERIALS

The requirements authors attended a training session on requirements writing (some details of which are available publicly [6]) prior to beginning work on the Software Requirements Specification (SRS) for their generation of the software. These training sessions focused on the issues with natural language, attributes of well-written requirements, a consistent syntax for requirements and an

introduction to Planguage ("Planning Language") [7]. This training class was a full day in length.

The training begins with the purpose of requirements. Specifically, requirements help establish a clear, common, and coherent understanding of what the system must accomplish. Clear means that all statements are unambiguous, complete, and concise. Common indicates that all stakeholders share the same understanding. Coherent ensures all statements are consistent and form a logical whole. Given the number of people that consume requirements and their differing experiences and backgrounds (SW, HW, testing, etc.), it is essential to project success to create a set of requirements that produces this clear, common and coherent understanding of what is being architected and designed. Without this, assumptions will be made and differences of interpretation will form leading to defects, rework, and schedule slips. A defect is defined as a mistake in a work product (SRS, code, etc.). Defects can be caused from lack of knowledge, lack of attention or both. Minimizing defects improves product quality.

A good analogy is building a house. If a house is built with improper specifications and with shoddy workmanship (i.e., a poor foundation), it will likely be unstable and require "propping up" (rework) to improve its stability. Similarly, a software product built with a poorly written set of requirements (i.e., high defect rate) will also be unstable and require "propping up" (defect fixes) to stabilize it. An unhappy customer is the likely result in both cases.

The class continues with a discussion of natural language, the language used in everyday conversations and writing (e.g., emails and other correspondences). It is typically the easiest form of language to learn and usually requires no formal training. Infants learn natural language by listening and repeating the words spoken by their parents or siblings. While this is true for any language, the training (and this paper) focuses on the natural language of American English.

While easy to learn, natural language presents a plethora of issues for requirements. These issues include ambiguity, weak words, unbounded lists and grammatical errors. Ambiguity occurs when a requirement can have multiple interpretations. Weak words lack a precise or common meaning (i.e., they are subjective). Unbounded lists have no beginning, no end or lack both. Grammatical errors include improper verb selection, using a slash (e.g., does read/write mean "read and write" or "read or write"), compound statements and passive voice.

Here are few examples that demonstrate the issues with natural language:

The software should log invalid access attempts.
Issue: "should" implies that this is optional

The software shall be easy to install.
Issue: "easy to install" is subjective

An error report shall be generated.
Issue: passive voice—who or what is generating the error report?

The software shall support 25 or more users.
Issues:
- "support" is a weak word, i.e., it lacks a precise meaning
- 25 or more is an unbounded list (there is no upper limit)

In order to reduce the issues with natural language, the class next presents the ten attributes of well-written requirements. They are listed in Table I below.

TABLE I: 10 ATTRIBUTES OF WELL-WRITTEN REQUIREMENTS

Attribute	Attribute
Complete	Prioritized
Correct	Unambiguous
Concise	Verifiable
Feasible	Consistent
Necessary	Traceable

A requirement is **complete** when development can proceed with minimal risk of rework or wasted effort. If a requirement is not complete enough, the development team will have to make assumptions about its meaning. These assumptions can lead to differences of interpretation among architects, coders and testers, which will result in a higher number of defects being filed by the testing team, and ultimately to inefficiency and unnecessary work.

Not Complete: *The software shall allow some number of incorrect login attempts.*

Complete: *When more than 3 incorrect login attempts occur for a single user ID within a 5 minute period, the software shall lock the account associated with that user ID until reset by the administrator.*

A requirement is **correct** when it has been reviewed by stakeholders and SMEs (both technical and requirements) and any errors have been fixed. These reviewers should ensure that the requirement is accurate and does not contain invalid assumptions, logic errors, typos, or conflicts with internal documents or industry specifications.

Not Correct: *The software shall calculate the area of a triangle as the base multiplied by the height.*

Correct: *The software shall calculate the area of a triangle as one-half of the base multiplied by the height.*

A requirement is **concise** when it conveys its intent as succinctly as possible. A requirement is not concise if it contains more words than necessary, multiple requirements

("ands" or multiple sentences), or superfluous information (opinions, rationale, etc.). A concise requirement is written using the least number of words needed to express its intent.

