Designing **Maintainability** in Software Engineering: a *Quantified* Approach.

Tom Gilb

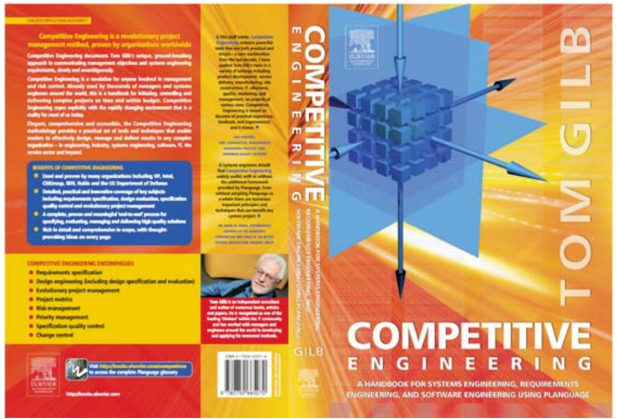
Result Planning Limited

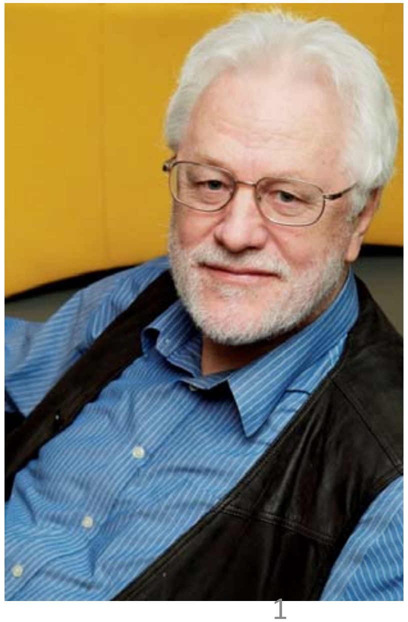
Tom.Gilb@INCOSE.org

For ACCU Oxford UK

Friday 4th April 2008

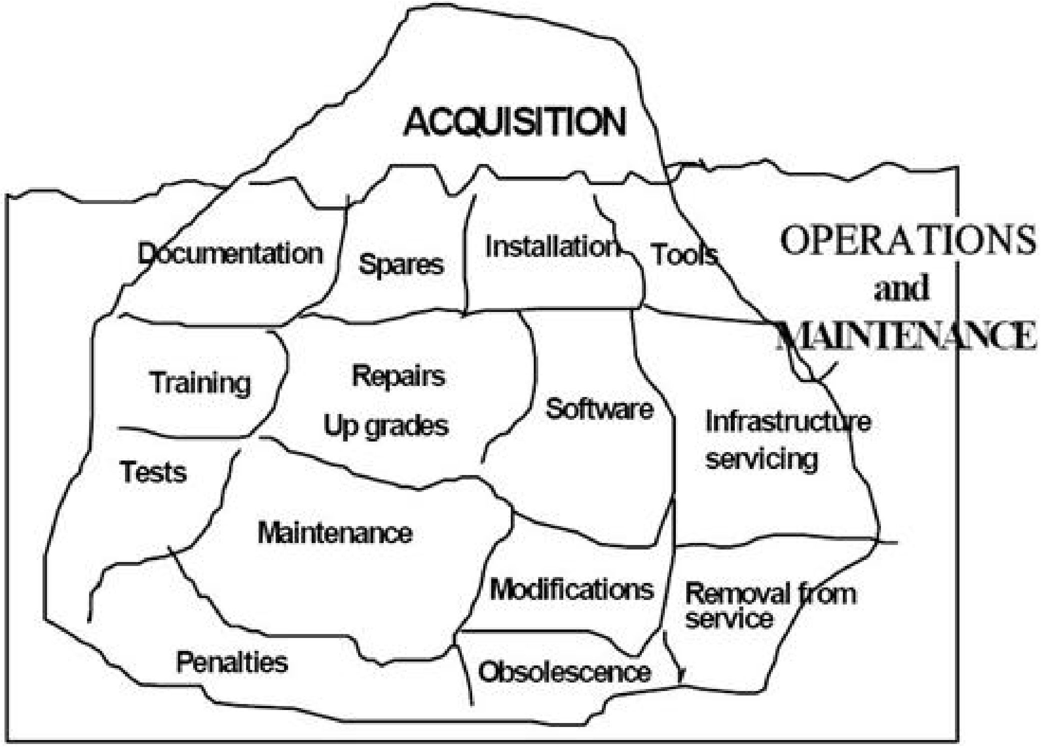
1400 90 MInutes


TOM GILB
COMPETITIVE
ENGINEERING
A HANDBOOK FOR SYSTEMS ENGINEERING, REQUIREMENTS ENGINEERING, AND SOFTWARE ENGINEERING USING PLANGUAGE



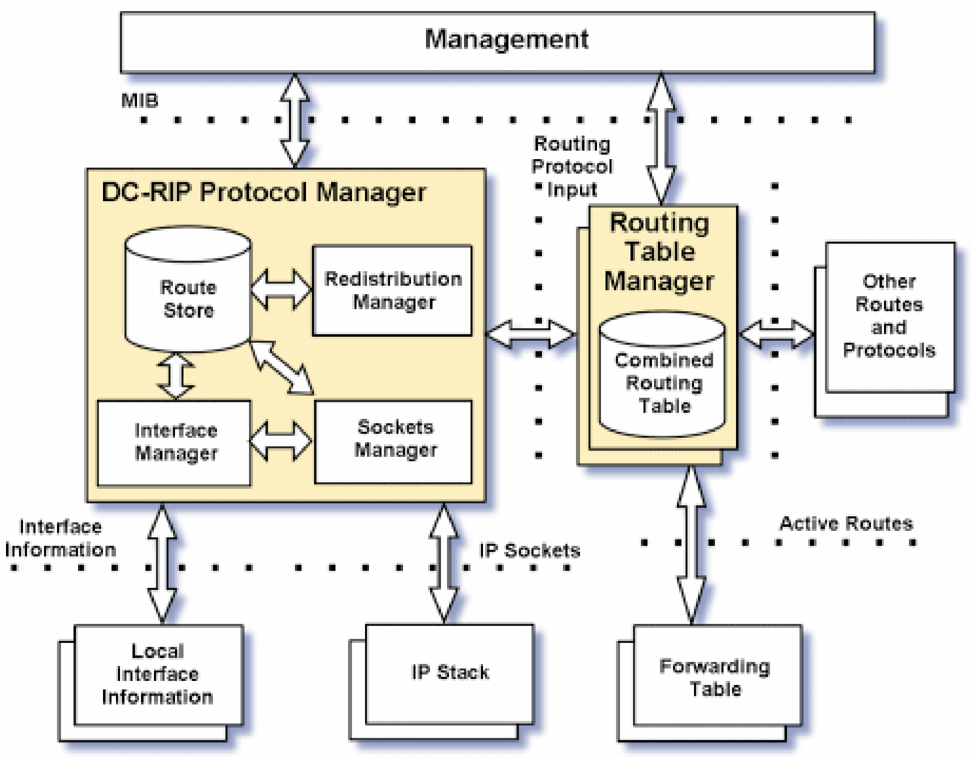
Abstract.

- Software system maintenance costs are a substantial part of the life cycle costs.
- They can easily steal all available effort away from new development.



Abstract

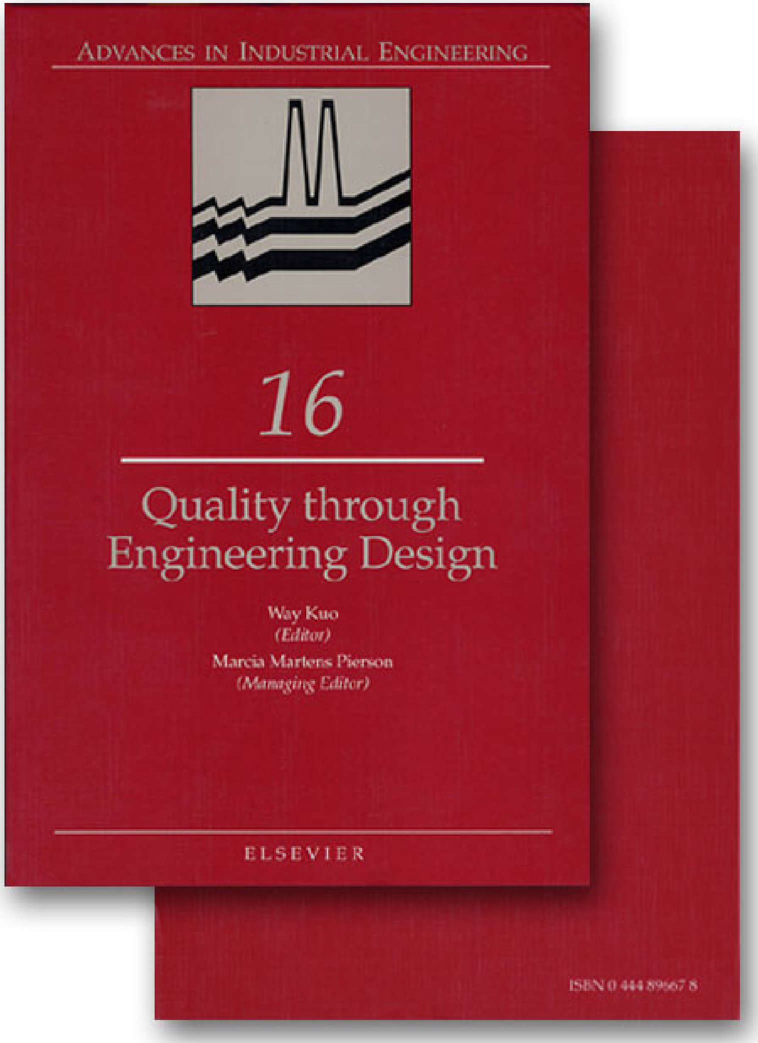
- I believe that this is because
  - **maintainability is, as good as never, systematically engineered into the software.**
- Our so-called software architects bear a primary responsibility for this, but they do not engineer to targets.
- They just throw in customs and habits that seem appropriate.



Did you ever see ideas like performance and quality, for example ‘Portability Levels’ in a software architecture diagram?

Abstract

- We need to
  - define our maintainability requirements quantitatively,
  - Set quality investment targets that will pay off,
  - pursue long-term engineered improvement of the systems, and then
  - 'architect' and 'engineer' the resulting system.
- Traditional disciplines may already in principle understand this discipline,
  - some may not understand it,
  - some may simply not apply the engineering understanding that is out there



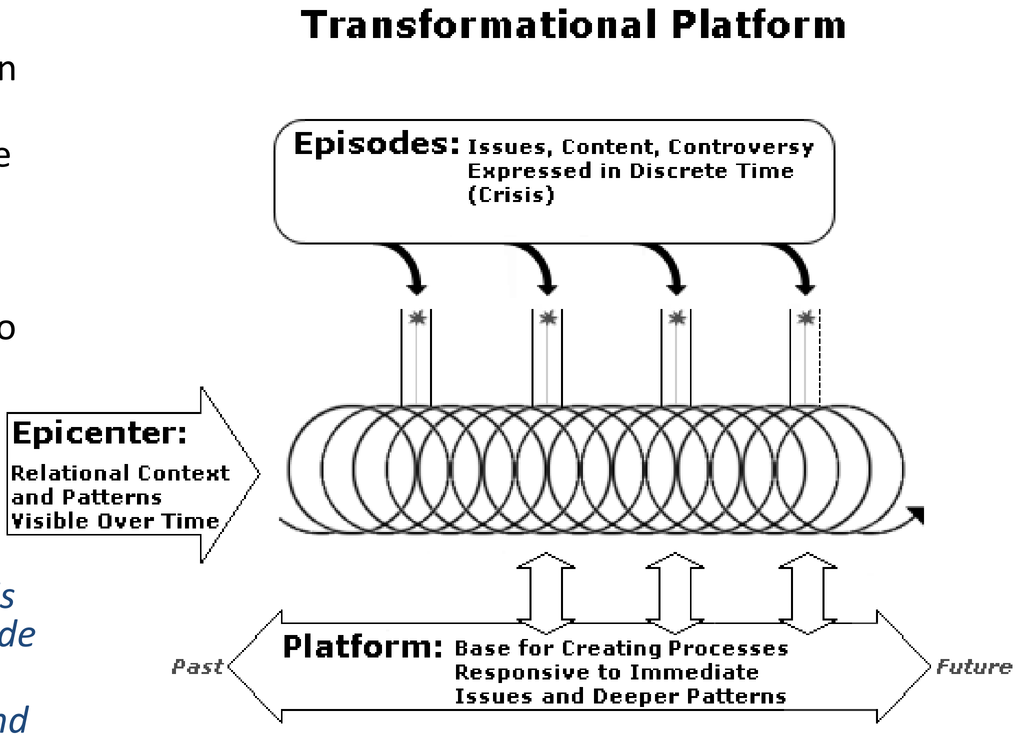
The Maintainability Problem

- Software systems are built under high pressure to meet deadlines, and with initial emphasis on performance, reliability, and usability.
- The software attributes relating to later changes in the software – maintainability attributes are:
  - never specified quantitatively up front in the software quality requirements
  - never architected to meet the non-specified maintainability quality requirements
  - never built to the unspecified architecture to meet the unspecified requirements
  - never tested before software release
  - never measured during the lifetime of the system.

*“A number of people expressed the opinion that code is often **not designed for change**. Thus, while the code meets its operational specification,*

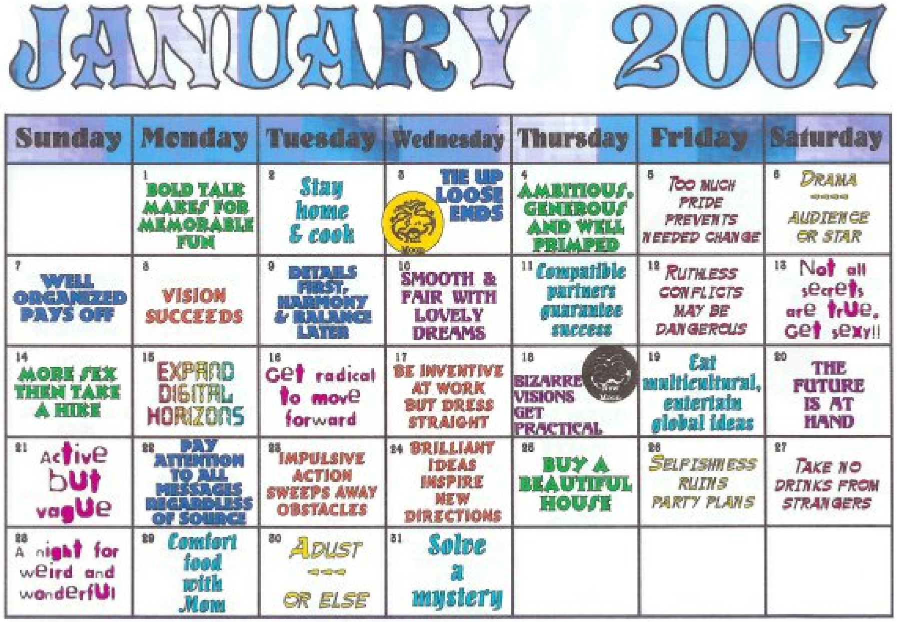
for maintenance purposes it is poorly designed and documented “ [Dart 93]

- In short, there is no engineering approach to software maintainability.


