Quality Manifesto:

Software Quality is a Systems Engineering Job

Tom Gilb
Result Planning Limited
Tom.Gilb@INCOSE.org

Abstract.

The main idea with this paper is to wake up software engineers, and maybe some systems engineers, about quality. The software engineers (sorry, 'softcrafters') seem to think there is only one type of quality (lack of bugs), and only one place where bugs are found (in programs). My main point here is that the quality question is much broader in scope. The only way to get total necessary quality in software, is to treat the problem like a mature systems engineer. That means to recognize all critically interesting types of quality for your system. It means to take an architecture and engineering approach to delivering necessary quality. It means to stop being so computer program-centric, and to realize that even in the software world, there a lot more design domains than programs. And the software world is intimately entwined with the people and hardware world, and cannot simply try to solve their quality problems in splendid isolation. I offer some principles to bring out these points.

A Quality Manifesto

A group of my friends spent the Summer of 2007 emailing discussions about a Software Quality Manifesto. I was so unhappy with the result that I decided to write my own. At least I was unhampered by the committee.

<u>Headline</u>:

'Software Quality' is a Systems Engineering Job.

<u>Slogan</u>:

Proposition:

"**Excellent system qualities** are a continuous management and engineering challenge, with no perfect solutions".

Corollary:

“when **management and engineering fail** to execute their quality responsibilities professionally, the quality levels are accidental; and probably unsatisfactory to most stakeholders.”

Quality Manifesto/Declaration

System Quality can be viewed as a set of quantifiable performance attributes, that describe how well a system performs for stakeholders, under defined conditions, and at a given time.

System Stakeholders judge past, present, and future quality levels; in relationship to own their perceived needs/values.

System Engineers can analyze necessary, and desirable, quality levels; and plan, and manage to deliver, a set of those quality levels, within given constraints, and available resources.

Quality Management is responsible for prioritizing the use of resources, to give a satisfactory fit, for the prioritized levels of quality: and for trying to manage the delivery of a set of qualities - that maximize value for cost - to defined stakeholders.

Quality Principles Heuristics for Action: An Overview.

1. **Quality Design**: *Ambitious* Quality Levels are **designed in**, not tested in. *This applies to work processes and work products.*

2. **Software Environment**: "Software" Quality is *totally dependent on its resident system* quality, and does not exist alone; 'software qualities' are dependent on a defined system's qualities – including stakeholder perceptions and values.

3. **Quality Entropy**: Existing or planned quality levels will deteriorate in time, under the pressure of other prioritized requirements, and through lack of persistent attention.

4. **Quality Management**: Quality levels can be systematically managed to support a given quality policy. *Example : "Value for money first", or "Most competitive World Class Quality Levels".*

5. **Quality Engineering**: A set of quality levels can be technically engineered, to meet stakeholder ambitions, within defined constraints, and priorities.

6. **Quality Perception**: Quality is in the eyes of the beholder: objective system quality levels may be simultaneously valued as *great* for some stakeholders, and *terrible* for others.

7. **Design Impact on Quality**: any system design component, whatever its *intent*, will likely have *unpredictable* main effects, and side effects, on *many* other quality levels, many constraints, and many resources.

8. **Real Design Impacts**: you cannot be *sure* of the totality of effects, of a design for quality, on a system, except by measuring them in *practice*; and even then, you cannot be sure the measure is *general*, or will *persist*.

9. **Design Independence**: Quality levels can be measured, and specified, independently of the means (or designs) needed to achieve them

10. **Complex Qualities:** many qualities are best defined as a subjective, but useful, set of elementary quality dimensions; this depends on the degree of control you want over the separate quality dimensions.[1]

[1] CE Chapter 5, download, http://www.gilb.com/community/tiki-download_file.php?fileId=26 will give rich illustration to this point. See for example Maintainability, Adaptability and Usability.

Quality Principles Heuristics for Action: **detailed** remarks

1. **Quality Design**: *Ambitious* Quality Levels are **designed in**, not tested in. *This applies to work processes and work products.*

There is far too much emphasis on testing and reviews, as a means to deal with defects and bugs. It is a well-known paradigm that you 'do not test quality into a system, you design it in'. We can look at this problem from both an economic and an effectiveness point of view.

