SOFTWARE REQUIREMENTS

AND THE ETHICS OF SOFTWARE ENGINEERING

Draft 6.0 July 9, 2012

Abstract

Software requirements are a weak link in the chain of software engineering technologies. Requirements are usually incomplete and change at rates in excess of 2% per calendar month. For many years one common definition of quality has been "conformance to requirements." However this definition ignores the fact that some requirements are hazardous or "toxic" and should not be included in software applications. Since clients themselves may not realize the dangers of toxic requirements, software engineers have a professional and ethical responsibility to point out the hazards of dangerous requirements and ensure that they are safely eliminated. An example of a "toxic requirement" is the famous Y2K problem which did not originate as a coding bug but rather as an explicit but dangerous user requirement.

Capers Jones, Vice President and CTO
Namcook Analytics LLC

Email: Capers.Jones3@Gmail.com
Web: www.Namcook.com

INTRODUCTION

There are a number of "standard" definitions for software quality which have not been rigorously examined for flaws and problems. One of the oldest of these quality definitions is:

"Quality means conformance to requirements."

There are several problems with this definition, but the major problem is that requirements errors or bugs are numerous and severe. Errors in requirements constitute about 20% of total software defects.

Defining quality as conformance to a major source of error is circular reasoning and therefore this must be considered to be a flawed and unworkable definition. Obviously a workable definition for quality has to include errors in requirements themselves.

One definition of quality which can be applied to requirements is:

"*The absence of defects that would cause an application to either stop completely or produce incorrect results."*

This definition would include requirements defects such as the former Y2K bug. Don't forget that the famous Y2K problem originated as a specific user requirement and not as a coding bug. Many software engineers warned clients and managers that limiting date fields to two digits would cause problems, but their warnings were ignored or rejected outright.

Another chronic problem with requirements is "requirements creep" or continued growth after the requirements phase. This is not unexpected because the business world is never static so changes will certainly occur.

However the software industry often is not prepared for the magnitude of changing requirements. The measured rates of change for requirements run from less than 1% per calendar month to more than 4% per calendar month.

For a large application that starts at a nominal 10,000 function points and might have a 36 month development cycle, it is easily possible for another 3,000 function points to be added in mid development, and these will probably add at least six months to the development schedule.

If software engineering is going to become a true profession rather than an art form, software engineers have a responsibility to help customers define requirements in a thorough and effective manner. They also have a responsibility to alert clients to the fact that changes in requirements will occur and need to be handled in an effective manner.

Far too often the literature on software quality is passive and makes the incorrect

assumption that users are going to be 100% effective in identifying requirements. This is a dangerous assumption. User requirements are never complete and they are often wrong. For a software project to succeed, requirements need to be gathered and analyzed in a professional manner, and software engineering is the profession that should know how to do this well.

It should be the responsibility of the software engineers to insist that proper requirements methods be used. These include:

1. Reusable requirements
2. Data mining of legacy applications to extract hidden requirements
3. Joint Application Design (JAD)
4. Quality Function Deployment (QFD)
5. Prototypes – evolutionary
6. Prototypes - disposable
7. Requirements inspections
8. Agile "embedded users"
9. Running readability software tools on requirements such as the FOG index
10. Running text static analysis tools on requirements
11. Requirements standards
12. Requirements change control boards
13. Requirements estimation tools that predict creep and deferred features
14. Requirements estimation tools that predict quality and costs
15. Focus groups
16. Domain specialists for topics such as security, performance, and ease of use
17. Use of graphical techniques such as the unified modeling language (UML)
18. Use of decision tables
19. Use of model or pattern-based requirements
20. Use of test-driven development

The users of software applications are not software engineers and cannot be expected to know optimal ways of expressing and analyzing requirements. Ensuring that requirements collection and analysis are at state of the art levels devolves to the software engineering team.

