



# INTERFACE

NEWSLETTER OF THE SOFTWARE ENGINEERING PROCESS GROUP

VOLUME 6, NUMBER 2 • MAY 1997

## THE PERFECT REQUIREMENT MYTH

by Geoff Mullery

© Requirements Engineering Journal, Springer-Verlag London Limited

We seem to have enshrined a concept of the Project Life Cycle in which a discrete Requirement Specification phase figures as the first or, at least, an early stage. The norm has been to attempt a complete, consistent, unambiguous specification of the requirements before later stages of the project are allowed to start. The author's personal experience on performing and monitoring projects over more than 20 years has shown that to be unattainable, at least for real, non-trivial projects. Although there are those who still would argue otherwise.

In order to preserve this ideal of perfection in the face of our inability to achieve it, the notion of "requirements maintenance" was introduced. This normally is based on the idea that the original requirements, though appearing largely to meet the goals of perfection, may have some unforeseen problems at the edges, which

require a "small" maintenance effort to put right.

But, for the major systems which are so notorious for disastrous failure, the belief that an initial "near-perfect" requirement specification exercise, followed by a minor maintenance activity, will be all that is needed is a myth. A myth which retains credibility through the inability of the development community to recognize that there is no such thing as a single requirement specification. There are at least three overlapping and generally conflicting requirement specifications involved in a large, long-running development project—each with its own built in forces for change. There is a continuous need to weigh the consequences of change against its impact on each specification.

First is the Domain (or Operational or User) Requirement, which

*continued on page 4*

## REALISTIC REQUIREMENTS ENGINEERING: DEALING WITH CHANGE

by Anthony Hall, Praxis Critical Systems, Bath, UK

One certain fact about requirements is that by the time you've fielded the system they will have changed. It is futile to think of this as a problem. Some method proponents treat it as a blemish on their otherwise perfect method—you may as well claim that you have a perfect method for space travel, but unfortunately it won't work if the planets move. Instead this fact has to be one of the starting points for a realistic requirements engineering approach.

Everyone knows that you can't build a system if you keep changing its specification. But it's a certainty that the requirements are going to change. There is only one possible conclusion: you are not going to get a system which meets all requirements when it is delivered, whatever you do. You have to accept that fact. Requirements creep happens when people pretend to themselves that there is some magic way round this inescapable fact.

There isn't, so you should give up trying to find one. It isn't possible to achieve perfection, so don't even try.

Even if requirements don't change very fast before you deliver the system, they will certainly change dramatically once the system is in place. This is a consequence of another fundamental law of requirements engineering, the IKIWISI principle: I'll Know It When I See It. What the IKIWISI principle tells us is that as soon as the system is delivered, people will try it out and then discover what it was they really wanted. It is crucial to realize that this does not mean that they will necessarily want more than they originally thought. On the contrary, they will often find that many things provided in response to explicitly stated user requirements aren't really needed at all. Remember this—it's important!

*continued on page 2*

*"Realistic Requirements Engineering:  
Dealing with Change"* continued from page 1

It's also important to realize that although you can substantially reduce IKIWISI effects by doing lots of prototyping, that is never sufficient. For one thing, what users want will change as a result of having the system. So, however clever you have been in searching out the proper requirements beforehand, you will never have captured those requirements which are due to the way the system changes the user's behaviors and his or her perceptions of the job.

What you can try to achieve, realistically, is a system which:

1. meets a reasonable number of requirements;
2. is delivered in time to be useful;
3. is highly adaptable to the anticipated environment, so that even after it's delivered, it can evolve as requirements change.

To achieve these more realistic goals, you do need to do three things:

First, get a clear definition of the requirements at the time you gather them. Write down not just what the requirement is, but who said so, and why they think that. Most important of all, write down what facts about the world the requirement depends on—the context of the requirement. For example, don't just say "vertical separations must be 2000 feet." Explain what assumptions about instruments, pilot's limitations (physical and psychological), and environmental conditions are used in deriving this requirement.