Not Concise: *We've had a lot of negative feedback about the format of the current local time. It is now displayed only in 24 hour format. We should have a configuration menu option to select 12/24 hour format.*

Concise: *The configuration menu shall display an option to display the current local time in either 12 hour or 24 hour format.*

A requirement is **feasible** if it can be shown to be implementable. Feasibility can be demonstrated through implementation in previous products, simulations, analysis and prototyping. Evolutionary requirements, those based on pre-existing, verified requirements, are typically easier to prove feasible. Revolutionary requirements, those that have no current basis for development, will require a much more thorough analysis by experienced architects and developers.

Not Feasible: *The software shall allow an unlimited number of concurrent users.*

Feasible: *The software shall allow a maximum of two thousand concurrent users.*

A requirement is **necessary** when it is needed from a customer, stakeholder, business or competitive perspective. These requirements may have been gathered during an elicitation process (internal or external), included as part of a strategic roadmap or business plan, required as the result of a competitive analysis or created to provide a product differentiator. Requirements that are not necessary create wasted effort and bloat project budgets.

Not Necessary: *The software shall be distributed on magnetic tape, 5.25 inch floppy disks, 3.5 inch floppy disks, CD-ROM and DVD media.*

Necessary: *The software shall be distributed on DVD media.*
Rationale: DVD media listed as the top choice for distribution based on feedback from our top 50 OEMs.

A requirement is **prioritized** when it is assigned a rank or order level relative to other requirements. Priority can be determined by a number of factors including value, risk, development time, project cost, and resources required. Priority levels are typically on a three point (High, Medium and Low) or five point scale (1 = Highest and 5 = Lowest). An alternative is to rank each from 1 to n, where n is the total number of requirements. However, this method is usually eschewed by most development teams if there are more than fifty requirements due to the time required to assess and assign a unique value to each.

Not Prioritized: *All requirements are critical and must be implemented.*

Prioritized: *80% of requirements High, 15% Medium and 5% Low.*

Priority is an important attribute for requirements to possess. Too often, all requirements are deemed "critical". If the schedule slips, the team has no basis for determining which requirements can be postponed to a future release since all are of equal priority.

A requirement is **unambiguous** when it has the same meaning for everyone. Since requirements will be read and utilized by many different stakeholders, writing them unambiguously is critical to achieving a common understanding. Each stakeholder has a different background and experience level, so the requirements must be written with precise language that is not open to different interpretation. All subjectivity must be removed.

Ambiguous: *The software must install quickly*

Unambiguous: *Where using unattended installation with standard options, the software shall install in under 3 minutes 80% of the time and under 4 minutes 100% of the time.*

A requirement is **verifiable** if it can be determined that the requirement will be or has been implemented properly. This can be accomplished in a number of ways including prototyping, analysis or testing. A requirement is not verifiable if it is incomplete, incorrect, not feasible or ambiguous, so there is a dependency on some of the other attributes.

Not Verifiable: *The user manual shall be easy to find on the DVD.*

Verifiable: *The user manual shall be located in a folder named "User Manual" in the root directory of the DVD.*

A requirement is **consistent** when it does not contradict any other requirements or documents. This is an attribute that must be evaluated for against the entire set of requirements, not just an individual requirement. The consistency test must be applied to other requirements, roadmaps, internal specifications and industry standards. Of all the attributes of a well-written requirement, this one is the most difficult to determine because of all the interrelationships that must be examined.

Not Consistent:
#1: The user shall only be allowed to enter whole numbers.
#2: The user shall be allowed to enter the time interval in seconds and tenths of a second.

Consistent:
#1: The user shall only be allowed to enter whole numbers except if the time interval is selected.
#2: The user shall be allowed to enter the time interval in seconds and tenths of a second.

A requirement is **traceable** if it has a unique and persistent identifier. Unique means that each requirement has its own identifier and there are no duplicate names. Persistent indicates that an identifier, once associated with a requirement, can never be used for another requirement. Traceability allows requirements to be linked to other design artifacts like use cases, test cases and even source code. Many requirements management tools automatically assign these identifiers.

Not Traceable: *The software shall prompt the user for the PIN.*

Traceable: *Prompt_PIN: The software shall prompt the user for the PIN.*

Many of these ten attributes are interrelated. For example, a requirement cannot be complete if it is ambiguous. Likewise, a requirement cannot be correct if it is not verifiable. With the exception of consistent, each requirement can be evaluated individually relative to the nine other attributes.