Three Chronic Requirements Problems

There are three widespread problems with software requirements that need better solutions than are customary for software projects:

1. Many requirements are dangerous or toxic and should be eliminated.

2. Requirements are never complete and grow at rates > 1% per calendar month.

3. Some clients insist on stuffing extra, superfluous features into software.

Software engineers have an ethical and professional obligation to caution clients about these problems and to assist clients in solving them, if possible. In other words, software engineers need to play a role similar to the role of physicians. We have an obligation to our clients to diagnose known requirements problems and to prescribe effective therapies.

Once user requirements have been collected and analyzed, then conformance to them should of course occur. However before conformance can be safe and effective, dangerous or toxic requirements have to be weeded out, excess and superfluous requirements should be pointed out to the users, and potential gaps that will cause creeping requirements should be identified and also quantified. The users themselves will need professional assistance from the software engineering team, who should not be passive bystanders for requirements gathering and analysis.

Unfortunately requirements bugs cannot be removed by ordinary testing. If requirements bugs are not prevented from occurring, or not removed via formal inspections or other methods, test cases that are constructed from the requirements will confirm the errors and not find them. (This is why years of software testing never found and removed the Y2K problem.)

Another issue is that for some brand new kinds of innovative applications there may not be any users other than the original inventor. Consider the history of successful software innovation such as the APL programming language, the first spreadsheet, and the early web search engine that later became Google.

These innovative applications were all created by inventors to solve problems that they themselves wanted to solve. They were not created based on the normal concept of "user requirements." Until prototypes were developed, other people seldom even realized how valuable the inventions would be. Therefore "user requirements" are not completely relevant to brand new inventions until after they have been revealed to the public.

Given the fact that software requirements grow and change at measured rates of 1% to more than 4% ever calendar month during the subsequent design and coding phases, it is apparent that achieving a full understanding of requirements is a difficult task.

Software requirements are important, but the combination of toxic requirements, missing requirements, and excess requirements makes simplistic definitions such as "quality means conformance to requirements" hazardous to the software industry.

Software requirements have many more than three problems of course. Table 1 shows the most common software requirements problems in rank order of seriousness:

Table 1: Overview of Common Software Requirements Problems in Rank Order

1	Toxic requirements that should not be implemented
2	Poor client understanding of their own requirements
3	Disputes between stakeholders about specific requirements
4	Conflicting and mutually contradictory requirements
5	Failure to include security requirements
6	Failure to include quality requirements
7	Arbitrary schedule requirements by clients or executives
8	Requirements defects > 1 per function point or 0.5 per page
9	Requirements gaps and omissions (goes up with size)
10	Requirements creep (> 1% per calendar month)
11	Requirements volume (may exceed lifetime reading speeds)
12	Inadequate change control by stakeholders and developers
13	Inadequate requirements templates that omit key topics
14	Superfluous requirements not needed by users
15	Inadequate requirements inspection methods
16	Inadequate requirements test methods
17	Inadequate requirements change control methods
18	Inadequate requirements gathering methods
19	Inadequate requirements analysis methods
20	Lack of an effective taxonomy for specific software features
21	Lack of automatic function point generation from requirements
22	Lack of effective, certified reusable requirements
23	Inclusion of potentially harmful uncertified reusable requirements
24	Requirement to include COTS packages without due diligence
25	Requirement to include open-source packages without due diligence
26	Poor communication between stakeholders and team members
27	Poor requirements communication among team members
28	Deferred requirements omitted from planned release
29	Poor traceability of requirements to other materials
30	Static representations of dynamic phenomena
31	Poor readability of text portions of requirements
32	Failure to update requirements after initial release
33	Failure to remove cancelled requirements
34	Obscure graphics and visual elements of requirements
35	Evolutionary prototypes may have quality and security flaws
36	Prototypes > 10% of key features may be cumbersome

Table 1 illustrates why requirements are such a tricky and complicated topic for software applications, and especially so for large software applications.

What Kinds of Requirements are most complete?