Do not fall for the nonsensical idea that you can be flexible by being vague. If you are vague, then you won't get anything that you want either now or in the future. If you are precise, then you will at least get what was once wanted, which isn't that bad, if you think about it.

Second, think about which requirements are least likely to change. It would be nice if you could build systems that were indefinitely flexible but you can't. So you've got to do some work to build in the right kind of flexibility. Concentrate efforts on the things that are going to change only slowly. This isn't as difficult as it might seem because more than you might think

will stay reasonably constant. Requirements always depend on facts about the real world, as well as people's wishes, and the facts change much more slowly than the wishes. Even if details change, the underlying facts remain the same. For example, aircraft continue to have positions, altitudes and headings, even if the technology for measuring them changes.

Third, build a small system that contains the core of the application. Make sure that this core system depends fundamentally only on the unchanging facts, and not on the changeable details. This system will have only a small fraction of the functionality that has been asked for. However, to your surprise, this will turn out to be quite a large fraction of the functionality that is really needed. Such a system can be produced more quickly than a "complete" system, because it has less in it, and of course it costs less. If you build this system carefully, you will be able to add to it and adapt it in the field as the detailed requirements change.

Beware: the kind of system I am suggesting is very different from a quick and dirty prototype. It must be properly architected, so that it is robust when the time comes to change it. It must capture the essence of the real world and the user's needs, so that it is of some real use straight away. Both of these require a substantial investment of time and careful thought during requirements analysis and architectural development.

This approach does not guarantee success, but unlike some other approaches such as freezing a "complete" requirements document, it does not guarantee failure. By recognizing the reality of change and managing its consequences, it does offer a fighting chance of producing something useful. [editor's note: Anthony has recently published an article in the March, 1996 issue of IEEE Software on "Using Formal Methods to Develop an ATC Information System."]

For further information, contact:

**Anthony Hall**  
Praxis Critical Systems  
20, Manvers Street  
Bath BA1 1PX UK

+44 1225 444700  
jah@praxis.co.uk. ■

## MANAGING LOW LEVEL REQUIREMENTS

Commercial-off-the-shelf (COTS) products for networks are typically bought and implemented very quickly—from a few weeks to a few months. They tend to be selected and tailored for highly specific problems, applications, and infrastructures. For example, a product may be purchased to track trouble calls for a specific application on the Internet. The highest and sometimes sole priority is to solve the immediate problem so that the end user will quickly have access to a usable application and/or the normal flow of work is not interrupted. Although the immediate problem gets solved, stovepipes are created which make future intercommunication and interoperation among applications virtually impossible.

There is a need to be able to solve such problems within a framework which can manage all of the major and minor assets of an enterprise, including local area networks (LANs), metropolitan area networks (MANs), and wide area networks (WANs).

We are not speaking simply of automated inventory control, but also of automated interface control. We must ensure that those applications that need to communicate and interoperate can indeed do so. Such wide-ranging needs require a top down approach—preferably based on an asset management tool which can be customized for

*continued on next page*

---

# interFACE

is published quarterly by SEPG

DOT/FAA/AIT-5  
800 Independence Avenue, SW  
Washington, DC 20591

▼  
Chief Scientist for Software Engineering  
**Arthur Pyster (202) 267-8020**

▼  
Editor  
**Norman Simenson (202) 267-7431**

▼  
FAX (202) 267-5080

*"Managing Low Level Requirements"*

*continued from previous page*

the hundreds of individual sites (nodes) within the larger enterprise.

A further complication is that the technology and its use are changing so rapidly that detailed, low level requirements frequently become obsolete and inaccurate even before they are completed. Consequently, these low level requirements are expensive to produce and maintain.