Transformational Platform
Episodes: Issues, Content, Controversy Expressed in Discrete Time (Crisis)
Epicenter: Relational Context and Patterns Visible Over Time
Platform: Base for Creating Processes Responsive to Immediate Issues and Deeper Patterns
Past
Future

What *do* we do in practice today?

- we might *bullet point* some high-level objectives
  - (‘• Easy to maintain’)
  - which are never taken seriously
- we might even decide the technology we will use to reach the vague ideal
  - (“• Easy to maintain through modularization, object orientation and state of the art standard tools”)
- larger institutions might have ‘software architects’ who carry out certain customs, such as
  - decomposition of the software,
  - choice of software platforms and software tools – generally intended to help – hopefully.
  - But with no specific resulting level or type of maintainability in mind.
- we might recommend more and better tools, but totally fail to suggest an *engineering* approach [Dart 93].
- We could call this a ‘craft’ approach.
- It is not ‘engineering’ or ‘architecture’ in the normal sense.

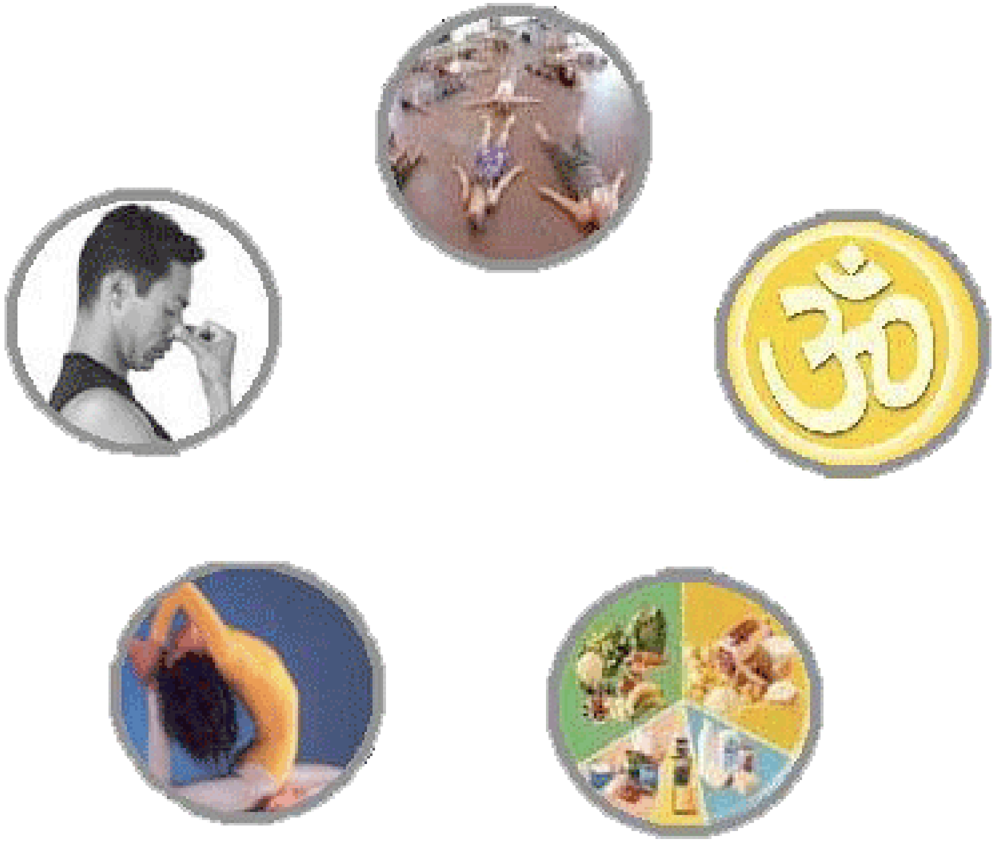


JANUARY 2007

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	1 BOLD TALK MAKES FOR MEMORABLE FUN	2 Stay home & cook	3 TIE UP LOOSE ENDS	4 AMBITIOUS, GENEROUS AND WELL PRIMPED	5 TOO MUCH PRIDE PREVENTS NEEDED CHANGE	6 DRAMA AUDIENCE OR STAR
7 WELL ORGANIZED PAYS OFF	8 VISION SUCCEEDS	9 DETAILS FIRST, HARMONY & BALANCE LATER	10 SMOOTH & FAIR WITH LOVELY DREAMS	11 Compatible partners guarantee success	12 RUTHLESS CONFLICTS MAY BE DANGEROUS	13 Not all secrets are true, Get sexy!!
14 MORE SEX THEN TAKE A HIKE	15 EXPAND DIGITAL HORIZONS	16 Get radical to move forward	17 BE INVENTIVE AT WORK BUT DRESS STRAIGHT	18 BIZARRE VISIONS GET PRACTICAL	19 Eat multicultural, entertain global ideas	20 THE FUTURE IS AT HAND
21 Active but vague	22 PAY ATTENTION TO ALL MESSAGES REGARDLESS OF SOURCE	23 IMPULSIVE ACTION SWEEPS AWAY OBSTACLES	24 BRILLIANT IDEAS INSPIRE NEW DIRECTIONS	25 BUY A BEAUTIFUL HOUSE	26 SELFISHNESS RUINS PARTY PLANS	27 TAKE NO DRINKS FROM STRANGERS
28 A night for weird and wonderful	29 Comfort food with Mom	30 ADUST OR ELSE	31 Solve a mystery			

Principles of Software Maintainability

- I would like to suggest a set of principles about software maintainability,
  - in order to give this talk a framework:



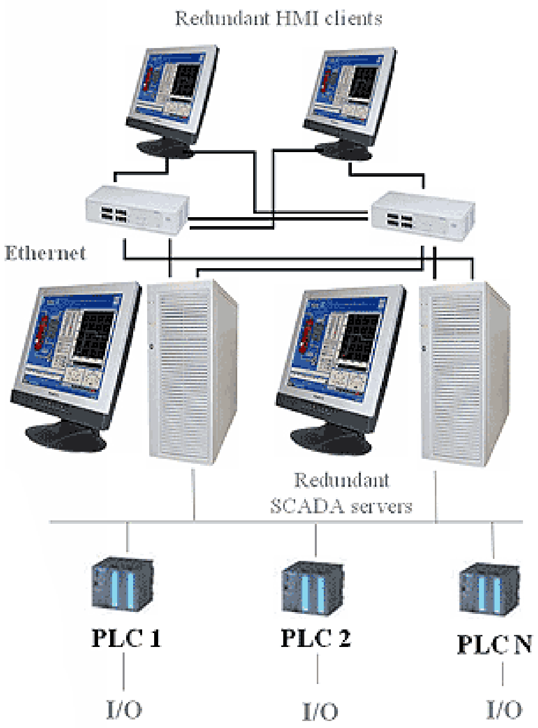
Body Maintenance: {Relax, Exercise, Breathing, Diet, Positive Thinking and Meditation}.

1. **The Conscious Design Principle**:

- Maintainability must be *consciously* designed into a system:
  - failure to **design** to a set of levels of maintainability
  - means the **resulting maintainability** is both *bad* and *random*.

Conscious Design

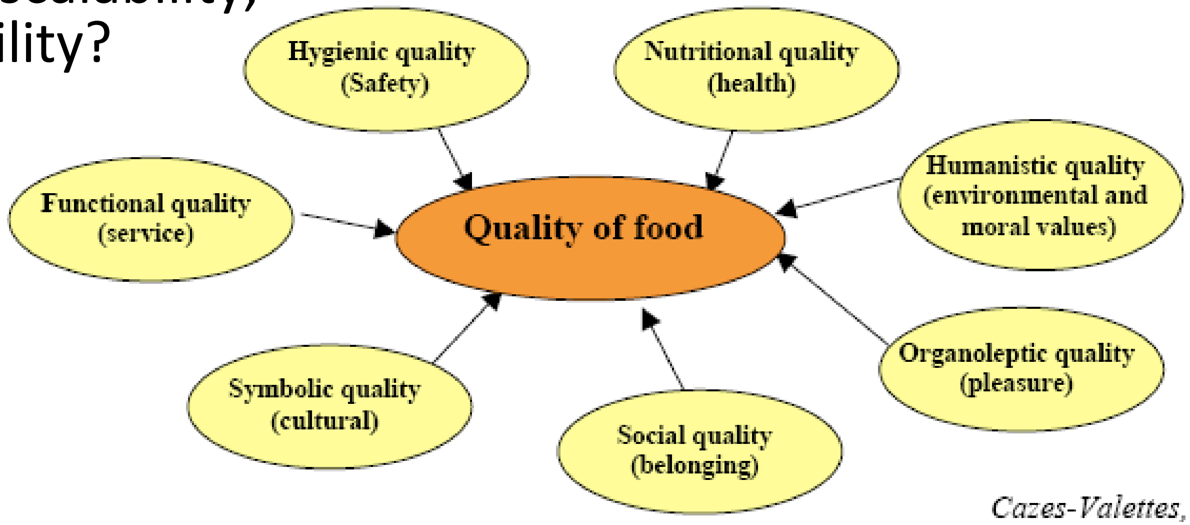
- Clarify
  - Robust →
    - 200 Days Between Restarts
- Find Solutions
  - Triple Redundant Systems ?
- Verify Solutions
  - 400 Days average achieved!



2. The Many-Splendored Thing Principle.

- Maintainability is
  - a **wide set** of change-quality types,
  - under a **wide** variety of **circumstances**:
  - so we must clearly define **what quality type** we are trying to engineer. Like:
    - Portability, scalability, maintainability?

THE ENDLESS JOURNEY TOWARD TOTALITY
EXPERIENCE THE COMMON ROOT OF HUMANITY
ETERNAL WELLNESS
Mental
Physical
Spiritual
Emotional
Social
Environmental

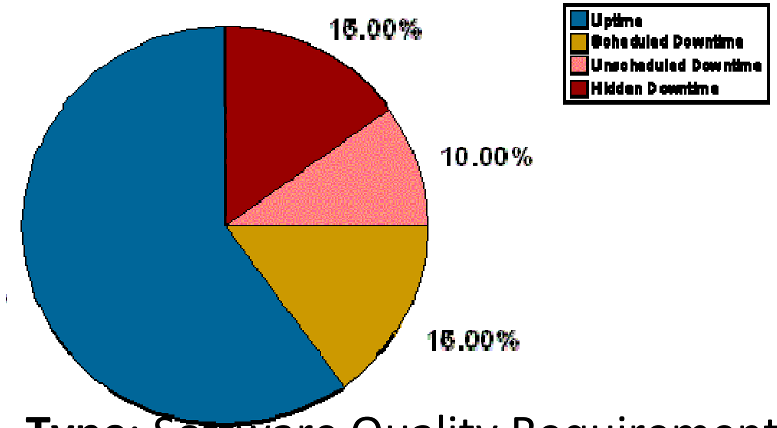

Quality of food
Hygienic quality (Safety)
Nutritional quality (health)
Functional quality (service)
Humanistic quality (environmental and moral values)
Organoleptic quality (pleasure)
Symbolic quality (cultural)
Social quality (belonging)

Cazes-Valettes, 2001.

http://www.youtube.com/watch?v=X-JiKA1vTRo = Nat King Cole "Love is…"

Rock Solid Robustness: *many splendored*

- **Type**: *Complex* Product Quality Requirement.
- **Includes**:
  - {*Software Downtime,*
  - *Restore Speed,*
  - *Testability,*
  - *Fault Prevention Capability,*
  - *Fault Isolation Capability,*
  - *Fault Analysis Capability,*
  - *Hardware Debugging Capability*}.



Software Downtime:

Type: Software Quality Requirement. **Version**: 25 October 2007.

Part of: Rock Solid Robustness.

Ambition: to have minimal downtime due to software failures <- HFA 6.1

Issue: does this not imply that there is a system wide downtime requirement?

Scale: <mean time between forced restarts for defined [Activity], for a defined [Intensity].>

Fail [Any Release or Evo Step, Activity = Recompute, Intensity = Peak Level] 14 days <- HFA 6.1.1

Goal [By 2008?, Activity = Data Acquisition, Intensity = Lowest level] : 300 days ??

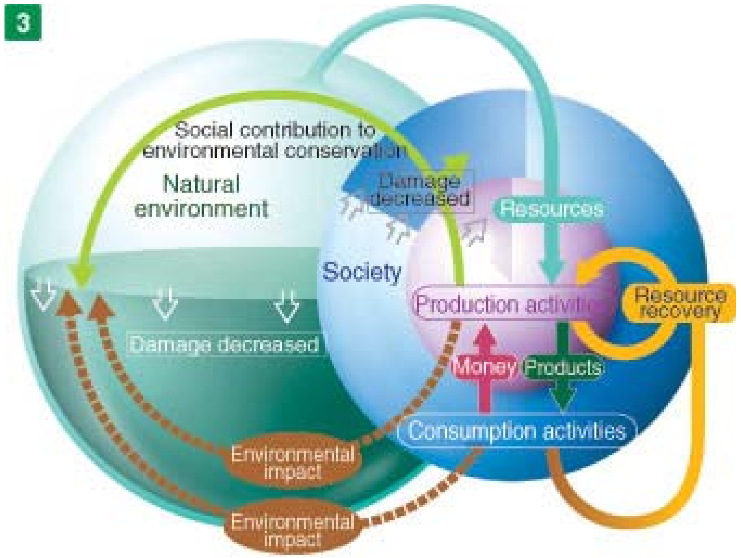
Stretch: 600 days.

Restore Speed:

Type: Software Quality Requirement. **Version**: 25 October 2007.

Part of: Rock Solid Robustness

Ambition: Should an error occur (or the user otherwise desire to do so), the system shall be able to restore the system to a previously saved state in less than 10 minutes. <-6.1.2 HFA.



Scale: Duration from Initiation of Restore to Complete and verified state of a defined [Previous: Default = Immediately Previous]] saved state.

<u>**Initiation**</u>: defined as {Operator Initiation, System Initiation, ?}. Default = Any.

Goal [Initial and all subsequent released and Evo steps] 1 minute?

Fail [Initial and all subsequent released and Evo steps] 10 minutes. <- 6.1.2 HFA

Catastrophe: 100 minutes.

Testability:

Type: Software Quality Requirement.

Part of: Rock Solid Robustness

Initial Version: 20 Oct 2006

Version: 25 October 2007.

Status: Demo draft,

Stakeholder: {Operator, Tester}.

Ambition: Rapid-duration automatic testing of <critical complex tests>, with extreme operator setup and initiation.

Scale: the duration of a defined [Volume] of testing, or a defined [Type], by a defined [Skill Level] of system operator, under defined [Operating Conditions].