In order to provide consistency, the Intel requirements training introduces a requirements syntax of the form:

[Trigger][Precondition] Actor Action [Object]

Note that the objects in square brackets are optional. The actor is the part of the software or system that implements the requirement. The action is the act taken by or event done by the actor. Finally, the object is what the actor takes the action on. When present, a trigger is some event or state that causes the requirement to occur. When present, the precondition must be satisfied for the requirement to be executed.

Intel has adopted the convention of using the imperative "shall" for functional requirements and "must" for non-functional requirements, which aligns with the common usage in industry. The words "should" and "may" imply optionality and thus are not used for requirements. Developers should not be given the option as to whether to implement the requirement or not. Any requirement assigned to a developer must be implemented.

While the words "shall" and "must" are generally recognized as imperatives in the U.S., it is not the case in some other countries. In fact, sometimes the exact opposite is true. The word "should" carries a stronger meaning than the word "shall". This discrepancy can be resolved by adding a note at the beginning of any requirements specification indicating that "shall" is the imperative.

Here is an example of a requirement written using the syntax above:

When the high temperature threshold limit is exceeded and event logging is enabled, the event monitoring software shall record the date and time of the high temperature event in the system log.

Trigger: the high temperature limit is exceeded
Precondition: event logging is enabled
Actor: event monitoring software
Action: record
Object: date and time of the high temperature event

To complement this syntax, the Intel requirements program has adopted EARS (Easy Approach to Requirements Syntax) that was developed by Alistair Mavin et al [8] at Rolls-Royce. This group applied EARS to requirements for the aviation industry. It establishes a small number of specific constrained natural language patterns for various types of requirements. They are summarized in Table II.

TABLE II: EARS PATTERNS

Pattern Name	Keyword(s)	Description
Ubiquitous	N/A	Always occurring or a fundamental property
Event-Driven	When	Occurring as the result of an event or trigger
Unwanted Behavior	If…then	Occurring as the result of an unwanted behavior or error condition
State-Driven	While	Only occurring while in a particular state
Optional Feature	Where	Only occurring where an optional feature is present
Complex	Combinations of the patterns when, if/then, while, and where	Occurring as the result of multiple patterns

Ubiquitous requirements are universal. They exist at all times and state a fundamental system property. They do not require any stimulus in order to execute. For most products, ubiquitous requirements are usually in the minority. Here is an example:

The software shall be available for purchase on the company web site and in retail stores.

Requirements that are **event-driven** occur as the result of an event or a trigger. In other words, there must be some stimulus that causes the requirement to execute. The keyword "when" denotes this pattern. Here is an example:

***When** a DVD is inserted into the DVD player, the software shall illuminate the "DVD Present" LED.*

The **unwanted behavior** pattern applies to requirements that handle unwanted behaviors including error conditions, failures, faults, alarm conditions, disturbances and other undesired events. The keywords "If" and "then" designate this pattern. Here is an example

***If** there are not sufficient funds in the account, **then** the software shall reject the withdrawal request.*

A **state-driven** requirement occurs if and only if the system is in a particular state. States can be conditions like operating on battery power, using cruise control and holding down a key. The keyword "while" indicates this pattern. Here is an example:

***While** the AC power is off, the software shall illuminate the yellow LED.*

The **optional feature** pattern applies to requirements that only occur if an optional feature is present. These features may be software or hardware related. Here is an example:

***Where** a HDMI port is present, the software shall allow the user to select HD content for viewing.*

A **complex** requirement occurs when multiple patterns are needed to describe the action or actions. It uses combinations of the four previous keywords (when, if/then, while, and where). Here is an example:

***While** in startup mode, **when** the software detects an external flash card, the software shall store video on the flash card.*

The last part of the class teaches an overview of Tom Gilb's Planguage [7], along with exercises to reinforce the concepts. Planguage utilizes a series of keywords to help define a more complete requirement by using a standard format. The result is that requirements have fewer omissions or missing information, reduced ambiguity and increased reuse. Examples of essential keywords for functional requirements appear in Table III.

TABLE III: KEYWORDS FOR FUNCTIONAL REQUIREMENTS

Keyword	Description
Name	a short, descriptive name for the requirement
Requirement	text defining the requirement
Rationale	justification for the requirement
Priority	importance of this requirement relative other requirements
Status	current state of the requirement
Contact	who to contact with questions
Author	who originally created the requirement
Revision	revision number for the requirement
Date	date of the latest revision
Defined	an acronym or term definition

Additional essential keywords for non-functional requirements appear in Table IV that follows. These keywords help bound the testing space for quality and performance requirements. The Scale and Meter define what the measure is and how it will be measured. Intel uses Minimum, Target and Outstanding (referred to in *Competitive Engineering* [7] as Must, Goal and Stretch) to define success for the non-functional requirement. Note that Planguage is flexible in allowing keyword names to be changed and other keywords to be added.