From an economic point of view, it pays off, by one or two orders of magnitude, to solve problems early. 44%-64% of all coding defects are the results of defects in specifications (requirements, design) given to programmers [Inspection for Managers [ATT, TRW], as reference for this and other facts about test and reviews]. The cost of removal of defects at late stages explodes by 10x to 100x and more. A stitch in time saves nine.

From an effectiveness point of view, both tests and reviews are ineffective. The range of effectiveness is roughly 25% to 75% (probability of actually detecting defects that are present. [Insp. For Mgt., Capers Jones]. Jones reckons that if we had an effective series of about 11 reviews and tests, we could only remove a maximum of 95% of the injected defects. My conclusion is that 'cleaning up injected defects' is a hopeless cause. There are better options.

The interesting option is that 'an ounce of prevention of worth a pound of cure'. We have to learn to avoid the infection of defects in the first place. It is clear that we can reduce the injection rates by at least 100 to 1. Most requirements documents today (my personal client measurements) contain about 100 major defects per page (300 words). The standard that advanced developers (IBM [Humphrey], NASA) have long since established is a tolerance (process exit level) of less than 1.0 majors/page (IBM : 0.25, NASA : 0.10). This is the primary focus of CMMI Level 5 (Defect Prevention Process [Mays, Robert, IBM]. It takes my clients about 6 months to reduce injection by factor ten, and another 2-3 years by another factor 10. This is obviously more cost-effective than waiting until we can test for defects, or until customers complain.

2. **Software Environment**: "Software" Quality is *totally dependent on its resident system* quality, and does <u>not exist alone</u>; 'software qualities' are dependent on a defined system's qualities – including stakeholder perceptions and values.

We tend to treat software quality as something inherently resident in the software itself. But all qualities (example Security, Usability, Maintainability, Reliability) are highly dependent on people, their qualifications, and they way the use systems. The consequence is that we must plan, specify and design with a stronger eye to identifying and controlling the factors that actually decide the system quality. We have to engineer the system as a whole, not just the 'code'). We must be systems engineers, not program engineers. This has large implications for how we train people, how we organize our work, and how we motivate people. We will also have to shift emphasis from the technology itself (the means) to the results we actually need (the ends, quality requirement levels).

3. **Quality Entropy**: Existing or planned quality levels will deteriorate in time, under the pressure of other prioritized requirements, and through lack of persistent attention.

Even the concept of numeric quality levels, for most qualities – example usability, security, adaptability – is alien to most software engineers, and to far too many systems engineers. But the basic concept of quantified quality levels is old and well established in engineering.

In spite of this poor starting environment, of too many people satisfied with using words ('easy to use') instead of numbers ('30 minutes to learn task X by Employee type Y'), we need to not merely achieve planned quality levels upon initial delivery and acceptance of systems. We need to imbed in the systems the measurement of these qualities, and the warning systems needed to tell us they are deteriorating or drastically fallen. We need to expect to take action to improve the quality levels back to planned levels, and perhaps improve them even more in the future.

4. **Quality Management**: Quality levels can be systematically managed to support a given quality policy. *Example: "Value for money first", or "Most competitive World Class Quality Levels".*

It is useful management if there is a policy about the levels of quality we aspire to, both at a corporate level, and a project level. We cannot really allow isolated individuals to make their dream levels of quality be taken as requirements, without due balance towards the priorities of the other competing levels. And we need to keep our eyes on available resources and technological limits and opportunities.

We need to decide if we are there to 'be the state of the art' (as Rockwell explained to me once) of 'get the most value for money', as others need to worry about.

A policy like this might be generally useful: "Quality levels will be engineered to a level that gives us arguably high return on the investment needed to get them there, and so that the levels do not steal resources for other parallel investment opportunities in quality, or elsewhere.”.

5. **Quality Engineering**: A set of quality levels can be technically engineered, to meet stakeholder ambitions, within defined constraints, and priorities.

It is a tricky business to decide which numeric quality levels are appropriate. Initially we cannot decide the right levels in isolation. We need to know about the larger environment, both the environment for the single quality attribute, and for the *set* of attributes – for their environment.

We need to learn to specify this environment together with the requirement ideas themselves. It will be easier to make decisions about the relative levels of quality and their priority if we have a decisive set of facts about each attribute. For example, it is useful to know things like the:

- Value for a level
- The stakeholders for a quality and for various levels
- The timing needs of levels of quality
- The planned strategies and their expected costs for reaching given levels

And quite a few other things – that will help us reason about the right levels of quality.