Goal [All Customer Use, Volume = 1,000,000 data items, Type = WireXXXX Vs DXX, Skill = First Time Novice, Operating Conditions = Field, {Sea Or Desert}. <10 mins.

Design Hypothesis: *Tool Simulators, Reverse Cracking Tool, Generation of simulated telemetry frames entirely in software, Application specific sophistication, for drilling – recorded mode simulation by playing back the dump file, Application test harness console <-6.2.1 HFA*

Another Real (Doctored) Example:
Financial Corp. Top Level Project requirements

DO YOU SEE ANYTHING RELATED TO MAINTAINABILITY?

*1. Reduce the **costs** associated with managing redundant / regionally **disparate** systems.*

*2. **Single** global portfolio management system.*

*3. Reduce overall **spending** with a reduction in redundant initiatives.*

4. Governance structures - system agnostic.

5. All projects in project portfolio system.

*6. **Reduce development** project **spend** on low priority work with better alignment between Technology and business demand.*

*7. Project portfolio Framework, Business Value metrics for **prioritization**.*

*8. **Reduction** in **cost** over runs.*

*9. **Definition** criteria for project **success**.*

*10. Metrics and exception reporting for **cost** management.*

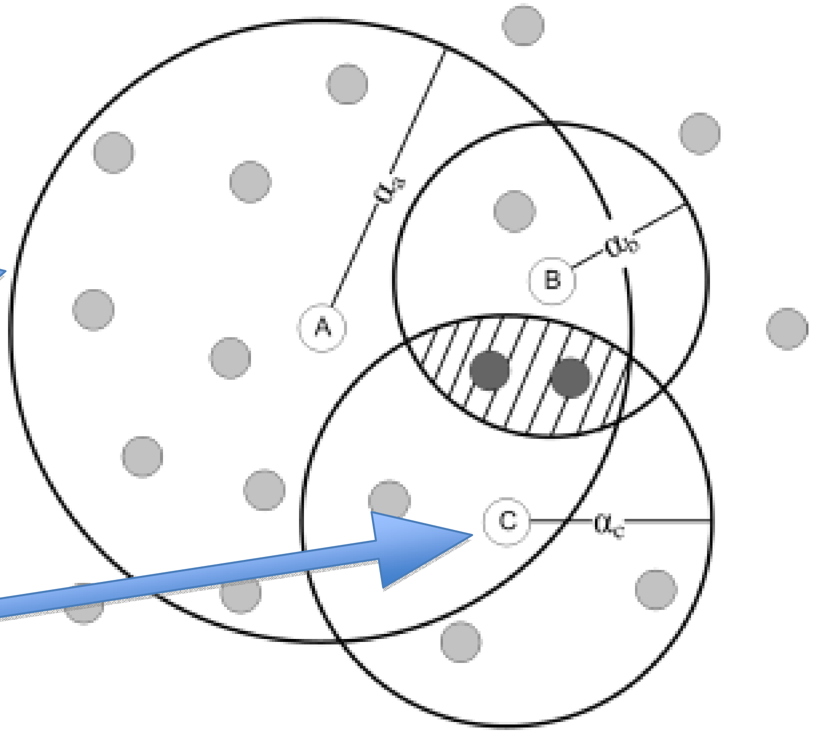
*11. Linkage of actual **costs** to forecast.*

*12. Increase **revenue** with a faster **time to market**.*

13. Knowledge management, project ramp up templates.

3. **The Multi-Level Requirement Principle.**

- The levels of maintainability we decide to require cab be
  - partly ‘**constraints**’,
    - a necessary minimum of ability to avoid failure,
  - and partly desirable **‘target’** levels
    - that are determined by what pays off to invest in.



Software Downtime: Multiple Levels

Type: Software Quality Requirement. **Version**: 25 October 2007.

Part of: Rock Solid Robustness.

Ambition: to have minimal downtime due to software failures <- HFA 6.1

Issue: does this not imply that there is a system wide downtime requirement?

Scale: <mean time between forced restarts for defined [Activity], for a defined [Intensity].>

Fail [Any Release or Evo Step, Activity = Recompute, Intensity = Peak Level] 14 days <- HFA 6.1.1

Goal [By 2008?, Activity = Data Acquisition, Intensity = Lowest level] : 300 days ??

Stretch: 600 days.

Restore Speed: Multiple Levels

***Type**: Software Quality Requirement.* ***Version**: 25 October 2007.*

***Part of**: Rock Solid Robustness*

***Ambition**: Should an error occur (or the user otherwise desire to do so), the system shall be able to restore the system to a previously saved state in less than 10 minutes. <-6.1.2 HFA.*

Scale: Duration from Initiation of Restore to Complete and verified state of a defined [Previous: Default = Immediately Previous]] saved state.

Initiation: defined as {Operator Initiation, System Initiation, ?}. Default = Any.

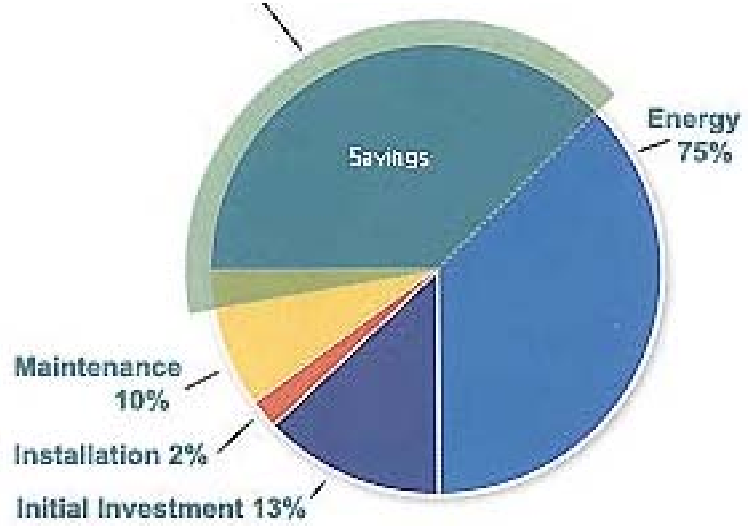
Goal [Initial and all subsequent released and Evo steps] 1 minute?

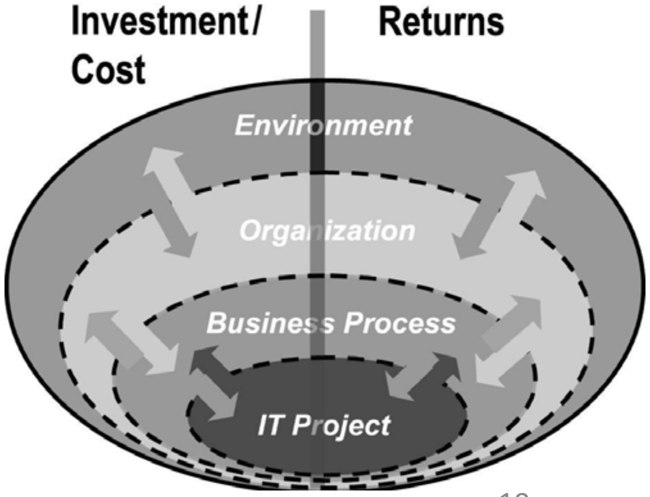
Fail [Initial and all subsequent released and Evo steps] 10 minutes. <- 6.1.2 HFA

Catastrophe: 100 minutes.

4. **The Payoff Level Principle.**

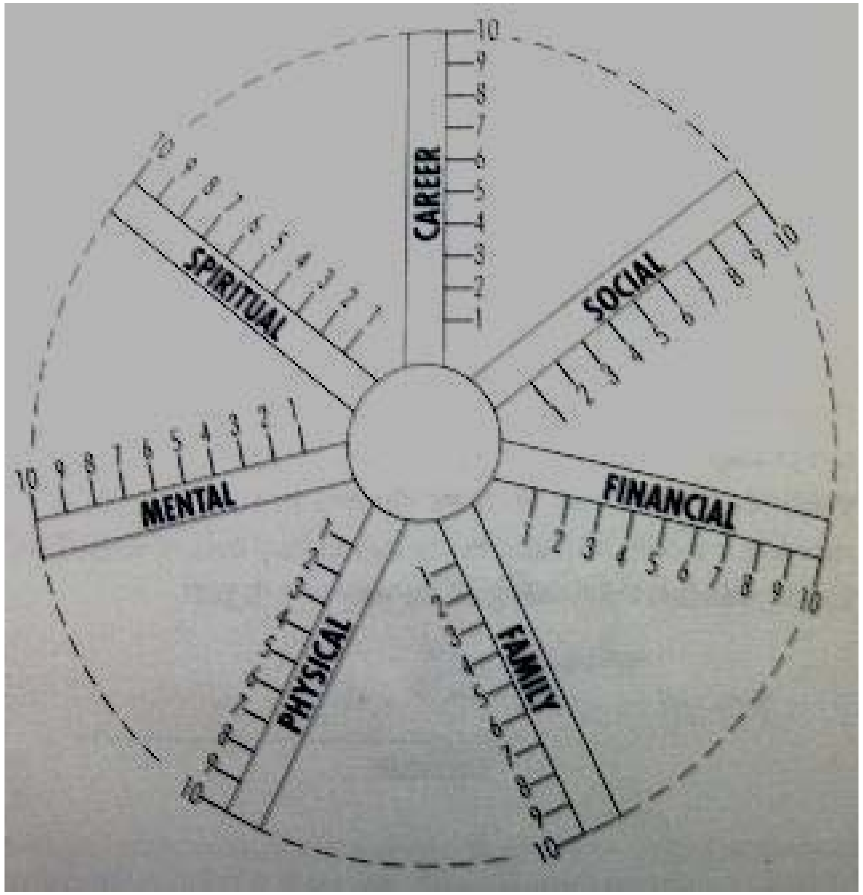
- The *levels of maintainability* it **pays off** to invest in,
  - depend on **many** factors –
  - but certainly on the system **lifetime** expectancy,
  - the **criticality**/illegality/cost of not being able to change correctly or change in time,
  - and the cost and availability of necessary skilled **professionals** to carry out the changes.





5. **The Priority Dynamics Principle.**

- The **maintainability** requirements must *compete for priority*
  - for **limited** resources
  - with all **other** requirements.
- We **cannot** simply **demand** arbitrary *desired* levels of maintainability.



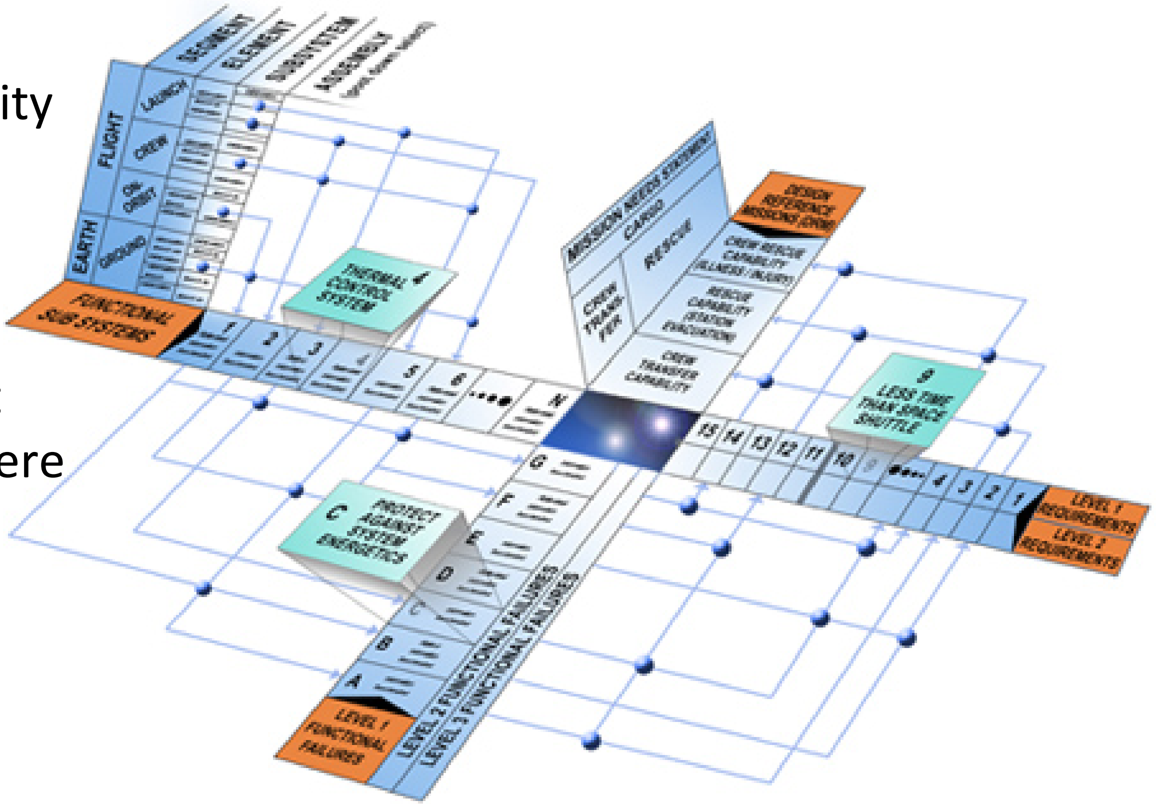
The Engineering Solution

There are many small and less critical software systems where

- engineering the maintainability would **not** be interesting,
- or would **not** pay off.
- **Nobody** cares.

This **talk is addressed** to the vast number of current situations where

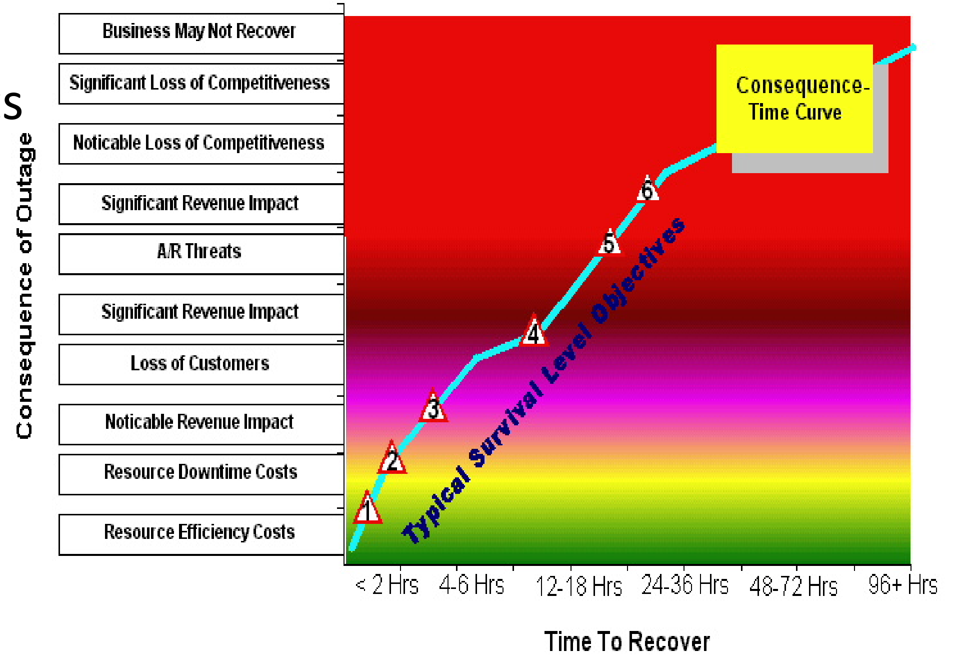
- the total **size** of software,
- the **growth** of software annually,
- the **cost** of maintenance annually – are all causing management to wonder – ‘
  - **Is there a better way?’**



The method is straightforward, and it is well-understood engineering in ‘real’ engineering disciplines.