TABLE IV: KEYWORDS FOR NON-FUNCTIONAL REQUIREMENTS

Keyword	Description
Scale	scale of measure used to quantify the requirement
Meter	process or device used to establish location on a Scale
Minimum	minimum level required to avoid political, financial, or other type of failure
Target	level at which good success can be claimed
Outstanding	feasible stretch goal if everything goes perfectly

The requirements previously presented would be entered into the "Requirement" keyword field. The other fields would be entered by the original author or added by others as more details about the requirement become available. The essential keywords should be entered for all requirements. Additional, optional keywords can be added as needed by team responsible for the requirements. If a Requirements Management Tool (RMT) is used, it may populate many fields automatically (e.g., persistent ID, author, revision, and date).

An example of a functional requirement written using Planguage is shown in Table V. The name is short and succinct. The text is written for an optional feature using the EARS pattern ("where") and the proper requirements syntax. All other keyword fields are populated. Any missing information is quickly identifiable.

TABLE V: EXAMPLE FUNCTIONAL REQUIREMENT

Keyword	Description
Name	Display_Optional_Thesarus_Icon
Requirement	Where a thesaurus is present, the software shall display a thesaurus icon on the toolbar.
Rationale	Only display the icon if the thesaurus has been purchased.
Priority	High
Status	Implemented
Contact	John Jones
Author	Sue Morris
Revision	1.1
Date	January 18, 2013

An example of a non-functional requirement written using Planguage is presented in Table VI. The word "minimize" in the requirement is ambiguous. However, since this is a non-functional requirement, the additional keywords Scale, Meter, Minimum, Target and Outstanding define what "minimize" means (between 2 and 5 seconds). The requirement describes what will be measured in the Scale (time) and how it will be measured in the Meter (from order submit to order complete displayed). Optional keywords could include Past (a list of previous order processing times), Record (the fastest processing time recorded) and Current (current order processing time).

TABLE VI: EXAMPLE FUNCTIONAL REQUIREMENT

Keyword	Description
Name	Order_Processing_Time
Requirement	The software must minimize order processing time.
Rationale	Improvement request from top 5 customers
Priority	High
Status	Committed
Contact	Nick Terry, Director of Marketing
Author	Kristina Smith
Revision	0.7
Date	November 19, 2012
Scale	Time
Meter	Measured from the user clicking on the "Submit Order" icon to the display of the "Order Complete" message on the order entry menu.
Minimum	5 seconds
Target	4 seconds
Outstanding	2 seconds

III. AUTHOR BACKGROUNDS & REVIEW PROCESS

The three lead requirements authors (denoted as Author1, Author2 and Author3) attended the requirements writing training described in the previous section. None had any prior experience writing requirements. All were senior software developers with extensive product experience and were located in the United States.

Author1 created the first SRS for the software (SRS1). Prior to this SRS, the "requirements" that existed were scattered across a variety of locations (documents, presentation slides, spreadsheets, emails and web sites) and lacked a consistent syntax. This author captured a combination of important legacy and new requirements that were stored in a RMT. No other authors wrote requirements for SRS1.

Author2 started with the final set of requirements from the first author's SRS (SRS1, revision 1.0). Due to the increasing complexity of the product, Author2 was assisted by four other authors starting with revision 0.4. They contributed to about 25% of the new requirements. None of these authors received the requirements writing training and they were all located in different countries. Their impact on requirements defect density will become apparent when the data is presented in a subsequent section.

Author3 began with the final set of requirements from the second author's SRS (SRS2, revision 1.0). This author was assisted by over a dozen other authors starting at revision 0.5. They wrote approximately two thirds of the new requirements. About half of these authors were in the United States and attended the requirements writing class. Those outside the U.S. did not. Only the composite data for all authors will be reported. No defect statistics by geographic location were collected.

Each of the requirements authors followed a similar process. After completing the requirements writing training, the authors began work on their SRS and submitted early samples for review. The same requirements SME provided feedback to each of them to provide consistency. Since requirements were reused across SRS generations, Author2 and Author3 were able to benefit and begin their work from a stable, well-reviewed set of requirements from their predecessors, although some defects did remain. There was approximately one year between the start of each SRS.

