# The First Requirements Elucidator Demonstration (FRED) Tool


## ABSTRACT

The problem of poorly-written requirements has been documented for at least the last ten years without much improvement. This paper describes applying technology in the form of a simple software tool containing an Expert System that can minimise the production of poorly worded requirements in the future and help ensure the completeness of the testing of poorly worded requirements. This paper also describes how the tool evolved and discusses how the tool can be used to prevent cost and schedule escalations due to some types of poorly worded requirements.


## 1 Introduction

The expensive cost and schedule impacts resulting from poor requirements have been repeatedly documented for at least 10 years (Hooks 1993; Kasser and Schermerhorn 1994; Standish 1995; Jacobs 1999; Carson 2001; etc). However, the continual documentation of the problem and its attributes has not resulted in a practical solution. Even requirements management tools, have, in the main, avoided dealing with the problem of poorly-written requirements. The problem of poorly-written requirements thus seems to be in the category of those problems for which a complete solution cannot be found, and consequently tends to be tolerated.

This paper describes a partial solution to the problem of poorly-written requirements by combining the principles of Total Quality Management, expert systems, and knowledge management. The part of the problem addressed with some degree of success, is the common practice of producing documents containing requirement statements that

- are poorly worded so as to be vague and unverifiable, and
- contain multiple requirements in a single paragraph, which complicates the traceability of tests to requirements as discussed below.

The format and characteristics of good and bad requirements have been published for years as discussed below, yet people are still producing poorly worded requirements because there does not seem to be any incentive for producing well-written requirements.

A solution to part of the problem of poorly-written requirements is implemented in a simple software tool. This tool, named the First Requirements Elucidator Demonstration (FRED) can

- Assist in preventing the production of poorly-written requirements for future systems;
- Minimise the impact on system testing based on defective requirements in existing requirements documents.

FRED contains an Expert System that performs syntactic processing on text mode requirements and notifies the user when characteristics of poorly-written requirements are present. It is then up to the user to determine of a defect exists and take corrective action.

### 1.1 Good and bad requirements

There seems to be little consensus on what constitutes the correct wording for requirements. However, Hooks (1993) stated that a good requirement states something that is necessary, verifiable, and attainable. To be verifiable, the requirement must state something that can be

verified by examination, analysis, test, or demonstration. Statements that are subjective, or that contain subjective words, such as “easy”, are not verifiable. Hooks also stated that in a specification, there are terms to be avoided and terms that must be used in a very specific manner. Authors need to understand the use of the words “shall”, “will”, and “should”. According to Hooks,

- requirements shall use the word “shall”;
- statements of fact shall use the word “will”; and
- goals shall use the word “should”.

Hooks then writes that

> *“These are standard usage of these terms in government agencies and in industry. You will confuse everyone if you deviate from them.”*

However not everyone agrees. For example, there seem to be various opinions on this subject in the United States of America (USA). The military standard in the form of MIL-STD-961D[1] (1995) long used as a ‘standard practice’ states that the word “shall” defines a requirement. On the other hand, the Federal Aviation Agency (FAA) has a plain language initiative and published guidelines for writing documents in plain language (FAA 2000). The document states

> *“Shall” is one of those officious and obsolete words that has encumbered regulations and other documents for many years. The message that “shall” sends to the reader is, “this is boring material.” “Shall” is imprecise. It can indicate either an obligation or a prediction. Dropping “shall” is a major step in making your regulation more reader-friendly many agencies already use the word “must” to convey obligations with no adverse legal effects.*
>
> *You can avoid “shall” by substituting “must” to indicate an obligation or “will” to indicate that an action will occur in the future. Be careful to consider which meaning you intend to communicate to your readers.*

Kasser and Schermerhorn (1994) showed how poorly worded requirements were able to easily add $500,000 to a project’s cost and identified an initial set of requirements for writing requirements to prevent that cost escalation. These requirements were later updated and published (Kasser 1995) as:

1. [The imperative construction section of a requirement] shall be written so as to be:
   - **Complete** - One of the basic categories of metrics.
   - **Testable** - If you can't test it, you can't demonstrate compliance.
   - **Relevant** - If it's not relevant to the system mission, it's not a requirement. This requirement is present so that (1) inherited requirements are considered carefully before acceptance, and (2) people’s wish lists are not accepted without discussion.
   - **Achievable** - If you can't meet it, don't bother to write it as time will be wasted trying to test it, or discuss it during the later stages of the System Life Cycle (SLC).

[1] While the Military Standards are no longer mandatory, some of them are still in use as ‘standard practices’ or have been incorporated in industrial standards. In any event, even of they are no longer mandatory; they still contain a vast amount of useful and applicable information.

- **Allocated as a single thought to a single requirement** - Each paragraph must also have a unique section number. This simplifies determination of completeness, ensuring compliance and testing.
- **Grouped by function** - Simplifies determination of completeness, ensuring compliance and testing.
- **Traceable;** both upward back to the source, and downwards into lower-level documents.

2. To simplify determination of completeness, ensuring compliance and testing, Requirements **shall not** be written so as to be:
   - Redundant
   - Overlapping
   - Vague
3. In keeping with this theme, the following [poor] words shall never appear within the text of a requirement:
   - **Including**, **i.e.**, **e.g.**, **etc.** - These words imply that the requirement is a subset of an unspecified (and consequently un-testable) superset. You may use these words to provide background information in the document. However, in the requirement, spell out each instance. Don't leave anything to the imagination.
   - **will** - A descriptive word to be used in the extrinsic information to provide background to the requirement when describing what the system will do once it is built.
   - **must** - It's an instruction not a requirement.
   - **should** - The conditional tense. It's a goal not a requirement.

Kar and Bailey (1996) supported these requirements but restated two of these requirements as desired characteristics,

- Necessary - relevant.
- Consistent - not redundant and not contradictory.