- In simple terms it is:

1. Define the maintainability requirements *quantitatively*.
2. Design to meet those requirements, if possible and economic.
3. Implement the designs and test that they meet the required levels.
4. Quality Control that the design continues to meet the required maintainability quality levels, and take action in the case of degradation, to get back to current required levels.



At Time Mark (Tn) Survival Level Objective Occurs as Shown Below

△
T1 - Invoke Problem Management Process
T2 - Escalation and Elevation
T3 - Indication of Crisis/Disaster
T4 - Invoke Disaster Recovery/Business Resumption Measures
T5 - Executive Decision Point - May Invoke Regulatory Attention
T6 - Business Viability Decreasing

Note: Time Marks in chart are typical and will be tailored to specific client requirements based on business imperatives, legal and regulatory requirements and other factors.

Let us take a simplified tour of the method.

COMPETITIVE ENGINEERING
TOM GILB
COMPETITIVE ENGINEERING
A HANDBOOK FOR SYSTEMS ENGINEERING, REQUIREMENTS ENGINEERING, AND SOFTWARE ENGINEERING USING PLANGUAGE
GILB

Requirement specification (using ‘Planguage’ [Gilb 2005]:

Bug Fixing Speed:

Type: Software Product Quality Requirement.

Scope: Product Confirmit [Version 12.0 and on]

Ambition Level: Fast enough bug fixing so that it is a non-issue with our customers.

Scale of Measure: Average Continuous Hours from Bug occurs and is observed in any user environment, until it is correctly corrected and sufficiently tested for safe release to the field, and the change is in fact installed at, at least, one real customer, and all consequences of the bug have been recovered from at the customer level.

Meter: QA statistics on bug reports and bug fixes.

Past [Release 10.0] 36 hours <- QA Statistics

Fail [Release 12.0, Bug Level = Major] 6 hours <- QA Directors Plan

Goal [Release 12.0, Bug Level = Catastrophic] 2 hours <- QA Directors Plan.

Goal [Release 14.0, Bug Level = Catastrophic] 1 hour <- QA Directors Plan.

Planguage Intelligibility

- It should be possible to read this specification,
  - slowly,
  - even for those not trained in Planguage,
  - and to be able to explain exactly what the requirement is.
-
- Notice especially the ‘Scale of Measure’.
  - **Scale** of Measure: **Average Continuous Hours from Bug occurs and is observed in any user environment, until it is correctly corrected and sufficiently tested for safe release to the field, and the change is in fact installed at, at least, one real customer, and all consequences of the bug have been recovered from at the customer level.**
  - It encompasses the entire maintenance life cycle
    - from first bug effect observation
    - until customer level correction in practice.
  - *That is a great deal more than just some* ***programmer staring at code*** *and* ***seeing the bug*** *and* ***patching it****.*
  - The corresponding **design**
    - will have to encompass **many** processes and technologies.
-

The Breakdown into Sub-problems

Here is a list of the areas we need to design for, and quite possibly have a secondary target level for each:

1. Problem Recognition Time.

   How can we reduce the time from bug actually occurs until it is recognized and reported?

2. Administrative Delay Time:

   How can we reduce the time from bug reported, until someone begins action on it?

3. Tool Collection Time.

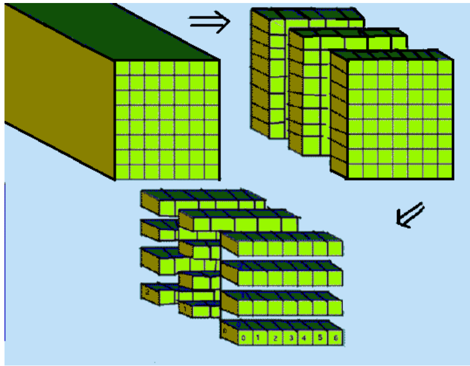
How can we reduce the time delay to collect correct, complete and updated information to analyze the bug: source code, changes, database access, reports, similar reports, test cases, test outputs.

4. Problem Analysis Time.

   Etc. for all the following phases defined, and implied, in the Scale scope above.

5. Correction Hypothesis Time

6. Quality Control Time



7. Change Time

8. Local Test Time

9. Field Pilot Test Time

10. Change Distribution Time

11. Customer Installation Time

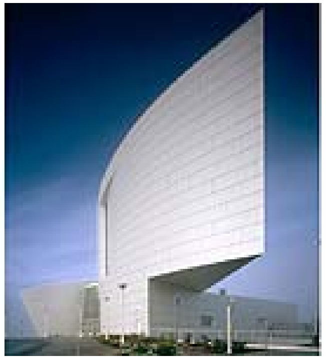
12. Customer Damage Analysis Time

13. Customer Level Recovery Time

14. Customer QC of Recovery Time

Let us take a look at a possible first draft of some design ideas:

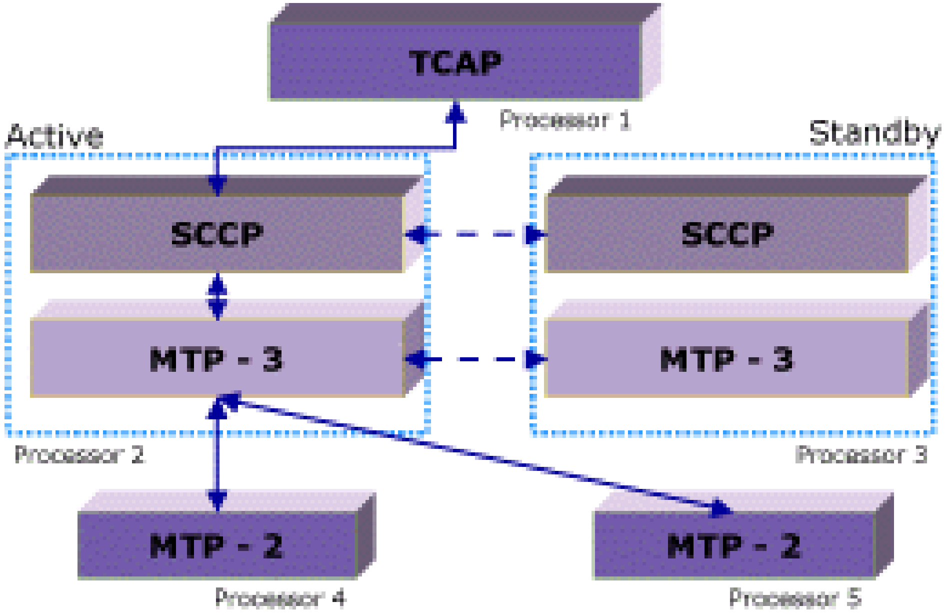
- Note: I have intentionally suggested some *dramatic* architecture,
  - in an effort to meet the *radically* improved requirement level.
- The reader need not take any design *too* seriously.
- This is an example of trying to solve the problem, using engineering techniques (redundancy)
  - that have a solid scientific history.



University of Alaska's Museum of the North in Fairbanks

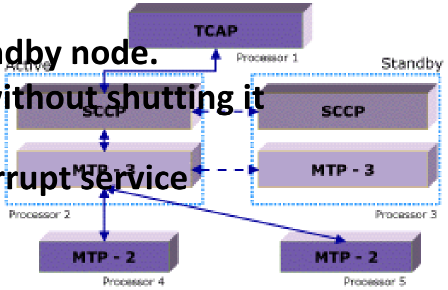
1. Problem Recognition Time.

- **Design: Automated N-version distinct software comparison [Inacio 1998]**
  - **at selected critical customer sites,**
  - **to detect potential bugs automatically.**



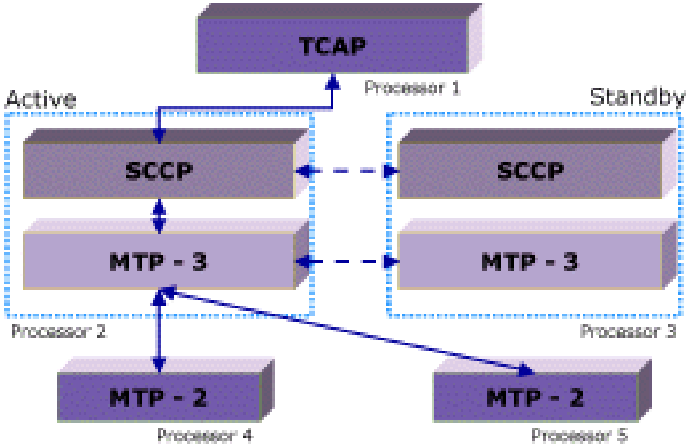
Trillium | Distributed Fault-Tolerant/High-Availability (DFT/HA) Core

- **Complete recovery during failure.**
  - **This feature is available in both pure fault-tolerant and distributed fault-tolerant systems.**
  - **When a failure occurs, failed protocol layers are able to completely recover stable state information.**
  - **All protocol resources present in a stable state during the failure are maintained on the standby.**
- **Application restart on processor loss.**
  - **This feature is applicable to pure distributed systems. If a processor in a pure distributed system fails, applications on the failed processor may be restarted on available processors to provide service for subsequent user traffic.**
- **Survive up to n-1 faults.**
  - **DFT protocol layers may survive up to n-1 faults without loss of service where n is the number of processors over which the protocol layer was distributed.**
  - **With the lost application restart feature enabled, a distributed protocol layer may continue to provide full service until the last processor in the system fails.**
  - **User defined system operations. Advanced distributed system operations such as dynamic load balancing may be implemented using basic services provided by the core software.**
- **Graceful node shutdown.**
  - **The system manager provides an operation to gracefully shutdown a node and an option to redistribute the protocol load onto remaining processors in the system**
  - **. The load redistribution is completely transparent to the system users.**
- **Maintenance operations.**
  - **The system manager provides an operation to swap the states of an active and standby node.**
  - **This functionality may be used to perform maintenance operations on the system without shutting it down**
  - **. These operations are completely transparent to the system users and will not interrupt service provided by the system.**



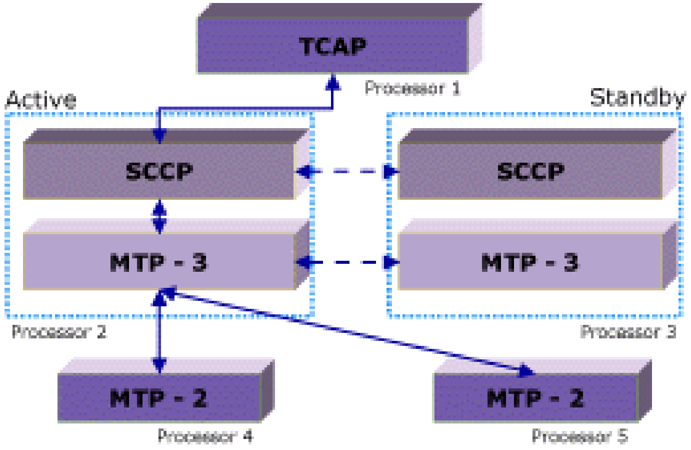
2. Administrative Delay Time:

- **Design: Direct digital report**
  - **from distinct software discrepancies**
  - **to our global,**
    - **3 zone,**
    - **24/7**
    - **bug analysis service.**



3. Tool Collection Time.

- **Design: All necessary tools are electronic,**
  - **and collection is based on**
    - **customers installed version and its fixes.**
  - **The distinct software, bug capture**
    - **collects local input sequences.**



4. Problem Analysis Time.

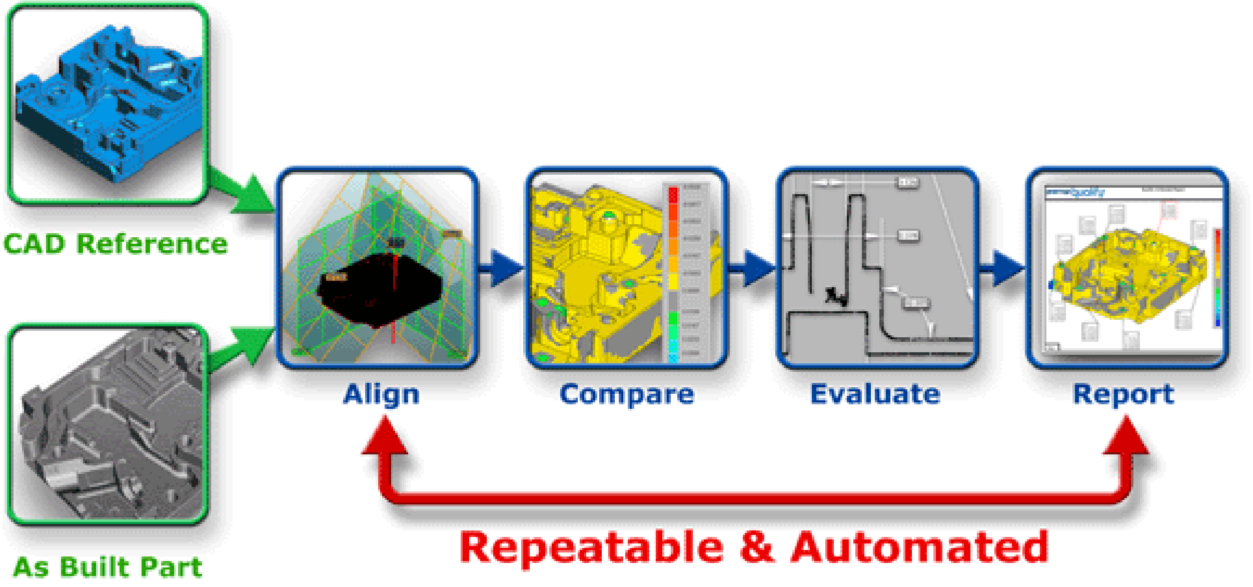
- **Analyst Selection:**
  - **Design: The fastest bug analysts are**
    - **selected based on actual past performance statistics, and**
    - **rewarded in direct relation to their timing**
      - **for analyzing root cause, or correct fix.**

5. Correction Hypothesis Time

- **Design: Same design as <u>Analyst Selection</u>,**
  - **but applies to correct change specification speed statistics.**