One approach, adopted by many Fortune 100 companies and several government agencies, is to write high level requirements permitting the selection of an asset management tool that is customizable for the rapidly changing, low level operations of an individual site. A requirement of this approach is the ability to capture and document the low level requirements automatically as they are developed and as they evolve. These low level requirements are derived from the dialog between the network site designer and the tool during the initial customization of the tool for a particular site. Later, as the requirements for the site evolves and the site is recustomized, changes to the

original low level requirements are captured and documented in the same way.

This approach expends far less resources on up front requirements development, but better insures that the detailed requirements remain in synchronization with each other and with the major requirements as the system is customized, and later recustomized, for current operations. This is Managed Evolutionary Development, which has been used for a decade in the Department of Commerce and the Department of Defense. The resources are spent on acquiring and/or developing, and on implementing, a system to automate operations and operational evolution. The system is selected via high level requirements and samples of operational scripts, not through detailed operational requirements which are quickly obsolete.

This is exactly the approach needed for desktop management. It is an industry tested approach to major cost avoidance in the very near (18 months or less) and further out future. It is a single, strategic solution to the hundreds of tactical problems arising from

setting up and maintaining many individual help desks and network operations centers.

Still, there are currently hundreds of such asset management tools being marketed by companies such as Hewlett Packard and IBM. Cutting through the hype and choosing the best tool for a particular enterprise is nontrivial. A major problem in deploying these systems is that most are weak in capturing and documenting the lower level operational requirements during customization. This seems to be a chronic, but not insurmountable, problem. Currently, the best method seems to be to take the data output by the tool during customization and manually translate them into requirements language.

The enterprise-wide COTS asset management system development and deployment would be in stages and/or by system boundaries. Each help desk or network operations center can be installed separately and the requirements documented. The end result will be a single, consistent enterprise system with many individually customized sites. ■

## ROME LABORATORY REQUIREMENTS ENGINEERING ENVIRONMENT

by Bill Rzepka, Rome Laboratory/USAF

The U.S. Air Force Rome Laboratory (RL) has developed a unique set of integrated rapid prototyping tools for capturing complex user requirements and modeling them in a functional prototype. The set of tools is known as the Requirements Engineering Environment (REE). REE allows requirements engineers to build functional, user interface, and performance prototypes of the systems components. The models can be constructed rapidly and easily from reusable components with varying levels of abstraction or granularity. The tools have been designed to make requirements engineering technology available to users at different levels of technical understanding.

The REE tools enable users and engineers to see and execute realistic prototypes of their systems early in the development cycle when changes to the requirements and specifications are least expensive.

The REE consists of two tools. The Rapid Interface Prototyping (RIP) tool uses menus, windows, scenerio and map

generators, and application graphics as aids for conceptualizing and designing user interfaces. The PROTO tool is a visual programming environment used for constructing system architectures, and models of logic control, information, and data flow.

Together, RIP and PROTO provide a mechanism for executing an entire target system prototype with minimal user effort. Prototypes can be quickly developed and presented to all the key stakeholders very early in the conception of a new system. Thereafter, it provides a visual and tactile demonstration of the developing system at each key milestone and review. It provides an ideal means of communicating about the developing system among all of the participants: customers, users, architects, requirements engineers, designers, etc. It can be used to reconcile assumptions tendencies, preferences, and perspectives and to experiment with different approaches, problem solutions, and desired outcomes. The PROTO model can be used to drive a RIP user interface prototype. The PROTO tool

has a set of predefined functions facilitating the manipulation of RIP objects. Minimal programming is needed to connect a PROTO model with a RIP user interface. The result is a user interface reacting to and acting upon real-world stimuli.

The REE has been used successfully on a number of actual programs. It is in use by NASA; Naval Underseas Warfare Center; The Analytic Science Corporation; Lockheed Martin; International Software Systems; Defense Research Establishment, UK; and the U. S. Air Force Satellite Control Facility.