The early review samples (part of the revision 0.3 release for each SRS) showed requirements defect densities of about 10, 5 and 4 defects per page for Author1, Author 2 and Author3 respectively. These figures represent the baseline for this paper. While some of the key concepts taught in the requirement writing training were applied (e.g., a consistent syntax and use of Planguage), other key concepts were not (including the authors' continued use of weak words, failure to check requirements for the ten attributes, and logic issues). With this baseline in place, the requirements SME began mentoring each of the authors.

Each SRS followed a similar path to a mature document. Revision 0.5 documents captured feedback from peer (other software developers) reviews of previous revisions. Revision 0.7 documents incorporated stakeholder (testers and other cross functional team members) feedback. Formal change control was started at revision 0.8. At that point, any changes to the requirements had to be formally submitted to and approved by a change control board. Revision 1.0 was the "official" release. All SRS revisions were managed from the RMT.

Note that the examples that follow have been slightly modified from their original form to maintain author and product confidentiality. Also, only the requirement itself is presented, not the full complement of Planguage keywords.

IV. Early Requirements Samples & Mentoring

To be most effective, requirements mentoring needs to occur early in the requirements lifecycle. In this way, writing style mistakes and tendencies can be corrected before they are copied and repeated on the hundreds of requirements that will follow. This is known as *defect prevention* (versus the *defect detection* that occurs as part of the testing process). However, many authors are reluctant to release requirements before they are "ready" from their perspective. At this point, many bad habits may have already been developed. To avoid this situation, the requirements SME must establish a trust relationship with the author.

How is this trust relationship established? First, the requirements SME must demonstrate a level of understanding of the product domain. The SME does not have to be a content expert but should know about the key functionality of the software. Second, the SME has to offer constructive feedback. The requirements should be reviewed against a checklist of criteria and the specific deficiencies clearly identified using objective feedback ("this word is ambiguous" vs. "this wording is bad"). Third, confidentiality has to be maintained. The author must feel comfortable that the feedback on the early requirements samples provided will not be provided to management or used in any way as part of a performance review. Fourth, the requirements SME has to provide the feedback in a timely manner. Otherwise, writing issues will propagate to other requirements.

Outside of an initial introductory face-to-face meeting, all interactions between the requirements SME and the primary authors were conducted over the telephone since they worked at different locations. In the case of the international authors, all meetings were held via telephone. As the data will demonstrate, geographic dispersion was not a detriment to the mentoring and learning process.

After establishing the trust relationship with Author1, the requirements SME reviewed early requirements samples, identified quality issues, documented those issues and then worked with the author to rewrite the requirements to remove the defects. Here is an initial sample requirement from Author1:

The software should have radio style buttons to enable/disable graphics cards.

Issues with this requirement include its optionality, the design constraint, use of a slash and over generalization. Specifically, the word "should" implies optionality. In other words, it is not mandatory. The word "shall" is the preferred choice for functional requirements. The term "radio style buttons" is a design constraint. Requirements should focus on the "what", not the "how". Why is this style of button specifically called out? Requirements should not constrain designs unnecessarily--leave the implementation details to the software developers. The slash ("/") can cause confusion as it can mean "and" or "or". In this case, the meaning is clear ("or") but in other cases, it may create confusion (e.g., administrators/users. Does this mean "administrators and users" or "administrators or user"?). Finally, the term "graphics cards" is an over generalization. Which type of graphics cards? All graphics cards? Specific graphics cards?

Having identified and documented the issues, the mentoring sessions focused on answering the questions about the missing pieces of information, discussing how to correct the defects and then rewriting the requirements. Some of this information could only be obtained through direct interaction with the author. In the previous example, the updated requirement became:

The software shall display an option to enable or disable graphics cards installed in the PCIe bus.

The requirement now has an imperative ("shall") and clearly identifies the action to be taken without ambiguity or unnecessary implementation details. Other requirements in this initial sample had similar types of defects. Additional mentoring sessions were conducted to discover and correct these requirements.

For later revisions of the SRS, the requirements SME reviewed all requirements and provided detailed feedback on the defects identified. Each requirement was then updated in a mentoring session. By the latter revisions of the SRS, this author was self-reviewing requirements using the checklists provided in the requirements training class. These SRS revisions required only minor rewrites and contained far fewer defects.