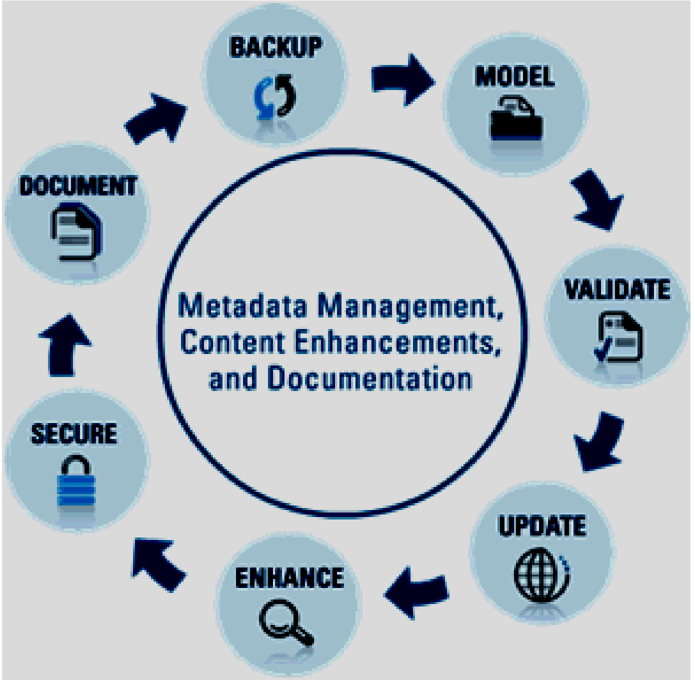
6. Quality Control Time

- **Design: Rigorous**
  - **30 minute or less inspection**
  - **of change spec by other bug analysts,**
  - **with reward for finding major defects**
    - **as judged by our defect standards.**



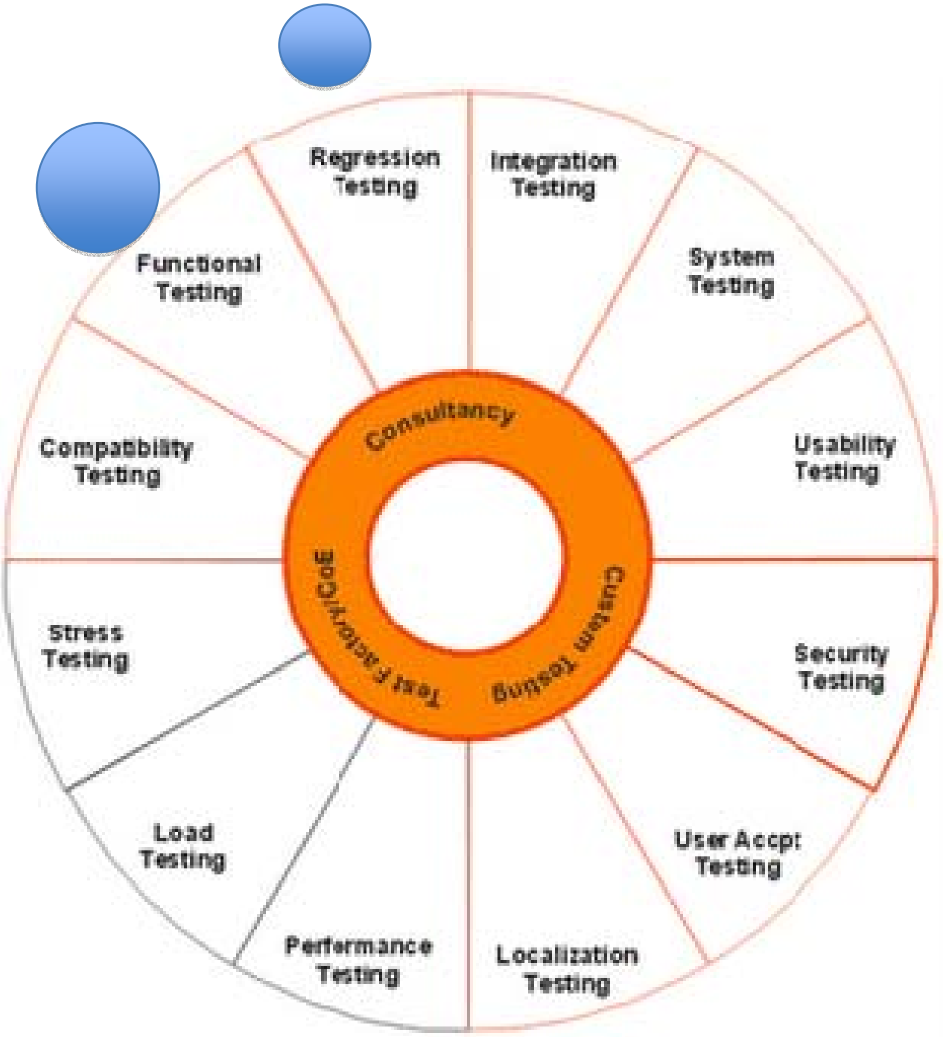
7. Change Time

- **Design: Changes are applied**
  - **in parallel with QC,**
  - **and modified only if change defects found in QC.**


BACKUP
MODEL
VALIDATE
UPDATE
ENHANCE
SECURE
DOCUMENT
Metadata Management, Content Enhancements, and Documentation

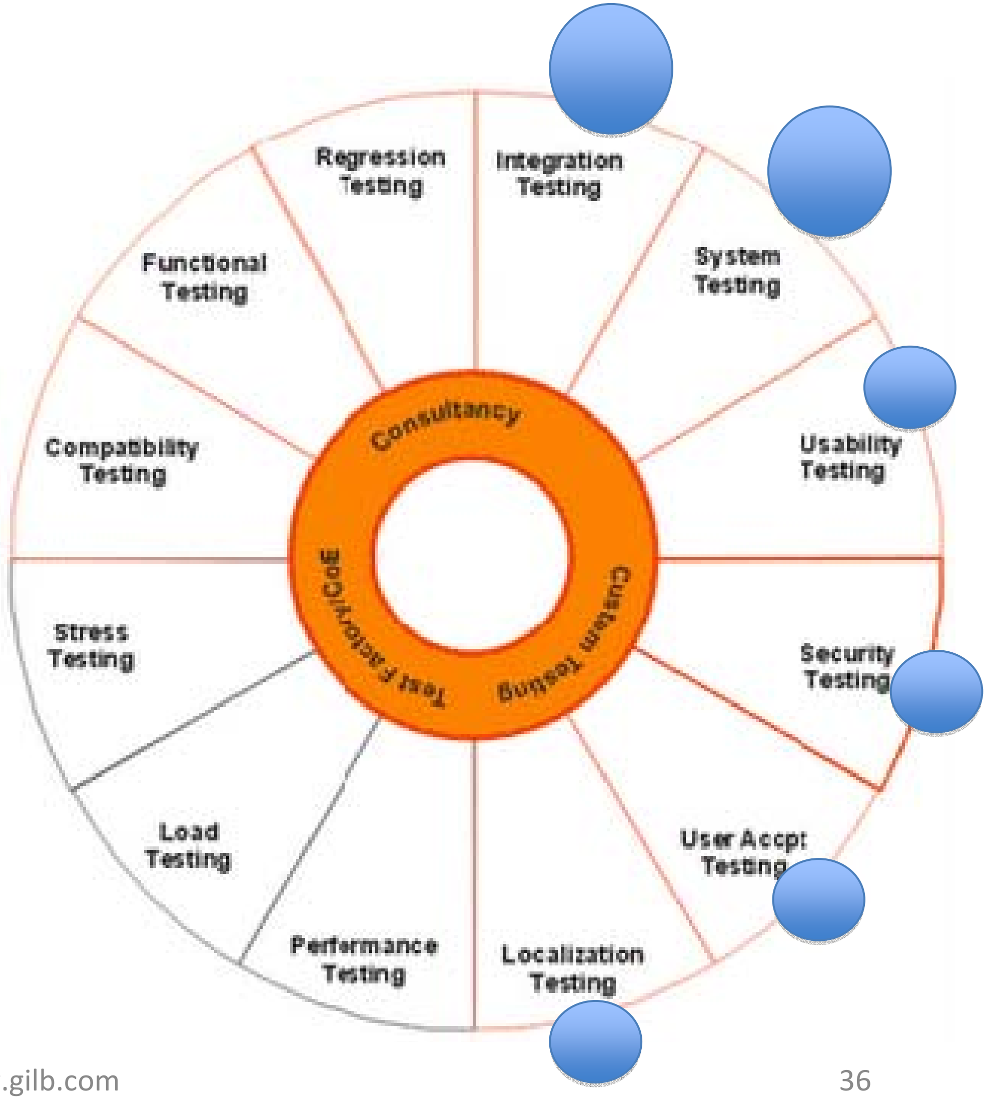
8. Local Test Time

- **Design: Automated Test. Based on distinct software (2 independent) changes**
  - **to distinct modules, and**
  - **running reasonable test sets,**
  - **until further notice**
  - **or failure.**



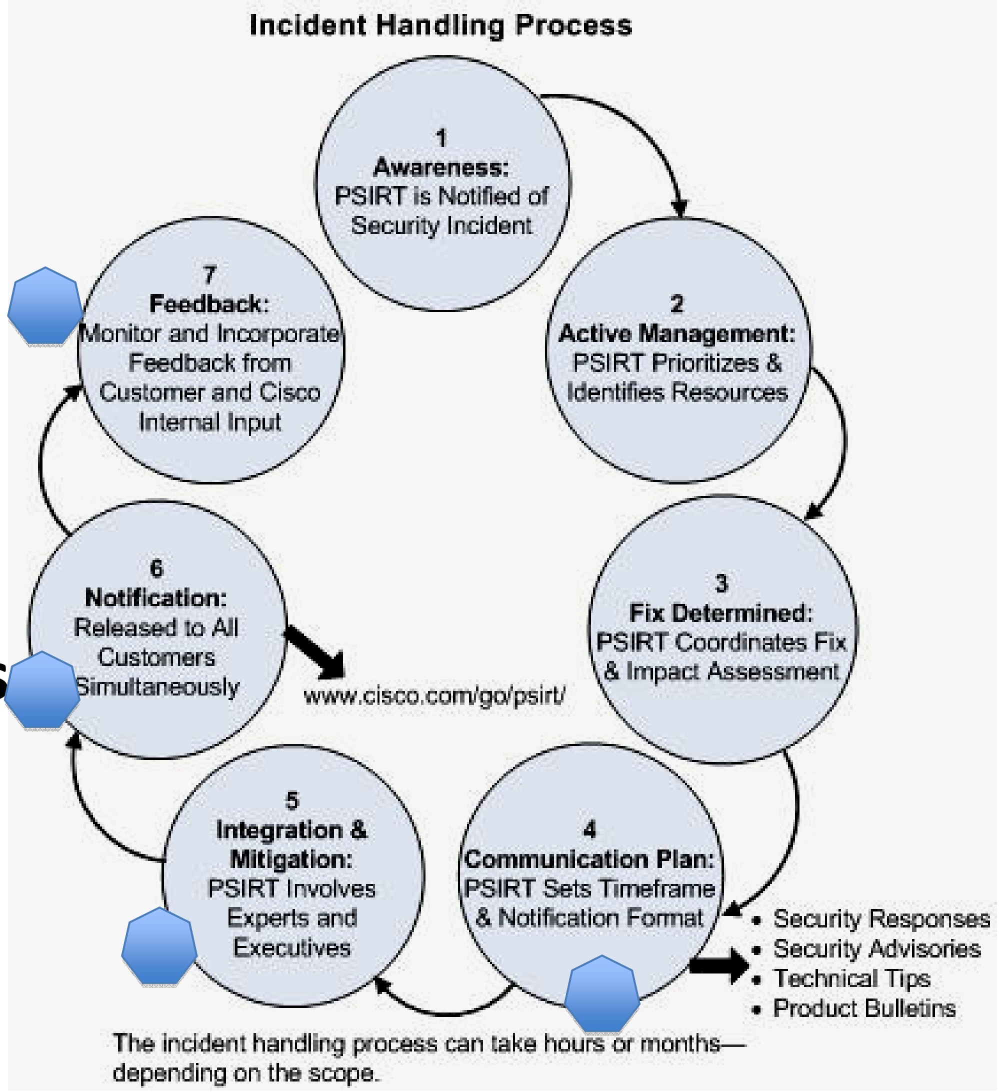
9. Field Pilot Test Time

- **Design:**
  - **After 30 minutes successful Local Test**
  - **the changes are implemented**
    - **at a customer pilot site**
      - **for more realistic testing,**
        - **» in operation,**
        - **» in distinct software safe mode.**



10. Change Distribution Time

- **Design: All necessary changes are**
  - **readied and**
  - **uploaded for customer download,**
  - **even before Local Tests Begin,**
  - **and changed only**
    - **if tests fail.**



Incident Handling Process

1 **Awareness:** PSIRT is Notified of Security Incident

2 **Active Management:** PSIRT Prioritizes & Identifies Resources

3 **Fix Determined:** PSIRT Coordinates Fix & Impact Assessment

4 **Communication Plan:** PSIRT Sets Timeframe & Notification Format

- Security Responses
- Security Advisories
- Technical Tips
- Product Bulletins

5 **Integration & Mitigation:** PSIRT Involves Experts and Executives

6 **Notification:** Released to All Customers Simultaneously

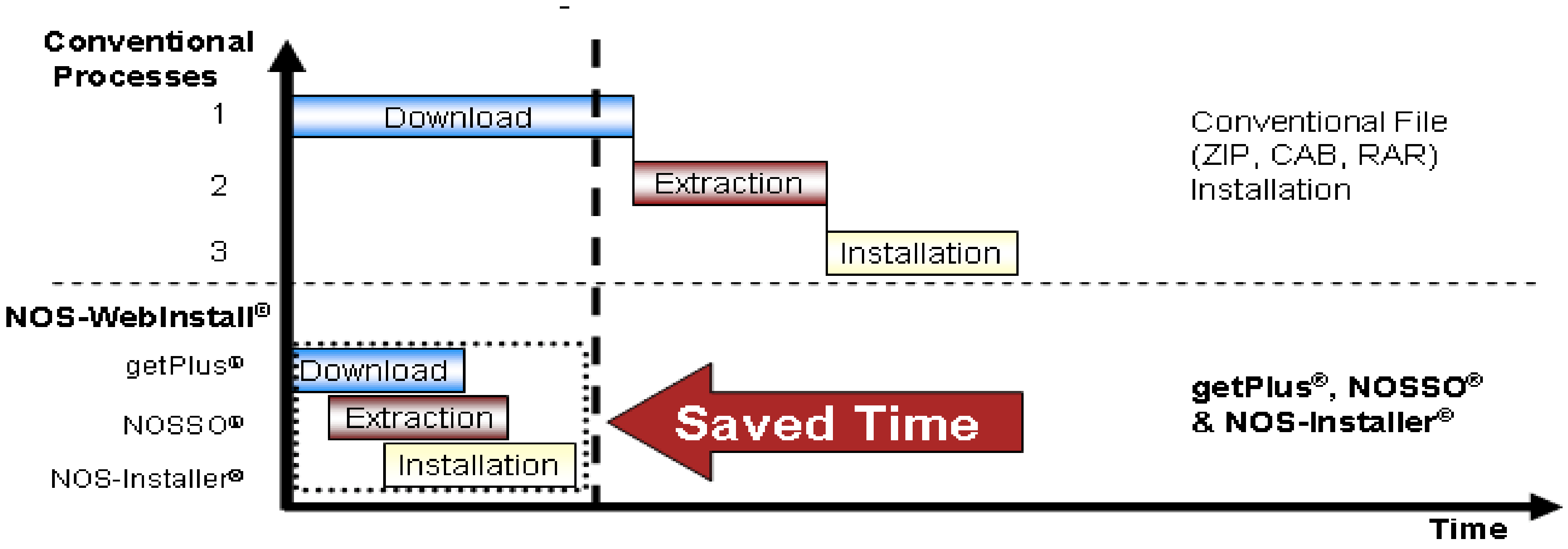
www.cisco.com/go/psirt/

7 **Feedback:** Monitor and Incorporate Feedback from Customer and Cisco Internal Input

The incident handling process can take hours or months—depending on the scope.