*For more information about getting copies, instruction materials, etc. about the REE, contact:*  
**Bill Rzepka**  
**Rome Laboratory/USAF**  
**525 Brooks Road**  
**Rome, NY 13441-4505**  
**(315) 330-2762**  
**eMail: bill@se.rl.af.mil**

*"The Perfect Requirement Myth"*  
continued from page 1

expresses those aspects of the domain where proposed changes need to be realized. Second is the requirement which has been traditionally produced to guide the engineers—the Technical (or System) Requirement. This requirement attempts to translate an understanding of the Domain Requirement into an engineering specification of what the proposed system must provide in order to achieve the desired affect. The third requirement, which is almost universally ignored but which severely compromises the quality of the other two, is the Contractual Requirement, which expresses the restrictions on the time and resources which must/can be used and the legal criteria for acceptance.

Nature is homeostatic. Introduction of a new element into a reasonably stable environment causes the environment to adapt, eventually to find a new balanced state. This may involve the disappearance of old elements (extinction) or modifications in their structure or behaviour (evolution) or rejection of the new element as unfitted (an evolutionary dead-end). Nevertheless, when a new Domain Requirement is expressed, it is generally assumed to be a simple extrapolation from the extant domain. Resulting perturbations of the existing domain are ignored unless compellingly obvious.

The problem which is invariably neglected is that the new system is guaranteed to cause the evolutionary perturbations mentioned above. The Domain Requirement as originally expressed, if treated rigidly (as the Contractual Requirement usually demands), is almost guaranteed to be rejected as an evolutionary dead-end,

since it cannot adapt to the changed domain behaviour it has itself introduced. Add to that the near certainty that the new system is not the only thing to introduce evolutionary change to the domain during the period of development and operational introduction. For example, in the military world, it is not only "our" side which introduces new weapon systems or sensors. You can see that the hope that a basically fixed Domain Requirement with just "minor maintenance" will be all that is needed is nonsensical.

Moreover, technology changes appear to grow exponentially. The rate of change and the frequently revolutionary nature of the change causes the same evolutionary pressures to build on the technology available for development of the proposed system—and that can dramatically influence both what is required by the domain and what technology is required to support it. It has frequently been the case that "new" systems are introduced to service with hardware or software that is already obsolete, or with missing capabilities which are critical to the now current environment, because other elements of the domain have recognized and adapted to a radical new technology. The world does not conveniently stand still while a new system is being developed. Sufficiently broad environmental changes can obsolete the architecture of the developmental system, which means the system is probably not even salvageable.

Mapping the Domain Requirement to the Technical Requirement is often attempted—but simplistically—for example, via compliancy tables relating a paragraph in one requirement to a paragraph in the other. This takes no account of the different change pres-

ures in the different requirements. Nor does it take into account the impact of discovered and undiscovered omissions, inconsistencies, and ambiguities in the two. The requirements are assumed to be free of such problems when, in reality, such problems are certain to be present—even if undocumented. A much more comprehensive and pragmatic approach to the development and inter relation of specifications is needed than is seen in current practice. Due allowance must be made for the orderly evolution of requirements.

From the point of view of technical development staff, the Cinderella of the specification process is the Contractual Requirement—the ignored step-child who suddenly bursts upon the scene to dominate and overwhelm the proceedings. It is the one which frequently turns the development process into a nightmare of impracticability and threat of insolvency. While a specification technique may propose the need for completeness, consistency, and freedom from ambiguity—and may even try to define techniques for their achievement—it makes the default assumption that the time and other resources needed will be available.

The Contractual Requirement specifies the schedule time actually available for development and may identify penalties for failure to deliver in that time. This means that the specifier must be able to forecast how long it will take to produce a complete, consistent, unambiguous specification, and then a product—bearing in mind the two types of evolutionary change (Domain and Technical) which will be under way—even as the system is under construction.