The word "requirements" is ambiguous and covers a wide range of features for a wide range of software applications. Table 2 shows the completeness ranks of 25 kinds of software requirement origins:

Table 2: Ranked List of Requirements Completeness by Origin

1	Applications invented by one person
2	Applications with certified reusable requirements
3	Applications < 100 function points in size
4	Replacement of a legacy application with no new features
5	Applications using requirements modeling
6	Small projects with embedded users in team (Agile)
7	Projects with stable requirements (any size)
8	Applications with quality function deployment (QFD)
9	Applications requiring FDA, FAA, or government certification
10	Replicating competitive features in commercial packages
11	Medium project with joint application design (JAD)
12	Large system with joint application design (JAD)
13	Applications requiring strict governance
14	Replacing a legacy application + new features
15	Large commercial package with focus groups
16	Embedded application with mixed software/hardware requirements
17	Commercial applications with requirements by marketers
18	Applications associated with frequent hardware changes (defense)
19	Informal interviews with users
20	Applications with complex data (taxation, health insurance)
21	Projects with unstable requirements due to business changes
22	Applications with unpredictable government mandates
23	Applications with clients who disagree about features
24	Applications with > 100,000 users
25	Massive applications > 100,000 function points

As can be seen software applications that are based on the inventions of a single inventor are the most complete, because all of the requirements are known to the inventor.

The least complete requirements are those for massive applications > 100,000 function points where it is probably impossible to know all requirements. Also incomplete are applications with > 100,000 users because it is impossible to know every permutation and combination of features used.

What Goes Into Software Requirements?

Software requirements obviously describe the key features and functions that a software application will contain. But requirements specifications also serve other business purposes. For example the requirements should also discuss any limits or constraints on the software, such as performance criteria, reliability criteria, security criteria and the like.

The costs and schedules of building software applications are strongly influenced by the size of the application in terms of the total requirements set that will be implemented. Therefore requirements are the primary basis of ascertaining software size.

By fortunate coincidence the structure of function point metrics are a good match to the fundamental issues that should be included in software requirements. In chronological order these seven fundamental topics should be explored as part of the requirements gathering process:

1) The *outputs* that should be produced by the application.
2) The *inputs* that will enter the software application.
3) The *logical files* that must be maintained by the application.
4) The *entities and relationships* that will be in the logical files of the application.
5) The *inquiry types* that can be made to the application.
6) The *interfaces* between the application and other systems.
7) Key *algorithms* that must be present in the application.

Five of these seven topics are the basic elements of the International Function Point Users Group (IFPUG) function point metric.

The fourth topic, "entities and relationships" are part of the British Mark II function point metric and the newer COSMIC function point.

The seventh topic, "algorithms" is a standard factor of the feature point metric, which added a count of algorithms to the five basic function point elements used by IFPUG.

The similarity between the topics that need to be examined when gathering requirements and those used by the functional metrics makes the derivation of function point totals during requirements a fairly straightforward task.

(In 1994 the author's company built a successful function point generation tool that worked on the Bachman Analyst Workbench. Unfortunately Bachman went out of business fairly soon thereafter.)

There are some 23 additional topics that also need to be decided during the requirements phase. Some of these are "non-functional requirements and some are business

requirements needed to determine whether funding should be provided. These additional topics include:

1) The *size* of the application in function points and source code.
2) The *schedule* of the application from requirements to delivery.
3) The *cost* of the application by activity and also in terms of cost per function point.
4) The *business value* of the application and *return on investment*.
5) The major *risks* facing the application; i.e. termination, delays, overruns, etc.
6) The *security criteria* for the application and its companion data bases
7) The features of *competitive applications* by business rivals
8) The *supply chain* of the application, or related applications upstream or downstream
9) The *legacy requirements* derived from older applications being replaced.
10) The *laws and regulations* that impact the application (i.e. tax laws; privacy etc.).
11) The *quality levels* in terms of defects, reliability, and ease of use criteria.
12) The *warranty* terms of the application and responses to warranty claims.
13) The *hardware platform(s)* on which the application will operate.
14) The *software platform(s)* such as operating systems and data bases.
15) The *nationalization* criteria, or the number of foreign language versions.
16) The *performance criteria*, if any, for the application.
17) The *training requirements* or form of tutorial materials that may be needed.
18) The *installation requirements* for putting the application onto the host platforms or making it available as a service-oriented application..
19) The *reuse criteria* for the application in terms of both reused materials going into the application and also whether features of the application may be aimed at subsequent reuse by downstream applications.
20) The *use cases or major tasks* users are expected to be able to perform via the application.
21) The *control flow* or sequence of information moving through the application.
22) Possible *future requirements* for follow-on releases.
23) The *hazard levels* of any requirements that might be potentially "toxic."