Elementary scalar requirement template <with hints>

Tag: <Tag name of the elementary scalar requirement>.

Type:
<{Performance Requirement: {Quality Requirement,
  Resource Saving Requirement,
  Workload Capacity Requirement},
Resource Requirement: {Financial Requirement,
  Time Requirement,
  Headcount Requirement,
  others}}>.

============================ Basic Information ===========================

Version: <Date or other version number>.
Status: <{Draft, SQC Exited, Approved, Rejected}>.
Quality Level: <Maximum remaining major defects/page, sample size, date>.
Owner: <Role/e-mail/name of the person responsible for this specification>.

Stakeholders: <Name any stakeholders with an interest in this specification>.

Gist: <Brief description, capturing the essential meaning of the requirement>.
Description: <Optional, full description of the requirement>.
Ambition: <Summarize the ambition level of *only the targets* below. Give the overall real ambition level in 5–20 words>.

============================ Scale of Measure ===========================

Scale: <Scale of measure for the requirement (States the units of measure for all the targets, constraints and benchmarks) and the scale qualifiers>.

============================= Measurement ============================

Meter: <The method to be used to obtain measurements on the defined Scale>.

============= Benchmarks ============= "Past Numeric Values" =============

Past [<when, where, if>]: <Past or current level. State if it is an estimate> <- <Source>.
Record [<when, where, if>]: <State-of-the-art level> <- <Source>.
Trend [<when, where, if>]: <Prediction of rate of change or future state-of-the-art level> <- <Source>.

============== Targets ============== "Future Numeric Values" =============

Goal/Budget [<when, where, if>]: <Planned target level> <- <Source>.
Stretch [<when, where, if>]: <Motivating ambition level> <- <Source>.
Wish [<when, where, if>]: <Dream level (unbudgeted)> <- <Source>.

============== Constraints ============= "Specific Restrictions" =============

Fail [<when, where, if>]: <Failure level> <- <Source>.
Survival [<when, where, if>]: <Survival level> <- <Source>.

============================== Relationships ==============================

Is Part Of: <Refer to the tags of any supra-requirements (complex requirements) that this requirement is part of. A hierarchy of tags (For example, A.B.C) is preferable>.
Is Impacted By: <Refer to the tags of any design ideas that impact this requirement> <- <Source>.
Impacts: <Name any requirements or designs or plans that are impacted significantly by this>.

======================= Priority and Risk Management ======================

Rationale: <Justify why this requirement exists>.
Value: <Name [stakeholder, time, place, event]: Quantify, or express in words, the value claimed as a result of delivering the requirement>.
Assumptions: <State any assumptions made in connection with this requirement> <- <Source>.
Dependencies: <State anything that achieving the planned requirement level is dependent on> <- <Source>.
Risks: <List or refer to tags of anything that could cause delay or negative impact> <- <Source>.
Priority: <List the tags of any system elements that must be implemented before or after this requirement>.
Issues: <State any known issues>.

Figure 1: an example of a template that tries to collect some of the information that I think we ought to know about in order to decide how to prioritize particular levels of quality for a single attribute. [CE book, page 135]

6. **Quality Perception**: Quality is in the eyes of the beholder: objective system quality levels may be simultaneously valued as *great* for some stakeholders, and *terrible* for others.

The point is that any real complex large system will have many different stakeholders. Even one stakeholder category [Novice User, Call Center Manager] can have many individuals, with highly individual needs and priorities. The result will inevitably be a compromise. But we can make that compromise as intelligent as possible. We do not have to design systems with only one level for all stakeholders. We can consciously decide to have different quality levels of the same quality, for different stakeholders, at different times and situations.

For example:

Learnability:

Scale: the time needed for a defined [Stakeholder] to Master a defined [Process].

Goal [Stakeholder = Top Manager, Process = Get Report] 5 minutes.

Goal [Stakeholder = Offshore Clerk, Process = Create New Account] 1 hour.

7. **Design Impact on Quality**: any system design component, whatever its *intent*, will likely have *unpredictable* main effects, and side effects, on *many* other quality levels, many constraints, and many resources.