They also added other desired characteristics including

- Rationale for the requirement.
- Verification methodology.
- Risk.
- Implementation-free.

Kasser (2000) studied all of these characteristics in the context of managing change over the SLC and defined the following set of Quality System Elements (QSE):

- **Unique identification number** - the key to tracking.
  - **Requirement -** the imperative construct statement in the text mode, or other form of representation.
- **Traceability to source(s) -** the previous level in the production sequence.
- **Traceability to implementation -** the next level in the production sequence. Thus requirements are linked to design elements, which are linked to code elements.
- **Priority -** knowing the priority allows the high priority items to be assigned to early Builds, and simplifies the analysis of the effect of budget cuts.
- **Estimated cost and schedule -** these feed into the management plan and are refined as the project passes through the SDLC.
- **The level of confidence in the cost and schedule estimates** - these should improve as the project passes through the SDLC.

- **Rationale for requirement -** the extrinsic information and other reasons for the requirement.
- **Planned verification methodology(s) -** developing this at the same time as the requirement avoids accepting requirements that are either impossible to verify or too expensive to verify.
- **Risk -** any risk factors associated with the requirement.
- **Keywords -** allow for searches through the database when assessing the impact of changes.
- **Production parameters -** the Work Breakdown Structure (WBS) elements in the Builds in which the requirements are scheduled to be implemented.
- **Testing parameters -** the Test Plans and Procedures in which the requirements are scheduled to be verified.
- **Traceability sideways to document duplicate links -** required when applying the QSE to an existing paper-based project.

However, Carson (2001) still addresses some of the issues associated with poor requirements. So, nothing changes as the years pass. The problem of poorly-written requirements seems to be in the category of those problems for which a complete solution cannot be found, and consequently tends to be tolerated.

# 2 An approach to solving the problem

The approach presented in this paper is to apply technology in the form of a software-based Expert System function that performs syntactic processing on requirements and notifies the user when the syntactic elements of the requirements for writing requirements are violated and “poor words” are found, together with the potential consequences of the use of the specific “poor words”.

The functionality of testing for poor requirements seemed to be an obvious candidate for incorporation in requirements management tools. However, it was noted that modern requirements management tools in general, did not seem to contain the functionality although the idea is not new. Indeed, several papers have been published on the topic as the following examples show

## 2.1 A methodology for deriving quantitative measures of quality of a software specifications document

Kenett (1966) provided a methodology for deriving quantitative measures of quality of a software specifications document by focussing on completeness, readability and accuracy. The methodology parsed the program performance and interface requirements into data categories, provided by a template, then processed the parsed data to provide a set of statistics about the document. A set of specification metrics was then computed to provide an overview of the quality of the document for presentation to management. While the methodology was presented there was no mention of any software employed to perform the functions.

## 2.2 Linguistic Engineering for Software Development

Alvarez, Castell and Slavkova (1996) discussed the Linguistic Engineering for Software Development (LESD) project. The LESD architecture comprised two parts: a syntactic-semantic and domain analysis of the specification, and reasoning mechanisms relating to the representation of requirements. While LESD did develop syntactic and semantic analysis tools, the focus of the research was the determination of metrics for requirements documents

based on traceability, modifiability, completeness, consistency, and verifiability of requirements.

## 2.3 NASA's Automated Requirements Measurement Tool

Wilson, Rosenberg and Hyatt (1997) discuss the development of an Automated Requirements Measurement (ARM) tool by the Software Assurance Technology Center at the Goddard Space Flight Center. The ARM tool focuses on the grammar of the sentence and the use of "weak phrases" or "poor words" such as "large", "rapid" and "many". The ARM tool does not attempt to assess the correctness of the document, it assesses the structure of the requirements document and the vocabulary used to state the requirements based on the desirable characteristics for requirements specifications (IEEE 830-1993), namely

- Complete;
- Consistent;
- Correct;
- Modifiable;
- Ranked;
- Traceable;
- Unambiguous; and
- Verifiable

The tool provides reports on the

- **Text structure** – the number of statement identifiers found at each hierarchical level of the document which provides an indication of the document's organization, consistency, and level of detail.
- **Specification depth** –the number of imperative statements found at each of the document's levels of text structure which can be used to provide an indication of how concise the document is in specifying requirements.
- **Readability statistics** – the four readability statistics provided by Microsoft Word, namely the Flesch Reading Ease index, the Flesch-Kincade Grade Level index, the Coleman-Liau Grade Level index, and the Bormuth Grade Level index.

The ARM Tool has evolved, is still in use and can be downloaded via (ARM 2003).

## 2.4 The Quality Gateway

Robertson and Robertson (1999) described a Quality Gateway that tested requirements for:

- Completeness
- Traceability
- Consistency
- Relevancy
- Correctness
- Ambiguity
- Viability
- [Not] Solution-bound
- [Not] Gold plated, and
- [Lack of] Requirement creep.

However, the Quality Gateway was based on a template rather than a tool.

### 2.5 The Precept Counsellor

James (1999) described the concept of a tool that could provide rapid and early feedback on the overall "goodness" of requirements during the elicitation process. The concept was called the Precept Counsellor and seems to have remained a concept.

### 2.6 A Program to Identify Potential Ambiguities in Requirements Written in English

Bellagamba (2001) described an approach to identifying ambiguities in requirements implemented in *Mathematica*, an interpreting language, because of its string-matching capabilities. While a useful prototype, the functionality does not seem to have been transferred into something useable in academia and industry.