The seven primary topics and the 23 supplemental topics are not the only items that can be included in requirements, but none of these 30 should be omitted by accident since they can all have a significant affect on software projects.

Note: As this paper was being written the International Function Point Users Group (IFPUG) issued a major change to function point counting. The new rules separate functional requirements from non-functional requirements. However these new rules are likely to change and they are so new that no examples of non-functional size are currently available. The data shown in this article is hypothetical and predicted by the author's Software Risk Master tool.

Gathering and Analyzing Software Requirements

Today in 2012 almost half of all major applications are replacements for aging legacy applications, some of which have been in use for more than 25 years. Unfortunately legacy applications seldom have current specifications or requirements documents available.

Due to the lack of available information about the features and functions of the prior legacy application, a new form of requirements analysis is coming into being. This new form starts by "data mining" of the legacy application in order to extract hidden business rules and algorithms from the source code. Data mining of legacy applications can also be used to gather data for sizing, in terms of both function points and code statements.

For new applications requirements gathering and requirements analysis methods will vary based on the overall size of the application as measured in function points.

Very small applications < 100 function points: Very small applications below 100 function points in size are very common for smart phones, tablets, and personal assistant devices. Usually the requirements for these small applications are created by the application developer or the inventor, rather than being gathered from potential customers.

For some small applications, such as board games like checkers or chess, the requirements may be hundreds of years old and are based on the rules of the game. All the software is doing is moving the game to a computer, so requirements are mainly those of aesthetics and ease of use rather than actual functionality.

Small applications < 1000 function points: For smaller applications between about 100 function points and 1000 function points the Agile method of having an embedded user is successful. Also successful are prototypes and the use of a variety of tools for both recording requirements, evaluating readability levels, and looking for errors and inconsistencies.

Large applications of 10,000 function points: For larger systems between 1000 and 10,000 function points the Agile approach of embedded users is no longer effective. The reason is that large systems usually have more than 1000 users, and no single user representative can possibly know what the others will use the software for.

For larger applications focus groups, Joint Application Design (JAD), and Quality Function Deployment (QFD) are all useful. Requirements standards are useful for large systems too. Requirements tools, readability tools, and static analysis tools for text are also valuable.

Prototypes are not as successful for very large systems because if a prototype is built that covers 10% of the features, then it becomes a large and difficult program in its own right.

Massive applications of 100,000 function points: For massive applications between 10,000 and 100,000 function points it is necessary to use state of the art requirements methods to have even a hope of delivering the application with acceptable quality levels and anywhere near the planned delivery date.

Requirements creep for these large multi-year applications can approach or even exceed 50% of the planned requirements. Such massive volumes of unanticipated requirements can delay deliveries by more than a year and raise costs by more than 65% compared to the original budgets.

Some of the massive applications in this size range include enterprise resource planning tools such as Oracle and SAP, large operating systems such as Windows 8, national air-traffic control, and a variety of large military systems.

Quantifying Requirements Size, Growth, and Quality for a Small Application

This section will show some typical requirements topics for both a small program of 100 function points and a large system of 10,000 function points in size. The large size was selected because requirements problems go up with size, and 10,000 function points is the range where requirements gaps, requirements creep, requirements defects, and toxic requirements become severe.

Table 3 shows traditional requirements using interviews and text documentation. Other methods such as the unified modeling language (UML) or the Agile method of embedding users in the team would create somewhat different results. However for large systems in the 10,000 function point size range requirements are troublesome using most methods.