Author2 had the advantage of starting with the well-reviewed set of requirements from Author1. This author had to determine what changes were needed from the baseline of existing requirements and then started writing requirements for new features. Despite the strong foundation, initial samples from Author2 demonstrated similar issues as Author1. Here is a sample:

The software needs to provide the ability to wake on a wireless LAN event.

An analysis of this requirement reveals that it is written as a ubiquitous requirement when it really is not ubiquitous, lacks an imperative, uses weak words and is ambiguous with respect to the wireless LAN event. First, this is not a requirement that is universal. It does require a stimulus. What causes the software to wake? Second, the word "needs" should be replaced with "shall". Third, the action

"provide the ability" uses a weak set of words. How is it provided? What ability? Finally, there are many different types of wireless LAN events. Which specific one is being referenced here?

During a mentoring session with Author2, the requirements SME was able to elicit the missing information. The key pieces of information were that this requirement should only occur in a certain OS state (sleep) and that there needs to be a trigger (detection of a "Magic Packet" on the wireless network). Once all the pieces of the requirement were identified, the rewrite became:

While the operating system (OS) is in a sleep state, when the software detects a Magic Packet on the wireless network, the software shall wake the OS.

Defined: Magic Packet: A broadcast frame containing anywhere within its payload 6 bytes of 1's (0xFFFF FFFF FFFF) followed by 16 repetitions of the system MAC address.

The requirements SME reviewed all new requirements from revision 0.3 through revision 1.0. Starting with revision 0.4, four additional international authors contributed to the SRS. Unfortunately, the requirements training was not available at their work sites. This made the mentoring more difficult as they were not familiar with the rules and concepts from the course. It also resulted in a noticeable increase in requirements defect density. However, the one-on-one mentoring sessions to discuss feedback and rewrite their requirements were eventually effective in counteracting that original trend. The SME was assisted by Author2, as this particular author embraced the training to the extent that he would help others to correct their requirements during review meetings.

Author3 benefited from the requirements work done by the previous two authors. This author inherited a document of slightly over 100 pages and feature requests from software developers and testers that added another 200 pages to the initial SRS release. The requirements SME did not get the opportunity to review any early samples of requirements. The first review of SRS3 was done at revision 0.3. Here is an example of a requirement from it:

In the past, we didn't handle image errors well. Need the ability to recover from a corrupt image.

This requirement has multiple issues. The first sentence is additional information and should not be part of the requirement text. The second sentence is written in the passive voice. There is no actor identified to do this "recover". In addition, "ability to recover" is vague and ambiguous. It needs to be defined more clearly. Finally, what is a corrupt image? How is that determined? With mentoring, this requirement became:

If the calculated and stored software image checksums do not match, then the software shall:

- *Display an error message indicating that the image is corrupt*
- *Prompt the user to select loading a new image from a USB port or to exit the update process*

Rationale: Customer feedback from our top OEM has indicated that error handling for corrupt software images needs to be improved.

V. RESULTS

The data in the tables that follow documents the requirements defect densities (measured in defects per page or DPP) for each revision of the SRS documents. A single requirement could have multiple defects (e.g., not feasible, weak words, ambiguity, etc.). Note that these formatted revisions were generated from requirements that were stored and maintained in the RMT. The elapsed time from initial to final release was approximately one year in each case. The same requirements SME mentored all contributing authors and reviewed all SRS revisions.

Table VII presents the requirements defect density for SRS1, which was written by Author1. From revision 0.3 to 1.0, the defect density dropped from 10.06 DPP to 0.22 DPP, a reduction of about 98%! Without mentoring, this author would have continued to inject about 10 defects per page of requirements. At revision 1.0, there would have been approximately 450 defects in the SRS. As a result of SME mentoring, the actual document had only 10 defects, a difference of 440 defects. The vast majority of these defects would have eventually propagated into the code, requiring rework to remove them.