## 3 The knowledge management problem

Given that the both functionality of the syntactic checker and the tool approach wasn't new, why were commercial requirements management tools ignoring the problem? The problem of poorly-written requirements should be solvable using a knowledge management approach and the use of an Expert System approach. After all, the problem is known, the knowledge of much of what needs to be done to correct it is known, the solution should just be a case of applying knowledge management principles. However, after some research into knowledge management it was found that the knowledge management community are facing the same types of complex and ill-structured problems as system engineering's problem of poorly-written requirements. For example, Kemp et al. (2001) wrote that the knowledge management community needed to address several essential issues immediately, including:

- **A systems approach.** Typically, knowledge management programs focus on narrow solutions. A holistic approach is needed.
- **An evolutionary process.** There is no currently accepted process model that supports the continued evolution of knowledge management capabilities within the organisation.

Systems engineering can provide a systems approach to the problem, and this article contains a perspective on developing an evolutionary approach as described below.

## 4 The development process

The traditional systems development process is a serial process in which the requirements (to implement a solution to a problem) are first identified. The system is then designed, implemented, tested, and delivered to the customer. In other words, systems engineering provides a solution to a structured problem. The problem of poorly-written requirements is complex and ill structured and is not amenable to the traditional systems engineering process. Other life cycles that use requirements to drive the design of systems also suffer from the problem of poorly-written requirements to varying extents. That is one reason why Kasser (2002) suggested that object-oriented systems engineering could eliminate the need for requirements. However, for those who still use requirements, the problem posed by poorly-written requirements may be solvable by combining systems engineering and knowledge management. One aspect of dealing with these complex and ill-structured problems is that an accumulation of knowledge in the system can often clarify the problem (Nii 1986). Nii suggests:

> *"Consequently a knowledge engineer needs to engage in exploratory programming, Exploratory programming, as defined by Beau Sheil (1983) is a 'conscious intertwining of system design and implementation.' Waterman (1985) notes that expert system*

*building is accomplished in developmental stages ranging from research prototype to filed prototype until a fielded systems is evolved. That is, expert systems are also developed incrementally."*

An evolutionary systems approach based on the Blackboard methodology pioneered by Nii was taken to solving the problem of poorly-written requirements. The approach is reductionist, namely to first provide a solution for a part of the problem, and then provide a solution for another part, and so on, until the entire problem (or at least a major part of it) is eventually solved. The evolutionary process is shown in Figure 1. The process begins with users interacting with information as shown in Figure 1a. The knowledge is in the users. The situational Use Case is monitored and once understood, a software tool or agent can be created to automate a subset of the process, namely the activities that are performed repeatedly as shown in Figure 1b. The evolution of the system can then be thought of as the migration of knowledge from the users to the tools or agents. FRED is such a tool.


**Figure 1a Start of Evolutionary Process for System Automation**

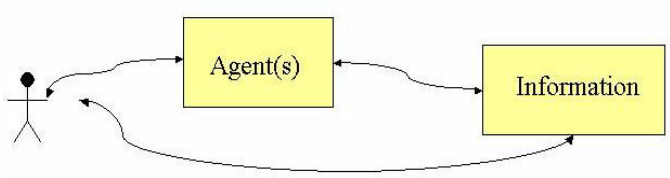


**Figure 1b Software Agents Perform Some Automated Functions, the User the Others.**

**Figure 1. Evolutionary Process for System Automation**

## 5 Poorly-written requirements

Written requirements that do not meet the "requirements for writing requirements" are poorly-written requirements – by definition. At this time, a number of "poor words" that characterise poorly-written requirements can be identified from published papers (e.g., Hooks 1993; Kasser 1995; Kar and Bailey 1996; Wilson et al. 1997, etc), and legacy corporate documents. The list of "poor words" grows by experience. Any time the Test and Evaluation (or the Verification and Validation) function have to clarify a word, it is a candidate for the "poor words" list on future programs. Five categories of poor words are shown in Table 1, and the effect they have on the systems and software engineering process is discussed below. A sample subset of "poor words" found in requirements documents is shown in Table 2.

| Category | Defect |
|---|---|
| 1 | Multiple requirements in a requirement |
| 2 | Possible multiple requirement |
| 3 | Not verifiable |
| 4 | Use of wrong word |
| 5 | User defined poor word |

**Table 1 Categories of defects in Requirements**

| Poor Words | Occurrence | Category of Defect |
|---|---|---|
| Adequate | 0 | Descriptive, not verifiable |
| And | 0 | Possible multiple requirement paragraph |
| Appropriate | 0 | Descriptive, not verifiable |
| Best practice | 0 | Descriptive, not verifiable |
| But not limited to | 0 | Unspecified superset, not verifiable |
| Easy | 0 | Descriptive, not verifiable |
| For example | 0 | Descriptive, not verifiable |
| Including | 0 | Unspecified superset, not verifiable |
| Large | 0 | Descriptive, not verifiable |
| Many | 0 | Descriptive, not verifiable |
| Maximize | 0 | Descriptive, not verifiable |
| Minimize | 0 | Descriptive, not verifiable |
| Must | 0 | Use of wrong word |
| Or | 0 | Possible multiple requirement paragraph |
| Quick | 0 | Descriptive, not verifiable |
| Rapid | 0 | Descriptive, not verifiable |
| Shall | 1 | Multiple requirements in requirement |
| Should | 0 | Use of wrong word |
| Sufficient | 0 | Descriptive, not verifiable |
| User-friendly | 0 | Descriptive, not verifiable |
| Will | 0 | Use of wrong word |

**Table 2: Partial List of "Poor Words" in Requirements**

While some of these words may have meanings, which can be carefully defined, in most instances they are not carefully defined, so it is better to avoid them, and use words that have verifiable meanings. Still, as there are exceptions to each of these rules, the tool needs to contain a mechanism to allow a specific "poor word" in a specific requirement to be ignored during the syntactic processing process. The tool also needs to be able to identify paragraphs that are descriptive and provide background information to the requirements.