*continued on next page*

Training Opportunities

**The FAA SEPG is offering the following classes in June:**

<ul style="list-style-type: none"> <li>■ <b>Practical Software Measurement</b> <i>June 4</i></li> <li>■ <b>SLIM/SLIM Control</b> <i>June 3-4</i> <i>June 18-19</i></li> </ul>	<ul style="list-style-type: none"> <li>■ <b>Software Development Cost and Schedule Estimation</b> <i>June 17-19</i> <i>June 23-25</i></li> <li>■ <b>Software Project Planning and Tracking with Delphi Estimation Techniques</b> <i>June 24-27</i></li> </ul>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

*"The Perfect Requirement Myth"*  
continued from previous page

The Contractual Requirement is highly likely to place significant difficulties in the path of responses to evolutionary pressures during all stages of a development—yet the pressures will occur. This means that, though the changed Domain Requirement may be understood, and the resulting Technical Requirement changes may be easy to apply, there is likely to be an enforced, extensive contractual negotiation before work can safely proceed without financial risk. This can make apparently simple changes become extraordinarily costly and even impractical. Ignoring the contractual implications can mean a build-up of "simple" changes which can result in serious financial penalty. Hewing rigidly to the Contractual Requirement is likely to result in an unusable system.

Resource availability and schedule limitations expressed in a Contractual Requirement are likely to arise from other components of the Domain, which are also under change and which share resources with the proposed system or otherwise require collaboration during the development and installation process, even though they may not directly communicate with the proposed system. This means that there may be schedule or cost pressures on the Contractual Requirement from forces external to the project.

Deadlines may be brought forward or resources may become unavailable when planned for through no fault of the project. Coping with all of this is largely a managerial responsibility, not technical. But it does require cooperation across disciplines (among user, management, and technician) in a way which typically does not currently happen.

*continued on page 7*

**FAA SOFTWARE ENGINEERING  
PROCESS GROUP**

Art Pyster Chief Scientist for Software Engineering	AIT-5
Linda Ibrahim SEPG Chairperson	AIT-5
Tanae Gilmore SEPG Secretary	SETA
Rebecca Deloney	AOS-1
Tom Marker	ASU-250
Natalie Reed	ACT-24
Ross Ridgeway	AMI-100
David W. Robinson	AIT-200
Raghu Singh	AIR-200
Cindy King Skiles	AUA-7
Tom Skiles	ATR-300
Rebecca Taylor	ASD-420
George Zerdian	AND-500
<b>Alternates</b>	
Adrian Caster	AOS-5
Rob Hanes	AUA-310
Bob Laws	ASU-250
Louis Pelish	AIT-500
Art Salomon	ASD-130
Herman Tharrington	AND-3



Letter from  
the **EDITOR**

**REQUIREMENTS**

At first blush, the lead articles by Anthony Hall and Geoff Mullery would seem diametrically opposed. Anthony advocates slashing the requirements to the bone, freezing them (or holding them firm) for the first delivery, and getting the thing out the door as fast as possible. This allows the user to see what he is getting and actually use it in a real environment before the bells and whistles are added. It usually results in a substantial alteration of what the user feels she can live with—and can result in a corresponding reduction of unneeded capability asked for. In my experience this is the way to go if the product being delivered is a wholly new item which is not replacing something else and if you can get the product out the door within one to two years. If you actually deliver at least 50% of the asked for capability, including the backbone and most of the critical functions, the customer and user are generally more than happy with what you have delivered as the first step in an evolutionary development.

However, if you are replacing a working system with which the user is reasonably comfortable and which has lots of capabilities (which may have accrued over the years), then you are in serious trouble if you try to give users much less capability than they already have. In this case, all of the warnings

in Geoff's article apply. No matter how unnecessary or rarely used, the user simply will not accept a system with many "missing" capabilities. She doesn't have to—she already has a "better," "more complete" system! Here we need to distinguish between the customer, the operational user, and the maintenance user. The new (abbreviated) system may well be far cheaper and easier to maintain—so the sponsor and maintenance user will be happy. But the operational user will not, and it is my experience that most of the stakeholders have veto powers, so consensus will not be obtained and the new system will be rejected as "inadequate."