I see far too much narrow reasoning, of the type: "we are going to achieve great quality X using technology X, Y and Z". This reasoning is not with numbers, but only nice words. Yet I have seen in it $100 million projects, often!

We have to learn to specify, analyze and think in terms of 'multiple numeric impacts of many designs, on our many critical quality and cost requirements'. Quality Function Deployment (QFD) takes this position, but I am not happy with the way in which numbers are used in QFD) – too subjective., too undefined [QFD].

We need to systematically, as best we can, estimate all the multiple effects or each significant design.

	<u>**On-line Support**</u>	<u>**On-line Help**</u>	<u>**Picture Handbook**</u>	<u>**On-line Help**</u> **+** <u>**Access Index**</u>
<u>**Learning**</u> Past: 60min. <<-> Plan: 10min.				
Scale Impact	5 min.	10 min.	30 min.	8 min.
Scale Uncertainty	±3min.	±5 min.	±10min.	±5 min.
Percentage Impact	110%	100%	67% (2/3)	104%
Percentage Uncertainty	±6% (3 of 50 minutes)	±10%	±20%?	±10%
Evidence	Project Ajax, 1996, 7 min.	Other Systems	Guess	Other Systems + Guess
Source	Ajax report, p.6	World Report p.17	John B.	World Report p.17 + John B.
Credibility	0.7	0.8	0.2	0.6
Development Cost	120K	25K	10K	26K
Benefit-To-Cost Ratio	110/120 = 0.92	100/25 = 4.0	67/10 = 6.7	104/26 = 4.0
Credibility-adjusted B/C Ratio (to 1 decimal place)	0.92*0.7 = 0.6	4.0*0.8 = 3.2	6.7*0.2 = 1.3	4.0*0.6 = 2.4
Notes: Time Period is two years.	Longer timescale to develop			

Figure 2: A systematic analysis of 4 designs on one quality level (10 minutes). This is an impact estimation table. [CE, page 267].

8. **Real Design Impacts**: you cannot be *sure* of the totality of effects, of a design for quality, on a system, except by measuring them in *practice*; and even then, you cannot be sure the measure is *general*, or will *persist*.

I have seen books, papers, and project specifications for software that confidently predict a good result (not usually quantified) from a particular design, solution, architecture or strategy. Maybe it is easier to be confident if no particular numeric impact is ever asserted.

In normal engineering, no matter what the engineering handbook says, no matter what we would like to believe; the prudent engineer takes the trouble to measure the *real* effects.

We need to carefully do early measurements, then repeat measurements when scaling up, at acceptance times, and later in long-term operation. In we can never take critical qualities for granted, or as if they are stable.

We can plan this in advance to a reasonable degree:

Learnability:

Scale: minutes to learn a Task by a User.

Meter [Weekly Development, 2 Users, 10 Normal tasks]

Meter [Acceptance Test, Duration 60 day, 200 Users, 10 normal tasks, 20 extreme tasks]

Meter [Normal Operation, Sampling Frequency 2%, Tasks = All Defined]

Each 'Meter' specification defines or sketches a different intended test to measure the quality level.

9. **Design Independence**: Quality levels can be measured, and specified, independently of the means (or designs) needed to achieve them.

There is far too much immediately coupling of named design ideas, with named quality types. "We will improve product agility using structured tools' – type of specification.

We need to focus our specifications on the quality levels we require, and studiously avoid mentioning our favored design idea in the same sentence.

Specifying a 'design', when you need to focus on the quality level, should be considered a major defect in the specification. Dozens or more such 'false requirements' per page of 'requirements' are not uncommon in our 'software' culture.