## 5.1 Multiple requirements in a requirement

More than one requirement in a requirement complicates the building of traceability matrices as shown below, and causes difficulties in the test and evaluation processes. By definition, if the word "shall" appears more than once in a requirement, there are multiple requirements in the requirement and hence this situation is a defect in the requirement. Correcting this defect means splitting the multiple-requirement into separate individual requirements.

## 5.2 Possible multiple requirement

Words such as "and" and "or" indicate that there may be multiple requirements in a requirement. The tool identifies this condition. For example, consider the use of the word "and" in a requirement. A requirement such as

"DADS shall display the number of requests pending **and** requests processed"

is two requirements, the first to display the number of requests pending, and the second to display the number of requests processed. This is a defect that must be corrected by writing the requirement as two requirements. On the other hand, a requirement such as

```
"DADS shall display the combined number of requests pending and requests
processed"
```

is a requirement to display a single total, hence the use of the word "and" is appropriate. This is not a defect and the tool can be notified of that fact by means of associating a 'poor word override' flag with the requirement in the requirements database or document.

### 5.3 Not verifiable

Some words are not verifiable or testable. Words that fall into this category include – large, many, few, including, etc. Some words may be understood at the time and in the context that they are used, but have no meaning further down the schedule when the requirement writers have moved on to other projects. The author encountered "statistically monitor" as but one example (ST-DADS 1992). The tool points this situation out to the writers who then must clarify the meaning.

### 5.4 Use of wrong word

MIL-STD 961D (1995) requires the use of the word "shall" to identify a requirement. The words "will", "should", and "must", may be valid in descriptions but are not to be used in requirements. The tool can point out this situation and advise the user to change the word.

### 5.5 User defined poor word

This category is provided to allow the user to define words that are "poor words" in the user's organisation.

## 6 The Requirements Workshop

The tool performing the syntactic processing function automates much of the activities in Requirements Workshops held in courses leading to the Master of Software Engineering (MSWE) and Master of Computer Systems Management (CSMN) degrees at University of Maryland University College (UMUC) in 1998, 1999 and 2000. The postgraduate students took part in a Requirements Workshop module, which was incorporated in one of the classes in three different courses (software requirements, software verification and validation, and software maintenance). The hypothesis for the workshop was that poorly articulated requirements could be prevented from being written once a set of "poor words" had been identified, and the students could discuss the "poor words" and their effect on projects. Thus, the focus of the workshop was on preventing the production of, and elucidating existing, poorly-written requirements in the documents. During the workshop, the students were presented with requirements document and they had to seek out the "poor words" and evaluate the quality of the document, i.e., they performed the syntactic analysis manually.

The workshop first ran in the requirements course. Students evaluated a document provided by the instructor and then produced requirements documents. Students in subsequent iterations of the workshop then evaluated the requirements document from different perspectives. Students in the course covering the requirements phase of the SLC had to produce requirements documents. The workshop helped to prevent them from writing poorly worded requirements. Students in the classes covering software design, testing, and maintenance, then used these requirements documents (after the deletion of author information) as inputs to their projects (Kasser and Williams 1999). Thus, students in the course covering phases of the SLC subsequent to the requirements phase (design, construct, test and operations and maintenance) experienced the effect of the "poor words" on their projects. The workshop discussions in the software requirements class focussed on the format

and structure of a requirement. However, the discussion the other classes focussed on how to deal with the problem of the poorly worded requirements documents[2] in their project as well as what constituted a poorly worded requirement. Most of the students wanted to rewrite the document and remove the defects. However, in the real world that option is not always available and other techniques have to be employed that effectively perform the same function.

Since the students could take the courses in any order, it was undesirable to use the same requirements document for each workshop. Thus, each requirements course was expected to produce a new set of documents suitable for use in the workshops in the other courses in subsequent semesters. However, the results of the workshop approach were effective and impressive, the quality of the student-produced requirements documents improved over several semesters to the point where the documents were no longer useable as examples of poorly-written requirements documents. The in-class dialogue during the requirements workshops at UMUC mirrored the dialogue that James (1999) provides in examples of how the Precept Counsellor could be used. The results achieved in the workshop thus support James' claims for the effectiveness of the concept.

## 7 The operations concept for the tool

An operations concept for a simple stand-alone tool that would automate the syntactic analysis performed in the Requirements Workshop by performing the following functions was developed based on the following Use Cases.

1. Extracting a set of requirements from DOORS (as a specific example of a requirements management tool), or from documents written in Microsoft Word format.
2. Feeding each requirement into a text parser performing syntactic analysis which matches each word in the list of "poor words" against the requirement.
3. Producing a report documenting each occurrence of a poor word.
4. Producing a Figure of Merit (FOM) for the document. The FOM is a simple one-dimensional measurement for the quality of a document based on the presence or absence of "poor words". The FOM allows comparisons to be made of the quality of documents of different sizes. The FOM was calculated using the formula

   FOM = 100 - (number of defects / number of requirements) * 100

   This formula results in a FOM of 100 for a document that contains zero defects, and a negative number for a document containing more defects than requirements since a single requirement may contain more than one defect. While other research efforts as described above have provided more statistical information about the quality of requirements, the FOM is all that is needed to elucidate the "poor words" in the requirements.
5. Storing the "poor words" in a table that can be edited by the user without reprogramming the tool, so as to allow the user to add new "poor words" as, and when, they are identified.

[2] These documents were based on student-produced documents in the class on requirements. The instructor removed references to authors and combined sections from different documents to preserve the confidentiality of student information.

## 8 Implementation decision

The implementation choices were between a stand-alone tool and an "add-on" to a requirements management tool such as DOORS. The stand-alone tool implementation was chosen for several reasons including

- There would be no need to obtain licenses for a requirements management tool.
- The focus is on the syntactic analysis. Current requirements engineering tools have a long learning curve, which sets a high threshold to be overcome when arguing for their adoption. Tools need to be simple to be used widely. The goal for the tool was that it should be no more difficult to use than a slide rule. That means that some functionality could be used with a minimal amount of learning and some functionality might require further learning about the capability of the tool.
- Building the tool as add-on functionality to a specific requirements management tool would preclude users of other requirements management tools from being able to use the functionality.