Moreover, if you are planning to require changes in operational procedures to go along with the new system which are not of obvious, immediate benefit to users—plan for very stiff resistance. In the long run, even if it adds to the cost of the new system, it is cheaper to build the new system so that it can accommodate the old procedures as well as the new, allowing for gradual change. People are (understandably) very reluctant to learn new ways of doing things if there is no obvious benefit to them.

Look at the major areas of agreement between Anthony and Geoff. Both are experienced practitioners and know full well

that the impact of a new system is substantial—and can rarely be predicted. Both agree that the best way to go is via a step-by-step development, building each successive step to take advantage of what has been learned by using the previous step in the real environment or close to it. Both recognize that requirements will change—and provide not dissimilar solutions. One recommends an explicitly evolutionary approach (using discrete steps completed one at a time) and the other recommends what amounts to an incremental approach (using overlapping steps) which allows for lots of downstream change.

My impression is that Anthony is not comfortable with the incremental approach. It is true that that approach can be a configuration nightmare where you are juggling concurrent developments each at different stages of completion. But for a very large system, that may be the only way to finish in a reasonable amount of time—like less than ten years—and the only way to keep the user from rejecting a step as wholly inadequate.

*Norm*

## REQUIREMENTS ENGINEERING

by Norman F. Simenson, AIT-5

The first major engineering project involving mankind was the building of Noah's Ark. The requirements specification was written by the Master Architect and was succinct and to the point.

"Make thee an ark of gopher wood; rooms shalt thou make in the ark, and shalt pitch it within and without.

"And this is the fashion thou shalt make of it: the length of the ark shall be three hundred cubits, the breadth of it fifty cubits, and the height of it thirty cubits."

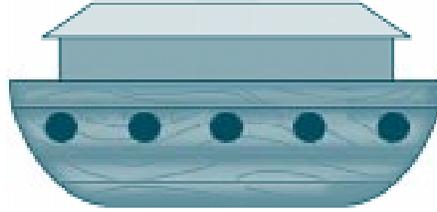
*[editor's note: a cubit is the length of the forearm to the tip of the middle finger, about 18 inches]*

"A window shalt thou make to the ark, and in a cubit shalt thou finish it above; and the door of the ark shalt thou set in the side thereof; with lower, second, and third stories shalt thou make it."

It is a matter of record that the ark was completed successfully on time and within budget and satisfied its requirements fully. But then, of course, the engineers who built it were its chief users. As A. Gustave Eiffel said, "Engineers should be made to live at the top of their own structures." He followed his own advice.

Mankind's second major engineering project was the tower of Babel. But, by then, human engineers had invented the "science" of requirements engineering and approached the task of obtaining the requirements from the Customer in the approved way. Unfortunately, there was a major miscommunication between the Customer and mankind which then deteriorated into total miscommunication among all of the workers on the project. They ended up speaking in totally different languages, with predictably disastrous results.

The moral of this story is simple: requirements "engineering" is neither engineering nor a hard science. At best, it can be characterized as a form of behavioral science. It's wise to get the full cooperation of the sponsor and user—and to clearly distinguish between the two. While good communications is important for almost any joint



human activity, it is crucial for getting the requirements right. In my experience, most projects fail because the engineers do not know how to specify, plan, design, or test a system novel to them which they do not understand. They fail at understanding the user or his problem, and this starts at the requirements phase with engineers who are ignorant of the user's environment and problem domain.

Hiring a systems or software engineer to build a system in a domain foreign to her is a little like hiring an English teacher to teach math. So the first purpose of the requirements engineering process is to teach the project engineers about the domain. At least some of the requirements engineers must be experienced in the user's domain and should be responsible for training all of the other engineers on the project.