The data in table 3 was produced using the author's Software Risk Master tool which predicts requirements size and defect volumes and supports all requirements practices.

Table 3: Requirements for Applications of 100 and 10,000 Function Points

Function Points	100	10,000
Functional requirements	*90*	*8,500*
Non-functional requirements	*10*	*1,500*
Source code (Java)	5,300	530,000
Specific requirements	65	7,407
Functional requirements	*60*	*6,295*
Non-functional requirement	*5*	*1,112*
Superfluous requirements	4	375
Requirements pages	50	2,500
English words	22,500	1,125,000
Requirements gathering/analysis days	10	60
Total requirements writing days	10	556
Words per requirement	349	152
Words per function point	180	113
FOG readability index	< 10	> 15
Diagrams	6	300
Requirements completeness	95%	60%
Requirements team	2	12
Requirements reviewers	10	40
Days to define requirements	10	60
Days to read and understand requirements	2	135
Total reading days (clients + team)	7	5,405
Requirements errors	10	875
Toxic requirements	0	18
Missing requirements	11	1,050
Creep per month in function points	2	150
Total creep function points	4	2,687
Deferred function points	0	1,522
Test cases for requirements	66	4,932
Average removal efficiency	96%	84%
Best removal efficiency	99%	96%
Average requirements defects delivered	2	178
Lowest requirements defects delivered	0	42
Major requirements defects	0	36
Requirements costs	$29,813	$2,089,162

As can be seen from table 3, requirements for small applications are relatively easy to gather and relatively free from serious problems.

Requirements for large systems, on the other hand, are very difficult to gather and also have a large number of significant errors, significant gaps, and a tendency to grow continuously throughout the development cycle.

Because users are not trained in expressing requirements or in understanding how many problems might occur due to requirements creep and requirements errors, it is up to the software engineering team to assist the users by ensuring that requirements gathering, requirements analysis, and above all requirements defect removal methods carried out in a professional fashion using proven techniques.

Continuous Requirements Growth after the Initial Release

Once software applications have been delivered to clients or customers, that does not mean that requirements stop growing and changing. For most applications growth is continuous for as long as the applications are in use. They tend to grow at rates of between 4% and 15% per calendar year forever.

Because requirements and applications continue to grow, this means that application size increases too whether measured with function points, logical code statements, or any other metric.

To illustrate the points about continuous growth, table 4 shows both typical development patterns and typical post-release patterns for a large system of 10,000 function points written in the Java language. Table 4 shows 15 intervals as predicted by the author's Software Risk Master tool. Table 4 uses integer values and makes some simplifying assumptions in order for the patterns to be clear.

Table 4: Ten-Year Requirements Growth after Initial Release

Measurement Intervals	Function Points	Logical Code Statements in Java
1 Size at end of requirements	10,000	530,000
2 Size of requirements creep	2,000	106.000
3 Size of planned delivery	12,000	636,000
4 Size of deferred features	- 4,800	- 254,400
5 Size of first delivery to clients	7,200	381,600
6 Size after year 1 usage	12,000	636,000
7 Size after year 2 usage	13,000	689,000
8 Size after year 3 usage	14,000	742,000
9 Size after year 4 usage (mid-life kicker)	17,000	901,000
10 Size after year 5 usage	18,000	954,000
11 Size after year 6 usage	19,000	1,007,000
12 Size after year 7 usage	20,000	1,060,000
13 Size after year 8 usage (mid-life kicker)	23,000	1,219,000
14 Size after year 9 usage	24,000	1,272,000
15 Size after year 10 usage	25,000	1,325,000

Table 4 shows larger than average growth at year 9 and year 13. For commercial software it is necessary to add significant new features in order to stay current with competitive applications. These are called "mid life kickers."

As can be seen from table 4, requirements growth never stops for as long as software applications are being used unless the developer withdraws support due to bringing out a new product of the same type. This is why Windows XP no longer changes except to fix bugs and security flaws.

Some applications continue well past 10 years. Several IBM applications and the U.S. Air Traffic control system have been in use for more than 30 years.