## 9 The First Requirements Elucidator Demonstration Tool

FRED was written in Borland's Delphi (Visual Pascal) in the FBRET context (Cook et al. 2001) using the methodology and architecture proposed by Kasser (1997). The implementation constraint based on Use Case 5, was to develop a table-driven approach, so that each user could extend the vocabulary of "poor words" without any further programming. The initial extendable table of "poor words" against which all the requirements had to be tested was based on Table 1. The table contained the following information

1. The "poor word".
2. The number of times the word was allowed to appear in the requirement. Except for the word "shall", which has to be present once to signify a requirement, the number of allowable occurrences was 0.
3. The category of defect associated with the "poor word". Categories of defects were as described above.

FRED was simple to construct. The hardest part of the implementation was determining a table-driven approach for storing the "poor words". FRED, being a prototype tool, displays status information and shows each requirement being parsed, as well as the summaries. A typical example of FRED is shown in Figure 2. FRED performs a syntax check on the text of the requirement pointing out potential defects in the same manner that a word processor grammar checker operates. In many instances, it is up to the user to determine if the defect is there and how best to correct it. The top window shows the requirements as they are read from the document; the lower window contains the report. The window in Figure 2 shows the report for the following requirement.

```
509.1 DADS shall monitor and provide reports (to the operator) on all
requests for DADS products and services. This capability shall include
recording the name and organization of the requester, the product or
service requested, the date and time of the request, the service priority,
the current disposition of the request, and the date and time of Service
completion.
```

FRED has pointed out 10 possible defects in the requirement that are highlighted below.

```
509.1 DADS shall monitor and provide reports (to the operator) on all
requests for DADS products and services. This capability shall include
recording the name and organization of the requester, the product or
```

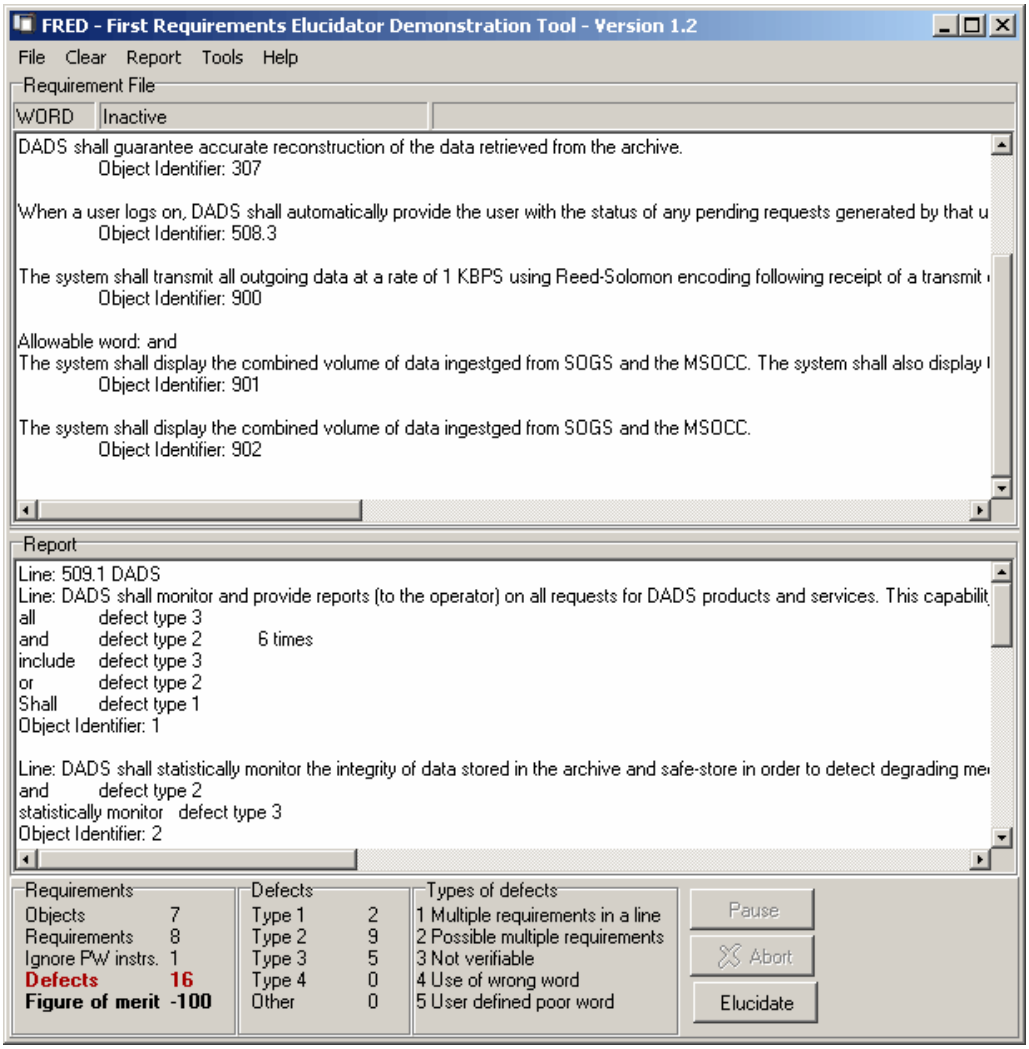


**Figure 2 Typical FRED Main Display**

```
service requested, the date and time of the request, the service priority,
the current disposition of the request, and the date and time of Service
completion.
```

The tools window in FRED is shown in Figure 3. The poor words and their associated defects can be seen. Other areas of the window allow customisation of files and parameters.