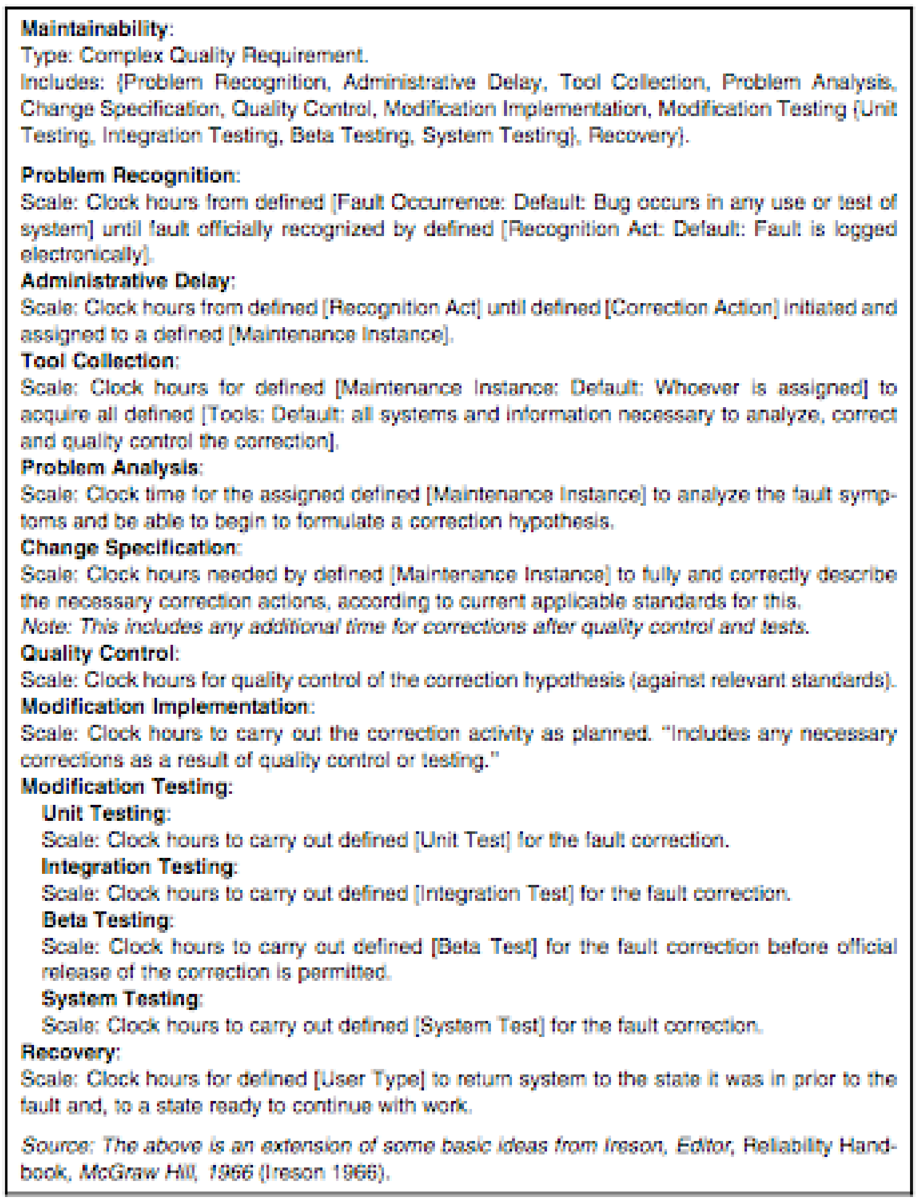
10. **Complex Qualities:** many qualities are best defined as a subjective, but useful, set of elementary quality dimensions; this depends on the degree of control you want over the separate quality dimensions.[2]

I think there is too little awareness of the fact that quality words often are the name of a set of qualities. The only way to define such complex qualities is to list all the components of the set. Only in this way will we understand what the real requirements are.

We need to learn the general patterns of the most common qualities, as in the example below.

[2] CE Chapter 5, download, http://www.gilb.com/community/tiki-download_file.php?fileId=26 will give rich illustration to this point. See for example Maintainability, Adaptability and Usability.

We need to avoid oversimplification of qualities, when, the detailed set of sub-attributes will give us a fair chance at getting control over the critical qualities we want to manage.


Maintainability:
Type: Complex Quality Requirement.
Includes: {Problem Recognition, Administrative Delay, Tool Collection, Problem Analysis, Change Specification, Quality Control, Modification Implementation, Modification Testing {Unit Testing, Integration Testing, Beta Testing, System Testing}, Recovery}.