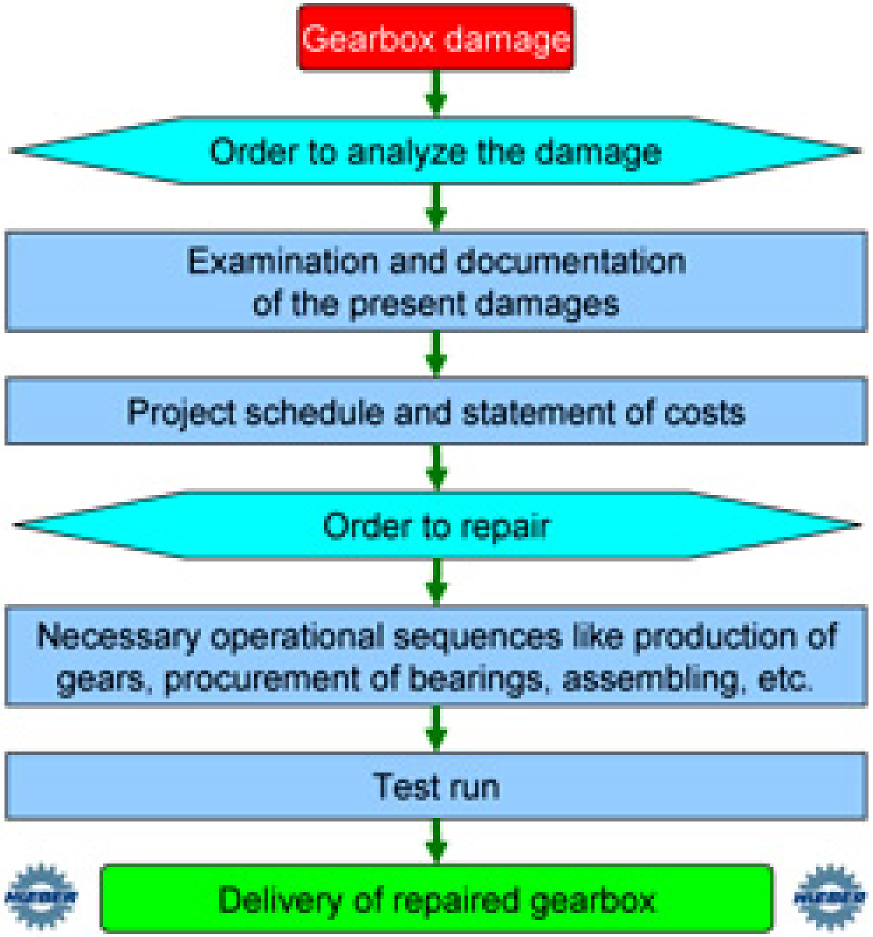
11. Customer Installation Time

- **Design: Customer is given options of**
  - **manual or**
  - **automatic changes,**
  - **under given circumstances**



12. Customer Damage Analysis Time

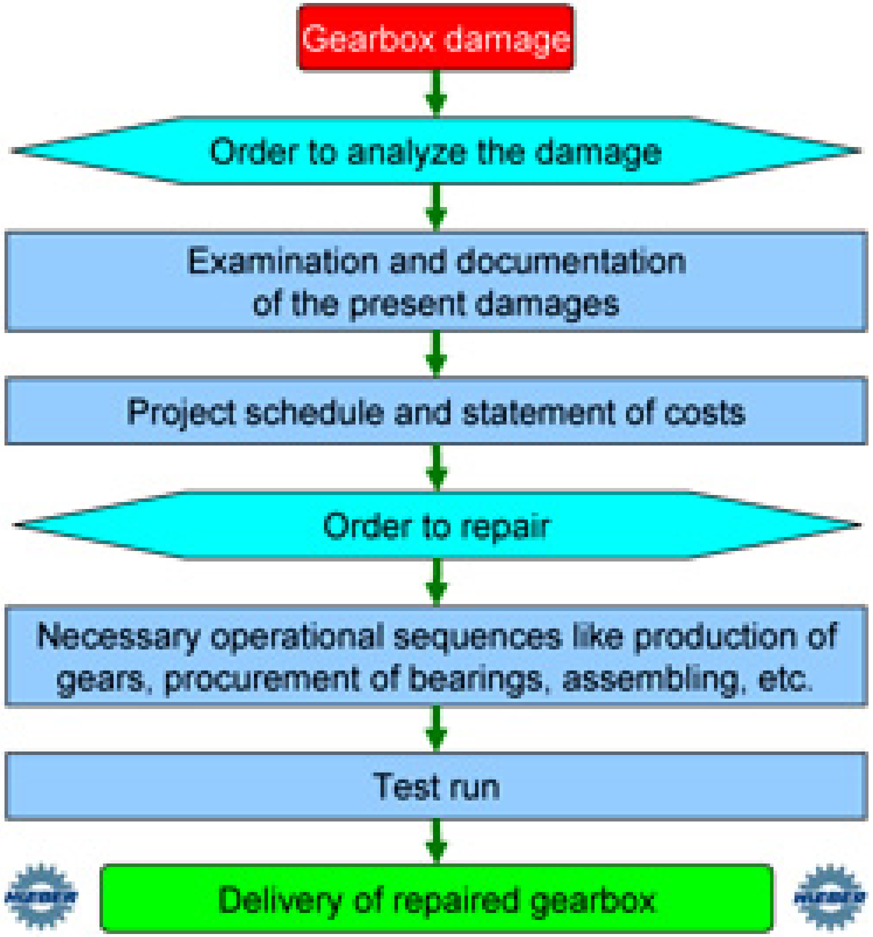
- Design:
- <local customer solution>.
- We don’t have good automation here.
- Assume none until proven otherwise.
- We need to be aware of
  - all reports **sent**
  - and databases updated that may need correction.
-



Gearbox damage
Order to analyze the damage
Examination and documentation of the present damages
Project schedule and statement of costs
Order to repair
Necessary operational sequences like production of gears, procurement of bearings, assembling, etc.
Test run
Delivery of repaired gearbox

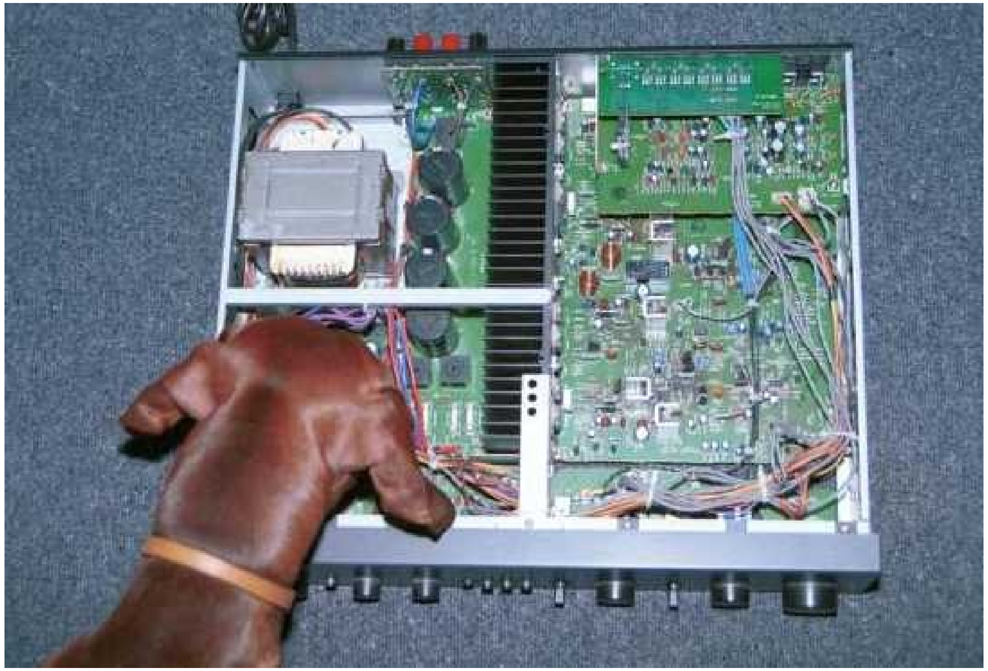
13. Customer-Level-Recovery Time

- **Design:**
- **same problem as Customer Damage Analysis Time**
- **may be highly local and manual.**
- **Is it really out of our control?**


Gearbox damage
Order to analyze the damage
Examination and documentation of the present damages
Project schedule and statement of costs
Order to repair
Necessary operational sequences like production of gears, procurement of bearings, assembling, etc.
Test run
Delivery of repaired gearbox

14. Customer QC of Recovery, Time.

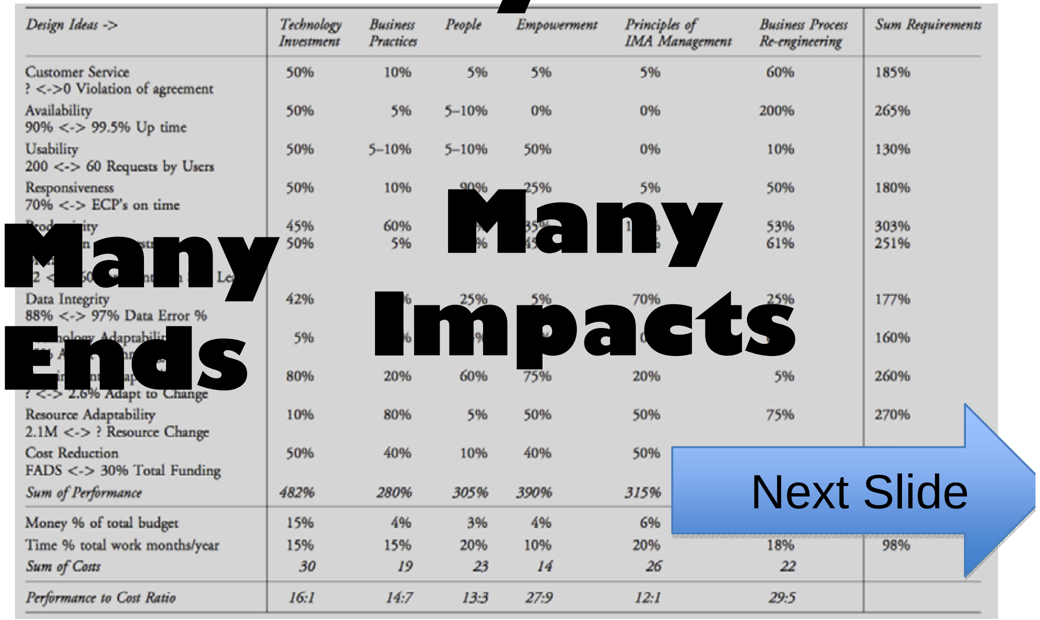
- **Design:**
- **30-minute Quality Control**
  - **of recovery results,**
  - **assisted by our quality standards,**
  - **and *for critical customers***
  - **QC By our staff,**
    - **From our office**
    - **or on customer site.**
-



Main Point

- My main point is
  - that each sub-process of the maintenance operation
  - tends to require a separate and distinct design (1 or more designs each).
- There is nothing simple
  - like software people seem to believe,
  - that better code structures,
  - coding practices, documentation,
  - and tools
  - will solve the

Many Means



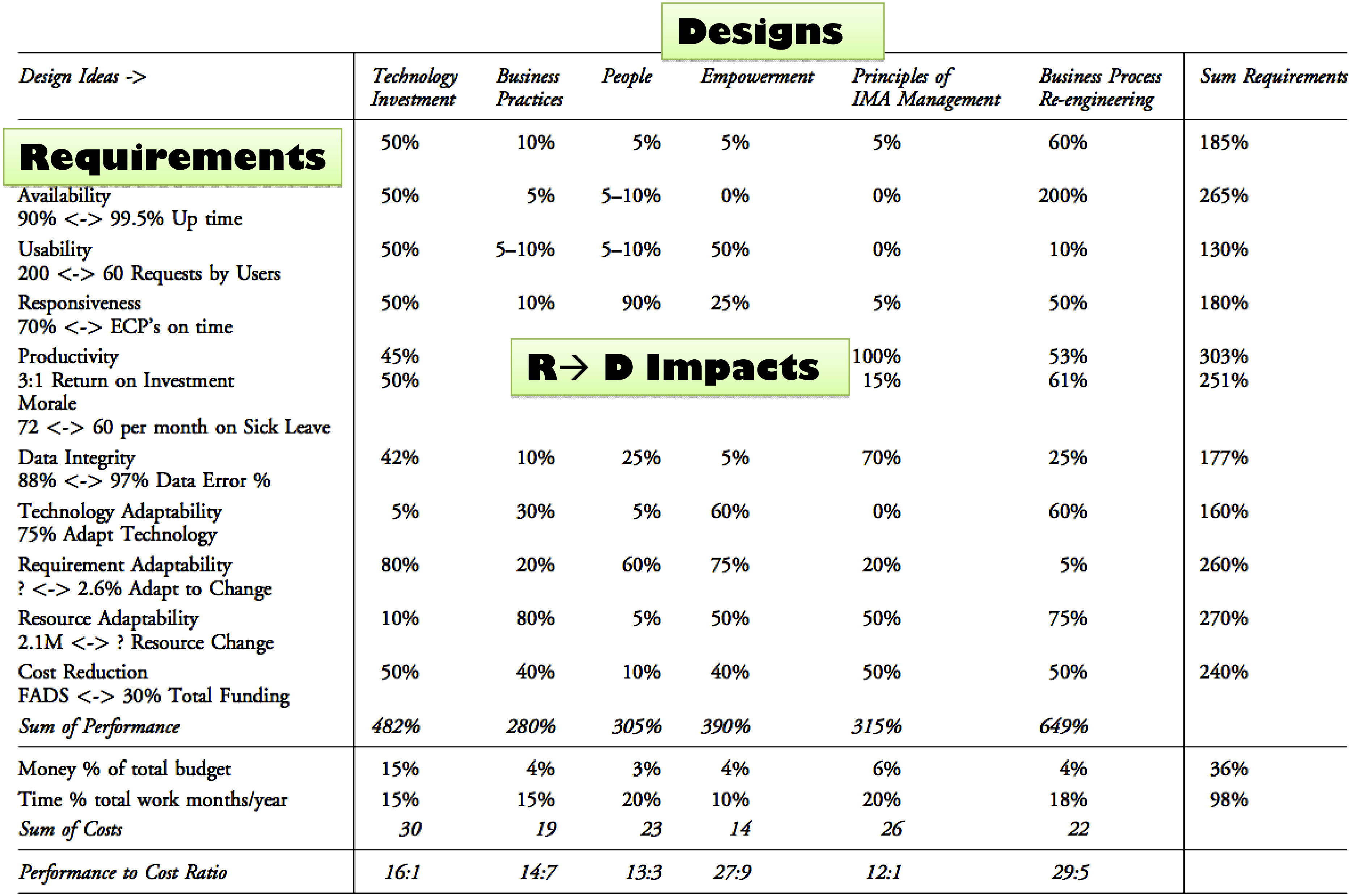
Design Ideas ->	*Technology Investment*	*Business Practices*	*People*	*Empowerment*	*Principles of IMA Management*	*Business Process Re-engineering*	*Sum Requirements*
Customer Service ? <->0 Violation of agreement	50%	10%	5%	5%	5%	60%	185%
Availability 90% <-> 99.5% Up time	50%	5%	5–10%	0%	0%	200%	265%
Usability 200 <-> 60 Requests by Users	50%	5–10%	5–10%	50%	0%	10%	130%
Responsiveness 70% <-> ECP's on time	50%	10%	90%	25%	5%	50%	180%
Prod[illegible]ity	45%	60%	[illegible]	35%	[illegible]	53%	303%
[illegible]	50%	5%	[illegible]	[illegible]	[illegible]	61%	251%
Data Integrity 88% <-> 97% Data Error %	42%	[illegible]	25%	5%	70%	25%	177%
[illegible]	5%	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	160%
[illegible] ? <-> 2.6% Adapt to Change	80%	20%	60%	75%	20%	5%	260%
Resource Adaptability 2.1M <-> ? Resource Change	10%	80%	5%	50%	50%	75%	270%
Cost Reduction FADS <-> 30% Total Funding	50%	40%	10%	40%	50%	[illegible]	[illegible]
Sum of Performance	*482%*	*280%*	*305%*	*390%*	*315%*	[illegible]	
Money % of total budget	15%	4%	3%	4%	6%	[illegible]	[illegible]
Time % total work months/year	15%	15%	20%	10%	20%	18%	98%
Sum of Costs	*30*	*19*	*23*	*14*	*26*	*22*	
Performance to Cost Ratio	*16:1*	*14:7*	*13:3*	*27:9*	*12:1*	*29:5*	