## 9.1 Fred's role in systems engineering

FRED is not a management or reporting tool in the manner of the ARM Tool. FRED is a prototype computer-enhanced systems engineering tool (Kasser 1995) that can be used in two roles. The first is to provide immediate feedback to requirements writers that the document that they are compiling contains poorly worded requirements in the manner of the ARM Tool. In this mode it helps prevent the publication of poorly-written requirements. In the second role, by identifying the individual requirements in a multiple-requirement paragraph of a Requirements Document, FRED can be used to ensure that existing requirements can be adequately tested. Consider the following examples of Fred's role.

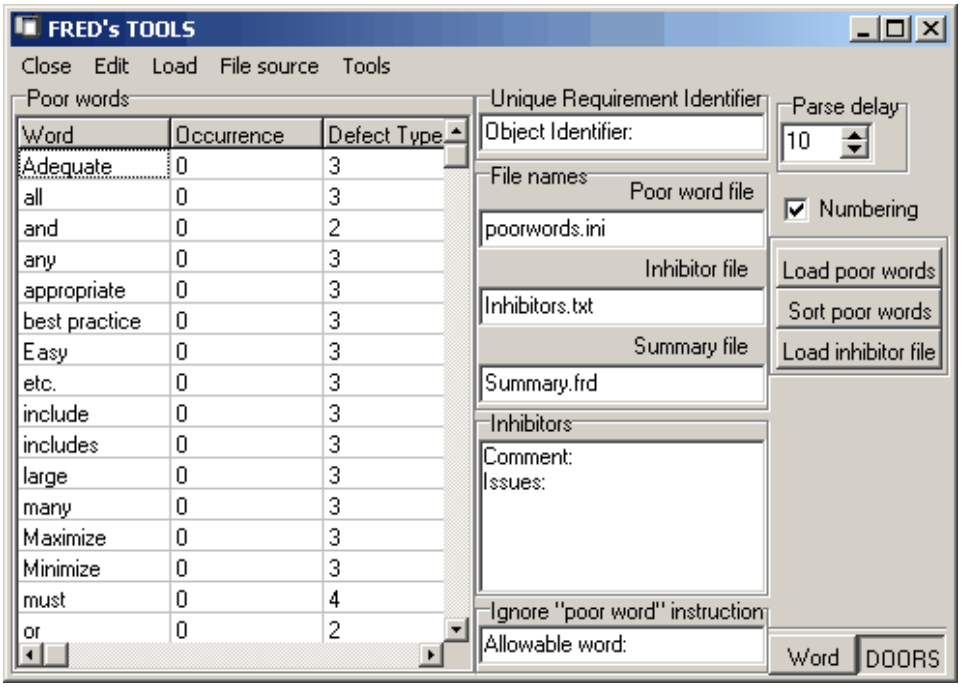


**Figure 3 Typical Tools Display**

### *9.1.1 Preventing poorly-written requirements*

The experiences in the requirement workshop at UMUC show that once sensitised to the effect of "poor words", students produced documents without those specific "poor words". FRED can be used to sensitise project personnel to "poor words" before they produce requirements documents. This is the Precept Councillor mode (James 1999). FRED can be also be used in classes in systems and software engineering for the same purpose.

### *9.1.2 Contractually mandating Quality in Requirements Documents*

The functionality in tools like FRED can be thought of as technology enabled product standards. From this perspective, Fred's functionality can be thought of as a technology enabled version of a sub-section of MIL-STD-961 or IEEE 830. Hence these types of tools could be used in the manner that the MIL-STDs were used to attempt to insert quality into a product. For example, FRED could be used to prevent poorly-written requirements by mandating in the contract that all requirement documents produced under that contract shall have a FOM of 100 for a given set of "poor words" and a specified release of FRED. Consider the cost and schedule impact of such a contractual mandate. Kasser and Schermerhorn (1994) provided one example of the cost of defective requirements documentation based on the formal and informal meetings during the life cycle of a typical large project resulting from trying to interpret a single defective document. If you multiply the time spent in these meetings, by the number of meetings and the numbers of attendees, the unplanned labour cost of these meetings can very quickly reach $500,000 or so over the course of the project. Now multiply that by the number of defective documents in a large project, and think about the effect on the project budget and schedule. This example does not consider that the discussions elucidating the requirements can also help to identify some missing and

wrong requirements. Incorporating FRED as a simple extension to today's system and software engineering paradigm alone has the potential to both save millions of dollars and prevent schedule escalations.

### *9.1.3 Quick review of documents*

FRED is useful for people who have to review requirements documents in short-turn around situations such as just before awarding contracts, as well as for those persons who are writing the documents. The FOM can be determined as part of the review process to give a rapid determination of the quality of the document. A low FOM could indicate that the producers of the document do not understand the need. The process of clarification of the vagueness would also help identify some of the missing and wrong requirements.

### *9.1.4 Ensuring the completeness of testing requirements contained in multiple requirement paragraphs*

FRED could be used to increase the effectiveness of the planning and testing of a system with defective requirements. This example is based on the approach used to plan the testing of Build 3 of the software for the Hubble Space Telescope Data Archiving and Delivery Service (DADS) processing centre in 1993. The tests were planned so that each test would test a group of requirements. However, the requirements were defective containing many examples of "poor words" and multiple-requirements in a single requirement paragraph. Thus, each requirement allocated to Build 3 had to be evaluated and split into a number of individual requirements to ensure completeness of the testing. Each "poor word" had to be identified and clarified to ensure that all parts of the defective requirements were tested. Some multiple paragraph requirements had to be tested in several different tests. This required unplanned and unbudgeted meetings between the test and design personnel and ways of documenting partial requirements in the Requirements Traceability Matrix. As an example of the work that FRED could have expedited, consider the following requirement (ST-DADS 1992):

*204.1* DADS shall automatically maintain statistics concerning the number of times and the most recent time that each data set has been accessed. These same statistics shall be maintained for each piece of media in the DADS archive.