Problem Recognition:
Scale: Clock hours from defined [Fault Occurrence: Default: Bug occurs in any use or test of system] until fault officially recognized by defined [Recognition Act: Default: Fault is logged electronically].
Administrative Delay:
Scale: Clock hours from defined [Recognition Act] until defined [Correction Action] initiated and assigned to a defined [Maintenance Instance].
Tool Collection:
Scale: Clock hours for defined [Maintenance Instance: Default: Whoever is assigned] to acquire all defined [Tools: Default: all systems and information necessary to analyze, correct and quality control the correction].
Problem Analysis:
Scale: Clock time for the assigned defined [Maintenance Instance] to analyze the fault symptoms and be able to begin to formulate a correction hypothesis.
Change Specification:
Scale: Clock hours needed by defined [Maintenance Instance] to fully and correctly describe the necessary correction actions, according to current applicable standards for this.
Note: This includes any additional time for corrections after quality control and tests.
Quality Control:
Scale: Clock hours for quality control of the correction hypothesis (against relevant standards).
Modification Implementation:
Scale: Clock hours to carry out the correction activity as planned. "Includes any necessary corrections as a result of quality control or testing."
Modification Testing:
Unit Testing:
Scale: Clock hours to carry out defined [Unit Test] for the fault correction.
Integration Testing:
Scale: Clock hours to carry out defined [Integration Test] for the fault correction.
Beta Testing:
Scale: Clock hours to carry out defined [Beta Test] for the fault correction before official release of the correction is permitted.
System Testing:
Scale: Clock hours to carry out defined [System Test] for the fault correction.
Recovery:
Scale: Clock hours for defined [User Type] to return system to the state it was in prior to the fault and, to a state ready to continue with work.

Source: The above is an extension of some basic ideas from Ireson, Editor, Reliability Handbook, *McGraw Hill, 1966* (Ireson 1966).

Figure 3: An example of Maintainability as a set of other measures of quality. [CE page 156].

Summary

Purpose [of Quality Manifesto]:

To promote a healthy view of software quality.

Gap Analysis:

To help people get to where they really need to be in order to meet their stakeholders expectations as well as resources permit.

Justifications [for positions taken here]

1. We must take a **systems-centric**, not a programming-centric view of quality.

Because: Software only has quality attributes in relation to people, hardware, data, networks, values. It cannot be isolated from the related world that decides

- which quality dimensions are of interest (critical)

- which quality levels are of value to a given set of stakeholders.

2. We must take a **'stakeholder' view** – not customer or user or any much-too-limited limited set of stakeholders.

Because: the qualities that must be engineered and finally present in a software system depend on the entire set of critical stakeholders, not a on a limited few.

3. We must make a clear **distinction** between **various 'defect' types,** as good IEEE engineering standards already do.

Because; we cannot afford to confuse specification defects, with their potential product faults, and product faults with potential product malfunctions. See these definitions.

References

Gilb, Tom, Competitive Engineering [**CE**], A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage, ISBN 0750665076,

2005, Publisher: Elsevier Butterworth-Heinemann. Sample chapters will be found at Gilb.com.

Chapter 5: Scales of Measure:

http://www.gilb.com/community/tiki-download_file.php?fileId=26

Chapter 10: Evolutionary Project Management:

http://www.gilb.com/community/tiki-download_file.php?fileId=77

Gilb.com: www.gilb.com. our website has a large number of free supporting papers , slides, book manuscripts, case studies and other artifacts which would help the reader go into more depth

For example:

Gilb, Inspection for Managers, a set of slides with facts and cases.

http://www.gilb.com/community/tiki-download_file.php?fileId=88

Gilb: What's Wrong with **QFD**?

http://www.gilb.com/community/tiki-download_file.php?fileId=119

INCOSE Systems Engineering Handbook v. 3

INCOSE-TP-2003-002-03, June 2006 , www.INCOSE.org

Software World Conference Website:

- Bethesda Md., USA, September 15-18th 2008
- http://www.asq509.org/ht/display/EventDetails/i/18370

Source: of Quality Opinions:
http://www.qualitydigest.com/html/qualitydef.html [2001]

Biography

BIOGRAPHY

Tom Gilb is an international consultant, teacher and author.

His 9th book is '**Competitive Engineering**: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage' (August 2005 Publication, Elsevier) which is a definition of the planning language 'Planguage'.

He works with major multinationals such as Credit Suisse, Schlumberger, Bosch, Qualcomm, HP, IBM, Nokia, Ericsson, Motorola, US DOD, UK MOD, Symbian, Philips, Intel, Citigroup, United Health, Boeing, Microsoft, and many smaller and lesser known others See www.Gilb.com .

Version: start 29 oct Monday am 03:00 --→03:36, completed Tuesday AM 00:49 30th oct.