The requirements engineers must also have lots of user involvement. The engineers have to start with a clear concept of what is needed and what is being asked for by the user, and organize requirements along the lines of a preliminary domain architecture. A formal approach can help here if it is used to model the proposed system as seen by the users and engineers. Development of the **system** architecture, however, should be delayed for as long as possible in order to accommodate the latest technology and the latest understanding of the problem.

A good **domain** architecture model will allow users and engineers to visualize the developing requirements and readily spot any glaring omissions. It can also help to bound the developing **system** architecture and spot any potential problems there, which can eliminate many design and interface problems before they occur. At this juncture, the developing architectures should reflect the growing understanding of the user's problem and be seen as a way to constrain the solution space. The architectures need to be presentable

in an easily understood form to the user for her concurrence.

Prioritizing requirements is key for any development, but different priorities will be assigned by the sponsor(s), the various users, and the various developers. A model must be constructed which will accommodate all stakeholders. This too is best handled by a commonly agreed upon **domain** architecture, which is primarily expressed in the major computer-human interfaces. The **system** architecture is the property of the engineers, and impacts the development only to the extent it constrains the direction of future evolution of the **domain** architecture or constrains priorities assigned to requirements. The strategy should be to leave the **system** architecture as open as possible and not freeze any part of it until absolutely necessary. This strategy is not usually the least expensive in the short run, but it is always the least expensive in the long run. It is maximally tolerant of change and accommodates well to the evolving understanding of requirements and requirements changes. It tends to produce system architectures which are maximally tolerant to future changes—modifications and enhancements. This reduces maintenance costs and results in longer lived systems.

So, the requirements engineers had better hold long and searching conversations with the users in a language that both understand. In that regard, the software community is a major part of the problem. It is bad enough that most software engineers believe that extensive domain knowledge is unnecessary, but they also tend to try to dazzle the users and other engineers (who could care less) by their technical grasp of the very latest hot thing in software. And the industry has a nasty habit of plowing everything under and reinventing itself every five years or so. Unwarranted claims and unproved methods form a sea of hype encapsulated in a technobabble whose primary function is to separate the new from the old.

We need to cut through the hype and rationalize the process of gathering requirements. For some time, it has been recognized that a more fundamental examination of the whole software engineering enterprise is needed, including the place of requirements engineering. ■

*"The Perfect Requirement Myth"*

*continued from page 5*

When writing a Contractual Requirement, it is necessary to allow for the influence of evolution by devoting significant down-stream resources to the requirement maintenance activity and by allowing greater flexibility in the time and cost profile. If cost and/or schedule are constrained, the user must be willing to settle for far less than he or she wants.

When writing a Domain Requirement, it is necessary to allow for the probability of incompleteness, inconsistency and ambiguity. This may well extend through the design and implementation stages, depending on the novelty and size of the proposed system. Even when using a good prototype, both the user and the technician still will have to learn how the domain really works in the area of interest, what changes really are required, and what impact the newly engineered changes really will have.

The best thing they can do is specifically to allocate resources to capture these effects and update the requirements as soon as possible. The best people to do it are the architectural team—who should be there looking out for the user throughout the development and fielding of the new system.

When writing a Technical Requirement, it is necessary to forecast where the Domain Requirement is most likely to change, to cater for the imperfections guaranteed to remain in the Domain Requirement, to insulate against the possibilities of foreseeable technology and domain change, and to allow for evolution of the requirements even as the system is developed.

We can go on trying to deal with requirement specification as a monolithic, near perfect entity concerned only with the proposed engineering solution system which is the primary purpose of the requirement exercise. If we do, we

will continue to fail when we attempt to specify and implement non-trivial novel, complex, and/or long-lived systems for a living, evolving environment. Alternatively, we can explore how to allow for the process of change—both during the initial specification process and later as development and in-service use proceeds.