Two vague phrases had to be clarified. In the absence of any traceability to an operations concept, the following clarifications were made:

- The vague phrase "automatically maintain statistics concerning" was interpreted to mean ONLY the "total number of times" and the "most recent time". Thus, it was interpreted as meaning that there was to be no test to determine if DADS kept a log of access information (times and user) as far as this requirement was concerned.
- The term "piece of media" was interpreted to mean the physical disk on which the data was stored.

The requirement was then split into the following four requirements to simplify tracking the completeness of the test plans:

204.1a DADS shall automatically maintain statistics concerning the number of times ~~and the most recent time~~ that each data set has been accessed. ~~These same statistics shall be maintained for each piece of media in the DADS archive~~.

204.1b DADS shall automatically maintain statistics concerning ~~the number of times and~~ the most recent time that each data set has been

```
        accessed. These same statistics shall be maintained for each piece
        of media in the DADS archive.
204.1c  DADS shall automatically maintain statistics concerning the number
        of times and the most recent time that each data set has been
        accessed. These same statistics shall be maintained for each piece
        of media in the DADS archive [has been accessed].
204.1d  DADS shall automatically maintain statistics concerning the number
        of times and the most recent time that each data set has been
        accessed. These same statistics shall be maintained for each piece
        of media in the DADS archive [has been accessed].
```

Leaving the sections of the requirement that were not being tested in place but stricken through clearly identified which section of the requirement was being tested. An unfortunate side effect was that it also clearly showed the defects in the requirement to the customer and by implication the competence of the project manager who had signed off on the requirements document. Note that the phrase '`has been accessed`' has been moved in the last two sub-requirements to clarify the sub-requirement.

Building the Test Plan was a labour-intensive process. Each requirement had to be manually scanned for vague words, the 'and' and 'or' words, as well as further occurrences of the word "shall". Having a FRED-like function parse the document and identify requirements needing clarification and splitting to ensure a complete test would have saved at least 200 person-hours of test planning effort; and DADS was not a large project!

### 9.2 Passing on lessons learned

FRED can be used to prevent time from being spent on clarifying meanings of poorly worded requirements in future projects. If the vague phrases extracted from requirements documents in one project are added to the list of "poor words", they can be prevented from appearing in future requirements documents. The students at UMUC produced documents that lacked the "poor words" they were shown. However, each iteration of a class produced its own crop of new "poor words". These could be added over time thus institutionalising the lessons learned in the effect of "poor words". Management of the "poor word" file would be a task performed by the Quality Assurance or Test department to ensure that should someone write that phrase in the future, it would be clarified before being adopted.

## 10 Limitations and further evolution

FRED does have a number of limitations. However, FRED is also evolving in the manner of software by the addition of functionality in product upgrades. Identification of these limitations provides areas for further research and subsequent evolution. Consider some of them as described below.

### 10.1 Limitations of FRED

Consider the following requirement

```
509.3 DADS shall notify a user when his request has been completed.
```

Apart from being sexist, the requirement is a good one (if the time between the completion of the request and the generation of the notification is specified elsewhere). However, the intent still needs to be known. Does the requirement only apply when the user is logged into DADS, or is DADS required to send the user an email or other type of notification when the user is not logged into the DADS? On the other hand, was the requirement sexist by virtue of the use of the word "his", or was it meant to only apply to manually generated requests? Is there

a difference between manually and automatically generated requests? This is where traceability to the Use Cases in the operations concept is important. FRED cannot identify this type of defect at present.

### 10.2 A simple tool

FRED is a simple tool and can only parse what it is given. It provided a solution to the partial problem of poorly-written requirements. It does not contain any information about the completeness and the appropriateness of the requirement. Nor does it have the ability to detect conflicts in requirements. Those functions are planned for the future but not as upgrades to FRED. Work is currently under way in the Systems Engineering and Evaluation Centre (SEEC) in building a series of FRED like tools addressing not only those aspects of the problem of poorly-written requirements but also enabling the generation, storage and use of the QSE. These tools are known as Prototype Educational Tools for Systems and software engineering (PETS).

### 10.3 Further evolution

As more is learnt about defects in requirements wording, FRED can be upgraded to incorporate that knowledge, or other simple FRED-like tools could be developed. Future iterations of FRED might evolve into a product having similar features to the commercially available StyleWriter, a program that can analyse a document and highlight all of its faults (at least according to its web site (StyleWriter 2002). Thus, FRED might evolve to

- Ingest the requirements into a QSE database.
- Provide a template for grammatically correct requirements. For example, it might suggest a rewrite of a requirement to begin with “the system shall…’.
- Suggest synonyms for specific poor words, such as replacing each occurrence of ‘should’ and ‘must’ with ‘shall’.
- Suggest ways in which a multiple requirement paragraph may be split into the appropriate number of single requirement paragraphs in the manner shown above.

Suitable interface dialogues would then perform the functions for the user.

## 11 Conclusions

FRED is not a silver bullet that will cure the problem of poorly-written requirements. However, it can prevent many types of poorly-written requirements from being written. FRED has demonstrated that the concept of using an evolutionary approach to developing a simple and useful tool to solve part of the problem of poorly-written requirements is feasible. After parsing documents, FRED produces a report that points out the “poor words” in the requirements. The requirements then have to be analysed and repaired to become specific and verifiable. During the discussions in the process of repairing the requirements, ambiguities were identified and resolved which improved existing documents and led to improved wording and fewer missing requirements in future documents thereby providing a multiplier effect.

Using FRED as a stand-alone tool, or having tool vendors add the functionality provided by FRED to current generation requirements management tools would prevent the costs and delays associated with many poorly-written requirements and result in better-written requirements.