Summary and Conclusions about Requirements

Software requirements have been a very weak link in the chain of software engineering technologies. Because requirements are always incomplete and always contain errors, it is the responsibility of the software engineering team to ensure that state of the art requirements methods are used. Users are not trained in requirements methods and cannot provide requirements that are complete and error-free without assistance from trained requirements experts, plus state of the art requirements tools.

References and Readings on Software Requirements

Ambler, S.; Process Patterns – Building Large-Scale Systems Using Object Technology; Cambridge University Press; SIGS Books; 1998.

Artow, J. & Neustadt, I.; UML and the Unified Process; Addison Wesley, Boston, MA; 2000.

Bass, Len, Clements, Paul, and Kazman, Rick; Software Architecture in Practice; Addison Wesley, Boston, MA; 1997; ISBN 13: 978-0201199307; 452 pages.

Berger, Arnold S.; Embedded Systems Design: An Introduction to Processes, Tools, and Techniques; CMP Books; 2001; ISBN 10-1578200733.

Booch, Grady; Jacobsen, Ivar, and Rumbaugh, James; The Unified Modeling Language User Guide; Addison Wesley, Boston, MA; 2nd edition 2005.

Cohn, Mike; User Stories Applied: For Agile Software Development; Addison Wesley, Boston, Ma; 2004; ISBN 0-321-20568.

Fernandini, Patricial L; A Requirements Pattern; Succeeding in the Internet Economy; Addison Wesley, Boston, MA; 2002; ISBN 0-201-7386-0; 506 pages.

Gack, Gary; Managing the Black Hole – The Executives Guide to Project Risk; The Business Expert Publisher; Thomson, GA; 2010; ISBSG10: 1-935602-01-2.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John; Design Patterns: Elements of Reusable Object Oriented Design; Addison Wesley, Boston MA; 1995.

Gilb, Tom and Graham, Dorothy; Software Inspections; Addison Wesley, Reading, MA; 1993; ISBN 10: 0201631814.

Inmon William H, Zachman, John, and Geiger, Jonathan G; Data Stores, Data Warehousing, and the Zachman Framework; McGraw Hill, New York; 1997; ISBN 10: 0070314292; 358 pages.

Jones, Capers; "Early Sizing and Early Risk Analysis"; private publication by Capers Jones & Associates LLC; Narragansett, RI 2011.

Jones, Capers and Bonsignour, Olivier; The Economics of Software Quality; Addison Wesley Longman, Boston, MA; ISBN 10: 0-13-258220—1; 2011; 585 pages.

Jones, Capers; Software Engineering Best Practices; McGraw Hill, New York, NY; ISBN 978-0-07-162161-8; 2010; 660 pages.

Jones, Capers; Applied Software Measurement; McGraw Hill, New York, NY; ISBN 978-0-07-150244-3; 2008; 662 pages.

Jones, Capers; Estimating Software Costs; McGraw Hill, New York, NY; 2007; ISBN-13: 978-0-07-148300-1.

Marks, Eric and Bell, Michael; Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology; John Wiley & Sons, New York; 2006; ISBN 10: 0471768944; 384 pages.

Orr, Ken; Structured Requirements Definition; Ken Orr and Associates, Inc, Topeka, KS; 1981; ISBN 0-9605884-0-X; 235 pages.

Robertson, Suzanne and Robertson, James; Mastering the Requirements Process; 2nd edition; 2006; Addison Wesley, Boston, MA; ISBN 0-321-41949-9; 560 pages.

Martin, James & McClure, Carma; Diagramming Techniques for Analysts and Programmers; Prentice Hall, Englewood Cliffs, NJ; 1985; ISBN 0-13-208794-4; 396 pages.

Warnier, Jean-Dominique; Logical Construction of Systems; Van Nostrand Reinhold, London, UK; ISBN 0-4442-22556-3; 177 pages.

Wiegers, Karl E; Software Requirements; 2nd edition; 2003; Microsoft Press, Bellevue, WA; ISBN 10: 0735618798; 544 pages.