TABLE VII: REQUIREMENTS DEFECT DENSITY SRS1

Revision	# of Defects	# of Pages	Defects/ Page (DPP)	% Change in DPP
0.3	312	31	10.06	
0.5	209	44	4.75	-53%
0.6	247	60	4.12	-13%
0.7	114	33	3.45	-16%
0.8	45	38	1.18	-66%
1.0	10	45	0.22	-81%
Overall % change in DPP revision 0.3 to 1.0: ***-98%***				

The data in Table VIII shows the requirements defect density for SRS2. This document was written primarily by Author2, who was assisted by four additional authors starting at revision 0.4. Their impact is immediately evident from the table. While the defect rate dropped slightly from revision 0.3 to 0.4, it rose by 20% from revision 0.4 to 0.5 with the contributions from the untrained authors. However, with mentoring from the requirements SME, the downward trend in defect density resumed with

revision 0.7 and subsequent revisions. Overall, this SRS went from an initial 4.58 DPP to a final 0.94 DPP, an overall decrease of 79%. Again, the importance of mentoring is quite apparent. At the 5.40 DPP rate present at revision 0.5 (due to the injection of requirements from untrained authors), the final revision of the SRS would have had about 659 defects versus the 115 defects present in revision 1.0. The result is 544 fewer defects introduced into the software development process.

As mentioned previously, the four additional requirement authors that contributed to SRS2 were located outside of the United States. The key challenges for the requirements SME were to provide mentoring without these authors having taken the training and to establish trust relationships without meeting these authors in person. The first challenge was addressed by reviewing the training materials with the authors on a one-on-one basis. While not as effective as full classroom training, the key concepts were conveyed. The second challenge was a bit more difficult due to the distance and language barriers. However, by providing previous testimonials on the advantage of mentoring and the data on the importance of minimizing requirements defects, the trust relationships were built. All requirements mentoring sessions were conducted via email and phone.

TABLE VIII: REQUIREMENT DEFECT DENSITY SRS2

Revision	# of Defects	# of Pages	Defects/ Page (DPP)	% Change in DPP
0.3	275	60	4.58	
0.4	350	78	4.49	-2%
0.5	675	125	5.40	+20%
0.7	421	116	3.63	-33%
0.75	357	119	3.00	-17%
1.0	115	122	0.94	-69%
Overall % change in DPP revision 0.3 to 1.0: ***-79%***				

The requirements defect density for SRS3 appears in Table IX. It was initially composed by Author3. Due to a significant increase in functionality and requirements requests from members of the cross functional team, the first release of SRS3 had almost triple the number of pages as the final release of SRS2. The initial defect density for Author3 was 3.67 DPP, which reflected the good foundation of requirements that the first two requirements authors had provided. With mentoring, this rate went down to 2.54 DPP at revision 0.5 (about a 31% decline).

Starting at revision 0.5, over a dozen other authors started contributing requirements to the SRS. Those authors located in the U.S. received the requirement writing training prior to entering requirements into the database. Those authors located elsewhere in the world were not trained. Again, the consequence of having untrained authors writing requirements is apparent. While the defect density dropped by 31% from revision 0.3 to revision 0.5 (as the requirements SME mentored Author3), it rose by 9% when the new authors contributed requirements for revision 0.6.

An intensive mentoring period ensued that focused on the large number of open defects (830 in total). The requirements SME scheduled phone meetings with the domestic authors. Due to the time zone differences, most of the mentoring with the international authors was done primarily via email. Requirements defects were identified and an explanation was provided as to the nature of the problem. Any defects that could not be resolved via email were eventually addressed with a phone meeting. While perhaps not as effective as one-on-one calls, the email mentoring was successful in reducing the number of defects from 830 to 212 from revision 0.6 to 0.68 (an almost 75% decrease). Overall, the requirements defect density for SRS3 dropped from 3.67 DPP at revision 0.3 to 0.40 DPP at revision 1.0 (an 89% decrease), despite the large influx of authors. At the original 3.57 DPP rate, the final 425 page document would have had over 1500 defects versus the actual number of 172. Mentoring continued to be very effective in reducing requirements defects.

TABLE IX: REQUIREMENT DEFECT DENSITY SRS3

Revision	# of Defects	# of Pages	Defects/ Page (DPP)	% Change in DPP
0.3	1126	307	3.67	
0.5	750	295	2.54	-31%
0.6	830	300	2.77	+9%
0.65	335	298	1.12	-60%
0.67	212	377	0.56	-50%
0.80	177	404	0.44	-21%
1.0	172	425	0.40	-9%
Overall % change in DPP revision 0.3 to 1.0: ***-89%***				

VI. CONCLUSIONS

This multi-year study yielded three key results. First, limited training alone is not sufficient to take untrained requirements authors and turn them into authors capable of writing high quality software requirements specifications. There is simply too much information for them to absorb and apply in a one or two day course. Second, mentoring, when combined with training, is effective in quickly correcting bad writing habits. The focus on requirements defect prevention yields dramatic reductions in overall defect density rates within several document revisions. Third, distance is not a barrier to mentoring. Excellent results can be achieved even with thousands of miles and double digit time zone differences separating the mentor from the mentee.