Many Ends

Many Impacts

Next Slide

DoDef: Persinscom Impact Estimation Table:



Designs

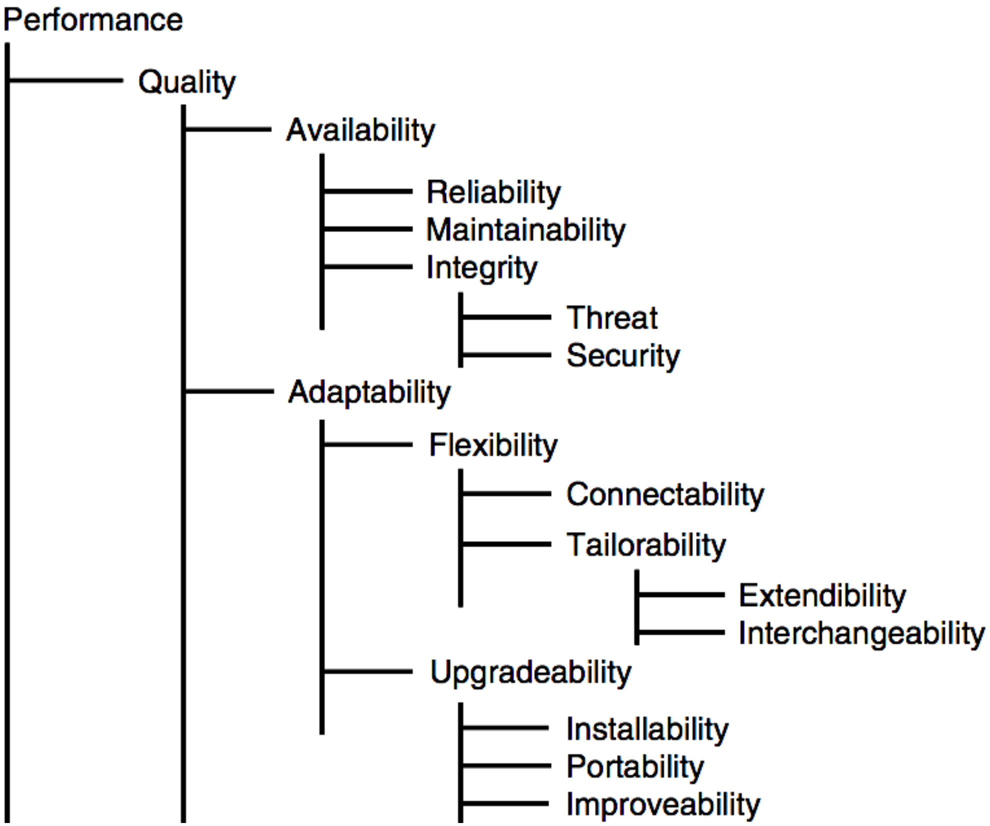
Requirements

R→ D Impacts

Design Ideas ->	*Technology Investment*	*Business Practices*	*People*	*Empowerment*	*Principles of IMA Management*	*Business Process Re-engineering*	*Sum Requirements*
	50%	10%	5%	5%	5%	60%	185%
Availability 90% <-> 99.5% Up time	50%	5%	5–10%	0%	0%	200%	265%
Usability 200 <-> 60 Requests by Users	50%	5–10%	5–10%	50%	0%	10%	130%
Responsiveness 70% <-> ECP's on time	50%	10%	90%	25%	5%	50%	180%
Productivity 3:1 Return on Investment	45%				100%	53%	303%
Morale 72 <-> 60 per month on Sick Leave	50%				15%	61%	251%
Data Integrity 88% <-> 97% Data Error %	42%	10%	25%	5%	70%	25%	177%
Technology Adaptability 75% Adapt Technology	5%	30%	5%	60%	0%	60%	160%
Requirement Adaptability ? <-> 2.6% Adapt to Change	80%	20%	60%	75%	20%	5%	260%
Resource Adaptability 2.1M <-> ? Resource Change	10%	80%	5%	50%	50%	75%	270%
Cost Reduction FADS <-> 30% Total Funding	50%	40%	10%	40%	50%	50%	240%
Sum of Performance	*482%*	*280%*	*305%*	*390%*	*315%*	*649%*	
Money % of total budget	15%	4%	3%	4%	6%	4%	36%
Time % total work months/year	15%	15%	20%	10%	20%	18%	98%
Sum of Costs	*30*	*19*	*23*	*14*	*26*	*22*	
Performance to Cost Ratio	*16:1*	*14:7*	*13:3*	*27:9*	*12:1*	*29:5*	

Broader Maintainability Concepts

- Maintainability in the strict engineering sense is usually taken to mean **bug fixing.**
- I have however been using it *thus far* to describe ***any software change activity or process.***
- We could perhaps better call it **'software change ability'.**
- Different classes of change, will have different requirements related to them,
  - and consequently **different technical solutions**.
- It is important that we be very clear
  - in setting requirements,
  - and doing corresponding design,
  - exactly what **types of change** we are talking about.


Performance
Quality
Availability
Reliability
Maintainability
Integrity
Threat
Security
Adaptability
Flexibility
Connectability
Tailorability
Extendibility
Interchangeability
Upgradeability
Installability
Portability
Improveability

General ‘Change Attribute’ **Tailoring**

- The following <u>slides </u>will give a **general set of patterns** for
  - defining and distinguishing *different* **classes** of ‘maintenance’.
- But in your *real* world, you will want to **tailor** the definitions to *your* domain.
  - You can initially tailor using the ‘**Scale’** of measure definition.
  - And continued tailoring can be done by defining **[conditions]** in the requirement level qualifier.

Scale:
% of transactions successfully completed by defined [Person] doing defined [Task].

Goal **[Task = Update, Person = New Hire, Deadline = Phase 3]** 60%

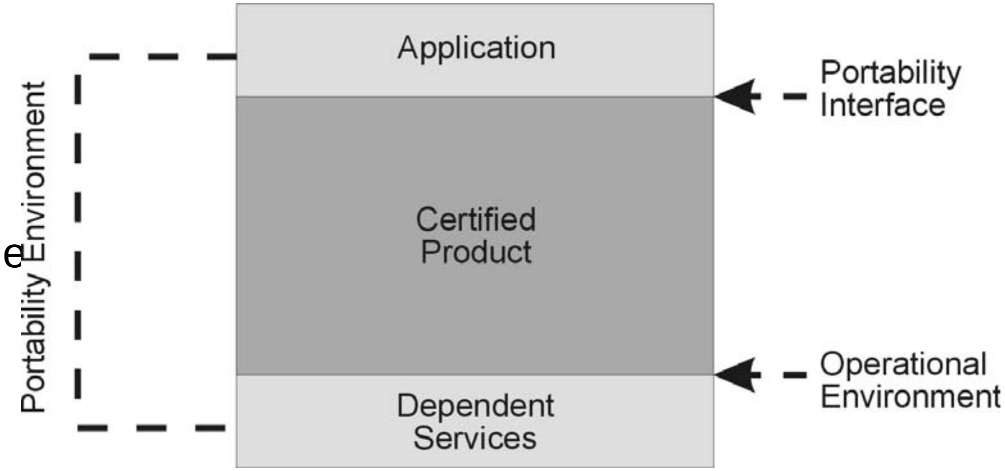
A generic set of performance measures, including several related to change.

For example:

<u>Code Portability:</u>

Scale:

Effort in Hours needed to Port each 1000 Non-Commentary Lines of Code from a defined [**Home Environment**] to a defined [**Target Environment**], using defined [**Tools**] and defined [**Personnel**].



Goal

[Home Environment = {.net, Oracle,} ,

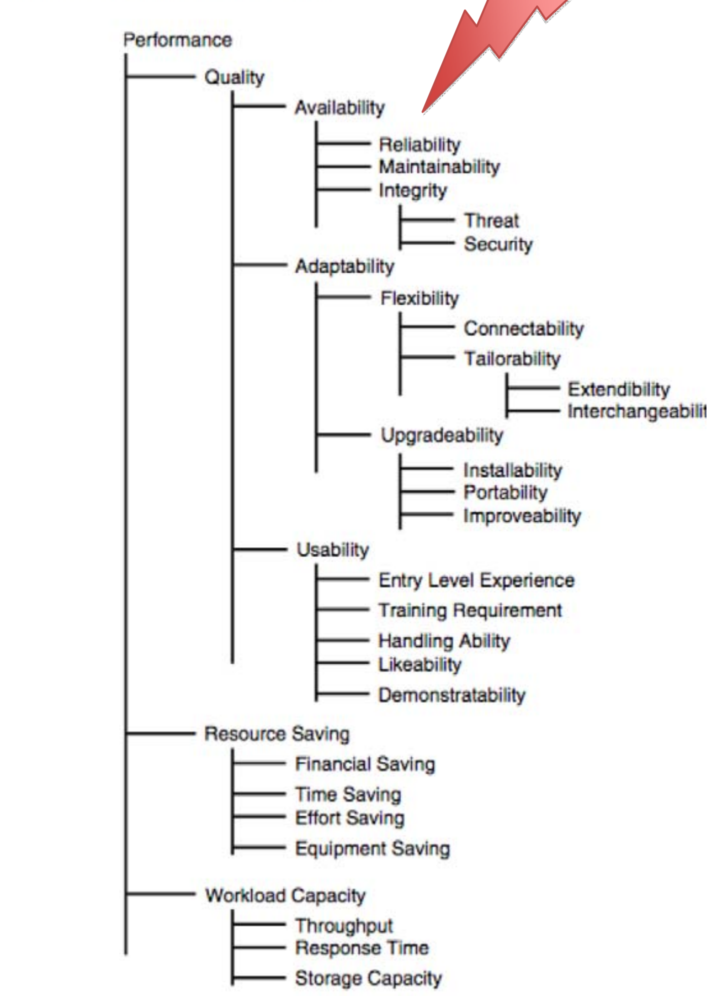
Target Environment = {Java++, Open Source, Linux},

Tools = Convert Open ,

Personnel = {Experienced Experts, India}] **60** hours.

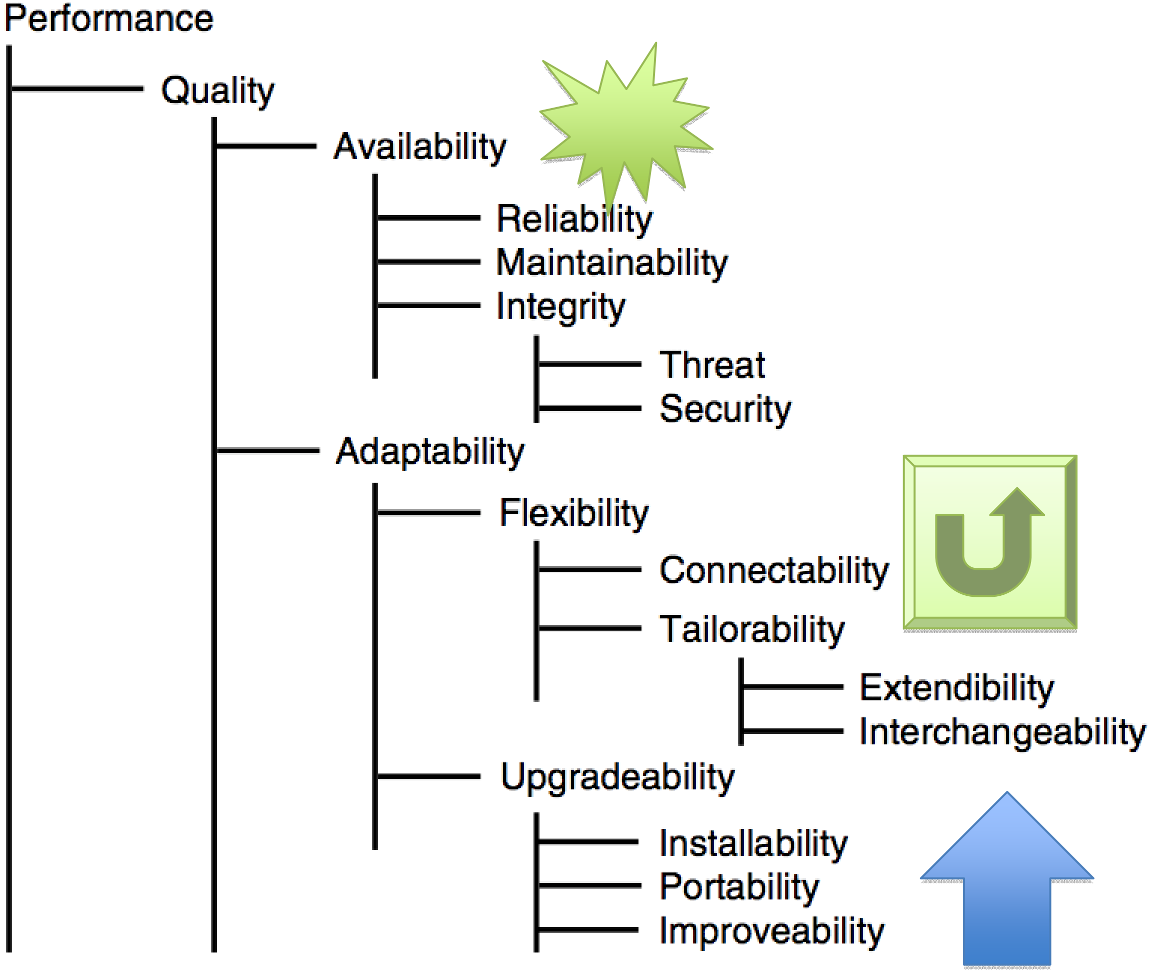
A Generic Set of Performance measures – including several related to ‘change’