*[editors's note: this article has been reduced from a somewhat longer article appearing in the Requirements Engineering Journal (1996) 1: 132-134 (Viewpoints section) Springer-Verlag London Limited, by permission of the editor and author. The Journal can be contacted for further information at <<http://www.mac.co.umist.ac.uk/RE/journal.html>>]*

*Geoff Mullery operates as an independent consultant on methods, tools and project support. His company is Systemic Methods Ltd., 12 Firs Close Farnborough, Hants GU14 6SR, UK. He has written extensively for the IEEE, the British Computing Society Requirements Engineering Specialist Group Newsletter, and other publications.*

## CONFERENCE CALENDAR

**Software Process Improvement Group (SPIN)  
DC Chapter**

June 18, 1997: Inspections, Tom Gilb  
July 2, 1997: Software Capability Evaluations  
Critical Success Factors, Paul Byrnes  
*Contact: Kathy Ditchkus (703) 641-2054*

**Federal Software Process Improvement Working Group  
(FEDSPIWG)**

Held monthly at NOAA  
*Contact: Martha Morphy at NOAA (301) 713-3345*

**Practical Software Measurement (PSM) Users Group**

July 21 - 24, 1997  
Vail, CO  
*Contact: Cheryl Jones (401) 841-4581*

**The Software Engineering Symposium**

August 25 - 28, 1997  
Washington, DC  
*Contact: SEI Customer Relations (412) 268-5800*

**Air Traffic Control Association (ATCA)**

September 28 - October 3, 1997  
Washington, DC  
*Contact:*

**International Function Point Users Group (IFPUG)**

September 15 - 19, 1997  
Scottsdale, AZ  
*Contact: (614) 895-7130*

## In This Issue

.....

**1**

The Perfect Requirement Myth  
*Geoff Mullery*

**1**

Realistic Requirements Engineering:  
Dealing With Change  
*Anthony Hall*

**2**

Managing Low Level Requirements

**3**

Rome Laboratory Requirements  
Engineering Environment  
*Bill Rzepka*

**5**

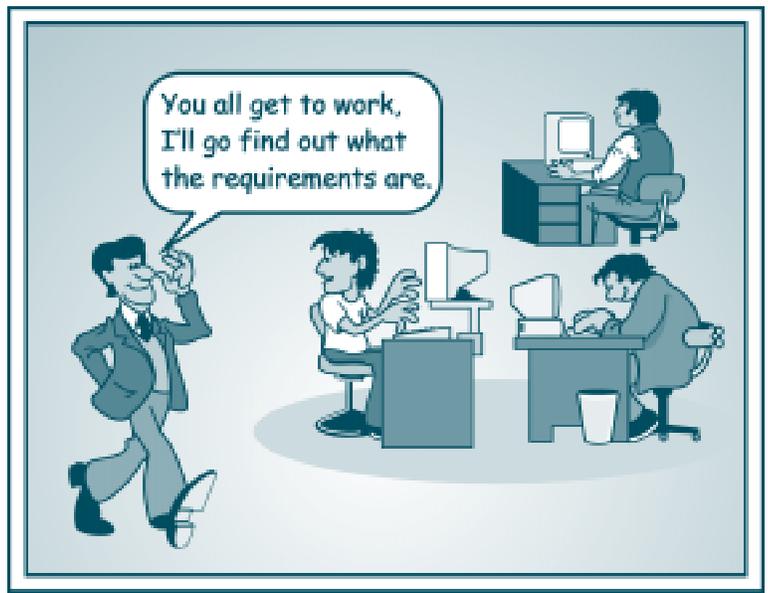
Letter from the Editor:  
Requirements  
*Norm Simenson*

**6**

Requirements Engineering  
*Norm Simenson*

**7**

Conference Calendar



# *inter*FACE

NEWSLETTER OF THE  
SOFTWARE ENGINEERING  
PROCESS GROUP

VOLUME 6, NUMBER 2  
MAY 1997



DOT/FAA/AIT-5  
800 INDEPENDENCE AVENUE, SW  
WASHINGTON, DC 20591