To the inexperienced requirements author, training on best requirements writing practices can be like “trying to drink from a fire hose”. There are so many new concepts presented, rules to follow and syntaxes to adhere to that the student may be overwhelmed and unable to fully apply all

the concepts. In this study, the defect rates for the three lead requirements authors at their initial SRS release were 10.06, 4.58 and 3.67 DPP (the defects rates for Author2 and Author3 are actually higher if the number of defects and pages that existed in the prior revision 1.0 documents are removed). Without mentoring, these authors would have produced final versions of their software requirements specifications with hundreds to thousands of defects. A significant percentage of these requirements defects would have appeared as code defects.

The impact of mentoring to supplement training is immediately evident in the defect density data. Author1 demonstrated a 50% defect density reduction in the first revision following the start of mentoring and a 98% drop by revision 1.0. Author2 and Author3 showed decreases of 2% and 31% respectively in their first revisions with mentoring. The defect density rate for the four new requirements authors on the second SRS declined by a collective 33% following mentoring. Similarly, there were reductions of 60% and 50% in the DPP rates for the requirements authors on the third SRS after engaging with the requirements SME. The benefits of this defect prevention focus were exemplified by the final defect density rates of less than 1 DPP at revision 1.0 for all three documents.

As noted, all requirements mentoring sessions were conducted remotely. Requirements authors were scattered across the United States and several other countries. Most of the lead authors were located several thousand miles and three time zones away from the requirements SME, so frequent in-person meetings were not economically feasible. When the international sites were added, travel was not an option. Hence, the majority of the mentoring time was conducted via the telephone. Despite the lack of direct contact, dramatic decreases in SRS defect density rates (>79% in each case) were made in all three documents.

This paper has provided data demonstrating the benefits of combining requirements SME mentoring to supplement classroom requirements training in order to produce higher quality software requirements specifications. Even with classroom training, inexperienced authors will continue to inject defects into their requirements. In a SRS with several hundred pages, a requirements defect rate of between 5-10 DPP will result in thousands of defects. Ultimately, these defects will need to be corrected in the software at a much higher cost than correcting them in the requirements phase. Requirements mentoring, which focuses on defect prevention through early reviews, is a cost effective way of improving SRS quality. This is a process requiring human interaction and evaluation. While word processors can be used to detect some defects (e.g., weak words or unbounded lists), the majority of the defect detection must be done by a requirements SME using established criteria. The benefits of fewer requirements defects will lead to less project rework and ultimately to improved overall software quality.

ACKNOWLEDGEMENTS

The author would like to acknowledge Erik Simmons, who authored the Intel requirements training course materials referenced in this paper (sections available from several conference proceedings including the 2011 Pacific Northwest Software Quality Conference [6]) and Bob Bogowitz and Sarah Gregory for their contributions to the review of this paper.

REFERENCES

[1] J. Terzakis, "Requirements defect density reduction using mentoring to supplement training," Proceedings of the Seventh International Multi-Conference on Computing in the Global Information Technology (ICCGI 2012), 2012, pp. 113-114.

[2] B. Berenbach, J. Kazmeier, D. Paulish, and A. Rudorfer, Software & System Requirements Engineering in Practice, McGraw Hill, March 26, 2009.

[3] IEEE Std 830-1998, "IEEE recommended practice for software requirements specifications," the Institute of Electrical and Electronics Engineers, Inc., June 25, 1998 .

[4] K. Wiegers, Software Requirements, 2nd Edition, Microsoft Press, March 26, 2003.

[5] G. Kotonya and I. Sommerville, Requirements Engineering: Processes and Techniques, John Wiley & Sons Ltd., August 25, 1998.

[6] E. Simmons, "21st century requirements engineering: a pragmatic guide to best practices," Proceedings of the 2011 Pacific Northwest Software Quality Conference (PNSPC), 2011, pp. 21-40.

[7] T. Gilb, Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage, Butterworth-Heinemann, June 25, 2005.

[8] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "EARS (Easy approach to requirements syntax)," Proceedings of 17th International Requirements Engineering Conference (RE '09), 2009, pp. 317-322.