## 12 Postscript

FRED is the first of a proposed suite of PETS for improving the systems and software engineering process, and is available at no cost for educational use. To download a copy, access the SEEC website at http://www.seec.unisa.edu.au and follow the Software Tools link. The plan is to add the other PETS as they are developed.

The author acknowledges the insight and comments on this paper by Professor Stephen C. Cook at the SEEC in the University of South Australia.

## 13 References

Alvarez, J., Castell, N., Slavkova, O., "Combining Knowledge and Metrics to Control Software Quality Factors," *Proceedings of the Third International conference on Achieving Quality in Software*, Florence, Italy, January 1996, pp. 201-12, available at http://citeseer.nj.nec.com/alvarez96combining.html, last accessed January 24, 2003.

ARM 2003, Available at http://satc.gsfc.nasa.gov/tools/arm/", last accessed January 29,2003.

Bellagamba, "Program to Identify Potential Ambiguities in Requirements Written in English", *The 11th INCOSE International Symposium, Melbourne, Australia, 2001.*

Carson, "Keeping the Focus During Requirements Analysis", *The 11th INCOSE International Symposium, Melbourne, Australia, 2001.*

Cook S.C., Kasser J.E. (2001) Asenstorfer, J., "A Frame-Based Approach to Requirements Engineering*", 11th International Symposium of the INCOSE*, Melbourne, Australia.

FAA 2000, "Writing User-Friendly Documents - A Handbook for FAA Drafters February 2000", available at http://www.faa.gov/language/, last accessed October 14, 2002.

Hooks, I., "Writing Good Requirements", *Proceedings of the 3rd NCOSE International Symposium, 1993,* http://www.incose.org/rwg/writing.html, last accessed November 7, 2002.

IEEE 830, Institute of Electrical and Electronics Engineers. Recommended Practice for Software Requirements Specifications (December 2, 1993), IEEE Std 830-1993.

Jacobs, S., "Introducing Measurable Quality Requirements: A Case Study", *IEEE International Symposium on Requirements Engineering*, Limerick, Ireland, 1999.

James, L., "Providing Pragmatic Advice On How Good Your Requirements Are - The Precept "Requirements Councillor" Utility", *The 9th INCOSE International Symposium, Brighton, England, 1999.*

Kar, P., Bailey, M., "Characteristics of Good Requirements*", The NCOSE 6th International Symposium*, Boston, MA, 1996, available at http://www.incose-wma.org/info/se/examples/goodreqs.htm, last accessed December 12, 2002.

Kasser, J.E., Schermerhorn, R., "Gaining the Competitive Edge through Effective Systems Engineering",", *The NCOSE 4th International Symposium*, San Jose, CA., 1994, http://www.seec.unisa.edu.au/people/Jk/Pubs/Gaining.pdf, last accessed December 2, 2001.

Kasser J.E., *Applying Total Quality Management to Systems Engineering*, Artech House, Boston, June 1995.

Kasser J.E. "Does Object-Oriented System Engineering Eliminate the Need for Requirements?", *Proceedings of the 12th International Symposium of the International Council on Systems Engineering (INCOSE),* Las Vegas, NV, 2002.

Kasser J.E., "A Framework for Requirements Engineering in a Digital Integrated Environment (FREDIE)", *The Systems Engineering, Test and Evaluation (SETE 2000) Conference*, Brisbane, Australia, 2000, available at http://www.seec.unisa.edu.au/people/Jk/Pubs/A Frame-based Approach 82.pdf, last

accessed December 16, 2002.

Kasser, J.E., "Yes Virginia, You Can Build a Defect Free System, On Schedule and Within Budget", *The INCOSE 7th International Symposium*, Los Angeles, CA, 1997, available at http://www.seec.unisa.edu.au/people/Jk/Pubs/YesVirginia.pdf, last accessed December, 12, 2002.

Kasser J.E., Williams V.R., "The Student Enrolment and Course Tracking System Meta-Project", *PICMET 1999*, Portland, OR, 1999, available at http://www.seec.unisa.edu.au/people/Jk/Pubs/sects.pdf, last accessed December 12, 2002.

Kemp L.L., Nidiffer K.E., Rose L.C., Small R., Stankowsky M., "Knowledge Management: Insights from the Trenches", IEEE Software, November/December 2001, pp. 66-68.

Kenett, R.S., "Software Specification Metrics: A Quantitative Approach to Assess the Quality of Documents," *Nineteenth Convention of Electrical and Electronics Engineers in Israel*, 5-6 Nov 1996, pp. 166 -169, June 1996.

MIL-STD-961D, Department Of Defense Standard Practice For Defense Specifications, 22 March 1995, Superseding MIL-STD-961C, 20 May 1988

Nii H.P., "Blackboard Systems", Knowledge Systems Laboratory Report No. KSL 86-18, Knowledge Systems Laboratory, Department of Medical and Computer Science, Stanford University, 1986.

Robertson, S., Robertson, J., "Reliable Requirements Through the Quality Gateway", *10th International Workshop on Database and Expert Systems Applications*, Florence, Italy, 1999.

Sheil B., "Power Tools for Programmers", Datamation, pp: 131 to 144, February 1983.

Standish (1995), *Chaos*, The Standish Group, http://www.standishgroup.com/chaos.html, last accessed March 19, 1998.

ST DADS Requirements Analysis Document (FAC STR-22), Rev. C, August 1992, as modified by the following CCR's:- 139, 146, 147C, 150 and 151B.

Waterman D., *A Guide to Expert Systems*, Addison-Wesley, 1985.

Wilson, W.M., Rosenberg, L.H., Hyatt, L., "Automated Analysis of Requirements Specifications," *Proceedings of the IEEE International Conference on Software Engineering*, Boston, MA, May 1997.