154 Competitive Engineering



- Performance
  - Quality
    - Availability
      - Reliability
      - Maintainability
      - Integrity
        - Threat
        - Security
    - Adaptability
      - Flexibility
        - Connectability
        - Tailorability
          - Extendibility
          - Interchangeabilit
      - Upgradeability
        - Installability
        - Portability
        - Improveability
    - Usability
      - Entry Level Experience
      - Training Requirement
      - Handling Ability
      - Likeability
      - Demonstratability
  - Resource Saving
    - Financial Saving
    - Time Saving
    - Effort Saving
    - Equipment Saving
  - Workload Capacity
    - Throughput
    - Response Time
    - Storage Capacity

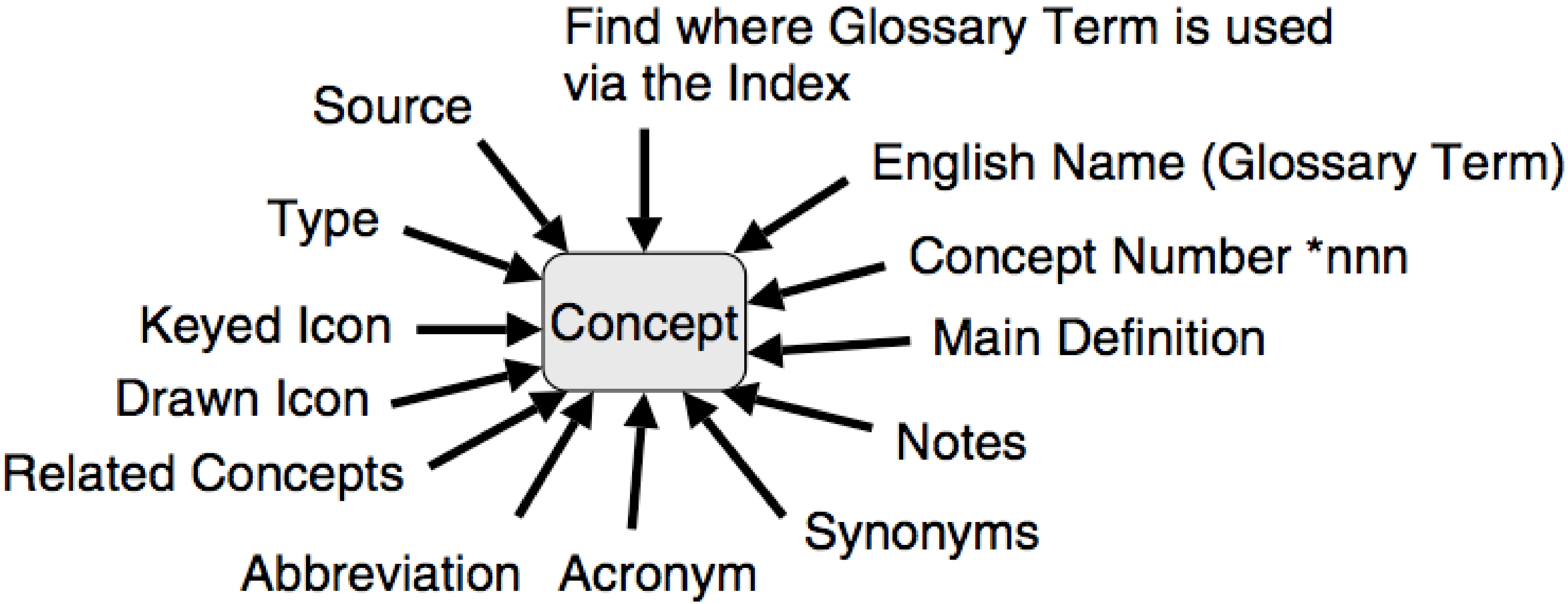
Figure 5.3
One decomposition possibility for performance attributes with emphasis on the detail of the quality attributes.



- Performance
  - Quality
    - Availability
      - Reliability
      - Maintainability
      - Integrity
        - Threat
        - Security
    - Adaptability
      - Flexibility
        - Connectability
        - Tailorability
          - Extendibility
          - Interchangeability
      - Upgradeability
        - Installability
        - Portability
        - Improveability

The *attribute names* used are arbitrary choices by the author.

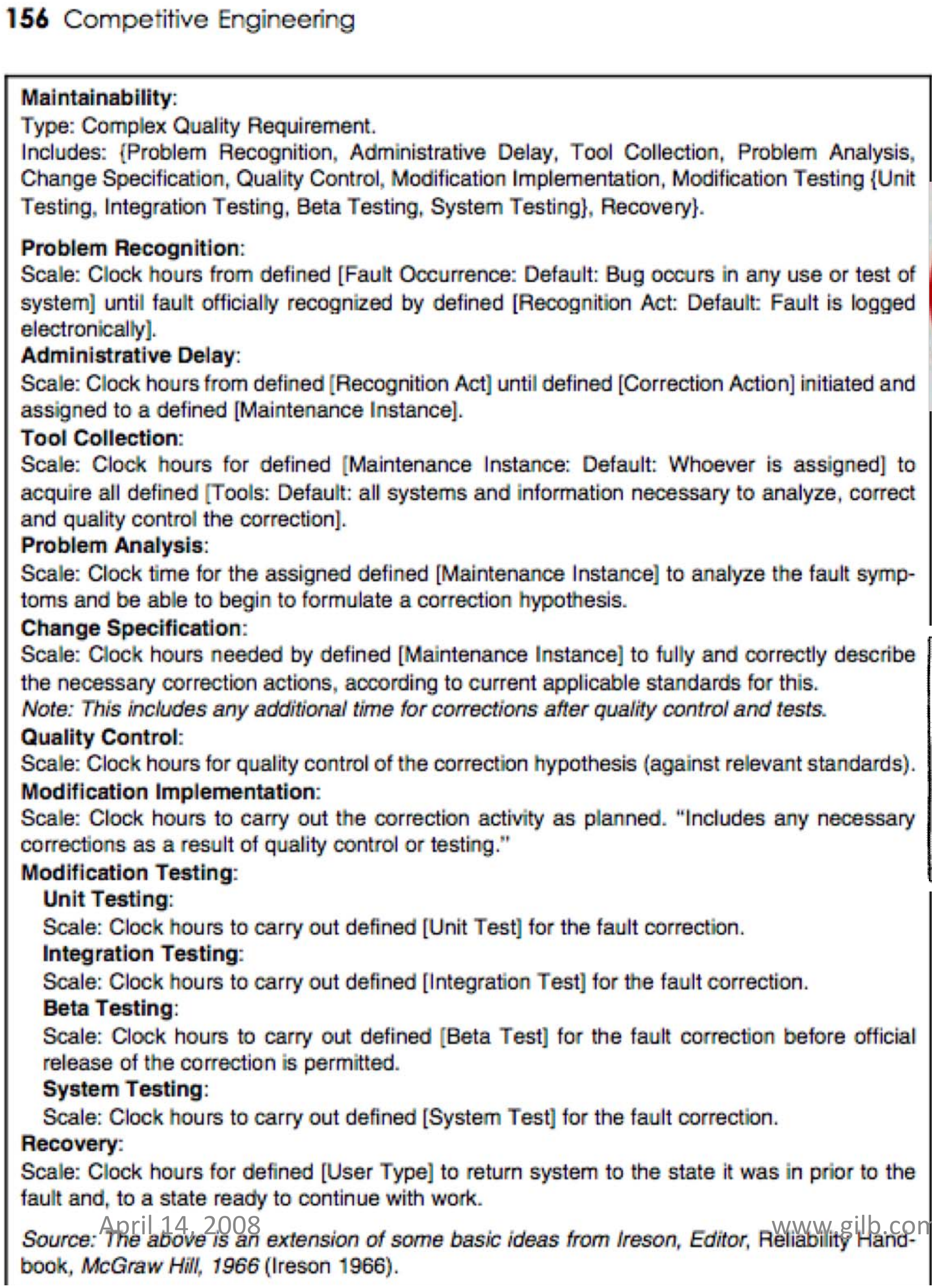
- They only start to take on meaning when defined,
  - with a Scale of measure.
- There are no accepted or acceptable standards here,
  - and certainly not for software.
  - Even in hardware engineering, there is an accepted **pattern** – such as "Scale: Mean Time to Repair".
  - But it is accepted that we have to *further* define such concepts *locally*,
    - such as the meaning of 'Repair'.



Maintainability Measures

- Here are some of the general **patterns** we can use to define and distinguish the different classes of change processes on software.
- First the 'Bug Fixing' pattern (from which we derived the example at the beginning of this talk).

Maintainability components, derived from a hardware engineering view, adopted for software.



156 Competitive Engineering

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.

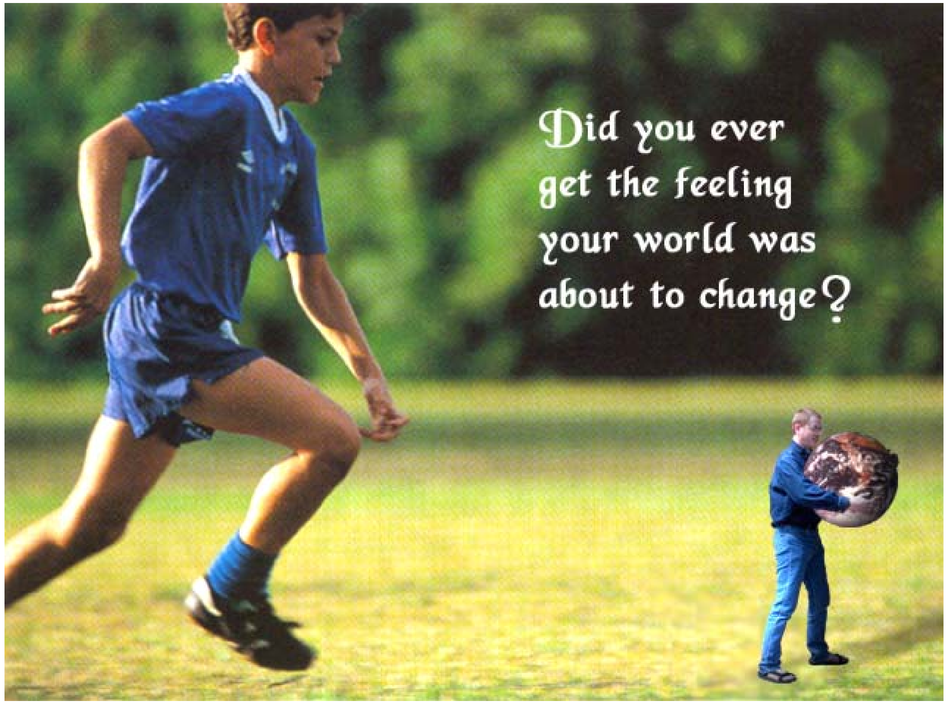
Recovery:
Scale: Clock hours for defined [User Type] to return system to the state it was in prior to the fault and, to a state ready to continue with work.

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

OUR GOAL IS TO WRITE BUG-FREE SOFTWARE. I'LL PAY A TEN-DOLLAR BONUS FOR EVERY BUG YOU FIND AND FIX.
YAHOO!
WE'RE RICH
YES!!! YES!!! YES!!!
I HOPE THIS DRIVES THE RIGHT BEHAVIOR.
I'M GONNA WRITE ME A NEW MINIVAN THIS AFTERNOON!
S. Adams E-mail: SCOTTADAMS@AOL.COM
© 1995 United Feature Syndicate, Inc.(NYC)

Notice that *Maintainability* in the narrow sense (fix bugs) is quite separate from other ‘Adaptability’ concepts.

- This is normal *engineering*,
  - Which places fault repair together with reliability and availability;
  - Those 3 determine the *immediate* operational characteristics of the system.
- The **other forms of adaptability** are more about *potential* **future upgrades** to the system,
  - *change*, rather than *repair*.
- Change and repair, have **in common** that
  - our system *architecture* has to make it easy to change, analyze and test.
- The system *itself* is unaware of
  - whether we are *correcting* a *fault*
  - or *improving* the system.
- The consequence is that
  - *much of the maintenance-impacting ‘design’ or ‘architecture’*
  - **benefits**
  - *most of the types of maintenance (fix **and** adapt)*.



Here are a *generic* set of definitions for the '*Adaptability*' concepts.

Adaptability: 'The **efficiency** with which a system can be changed.'

Gist: Adaptability is a measure of a system's ability to change.

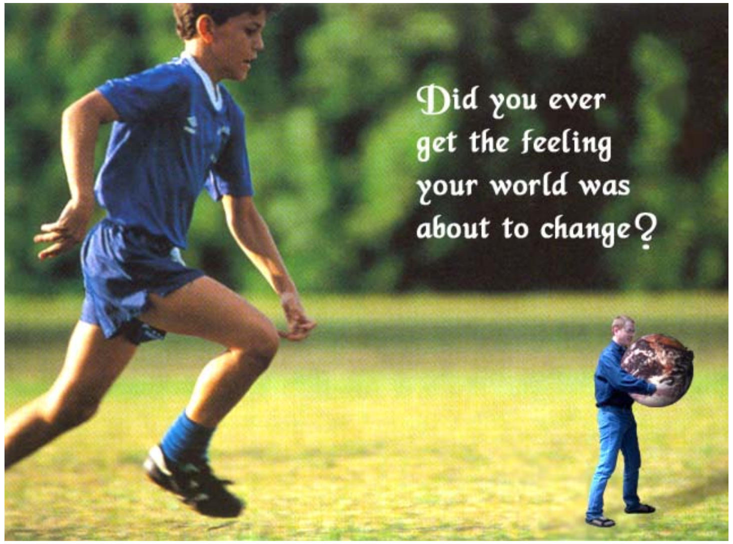
Includes: { a set of scalar variables, such as Portability}.

Note: probably not simple enough to define with a **single** Scale.

Type: *Complex* Quality Attribute.

Since,

- **if given sufficient resource, a system can be changed in**
  - **almost any way,**
- **the primary concern is with the amount of**
  - **resources**
    - **(such as time, people, tools and finance)**
- **needed to bring about specific changes**
  - **(the change 'cost').**


Did you ever get the feeling your world was about to change?

The Adaptive Cycle

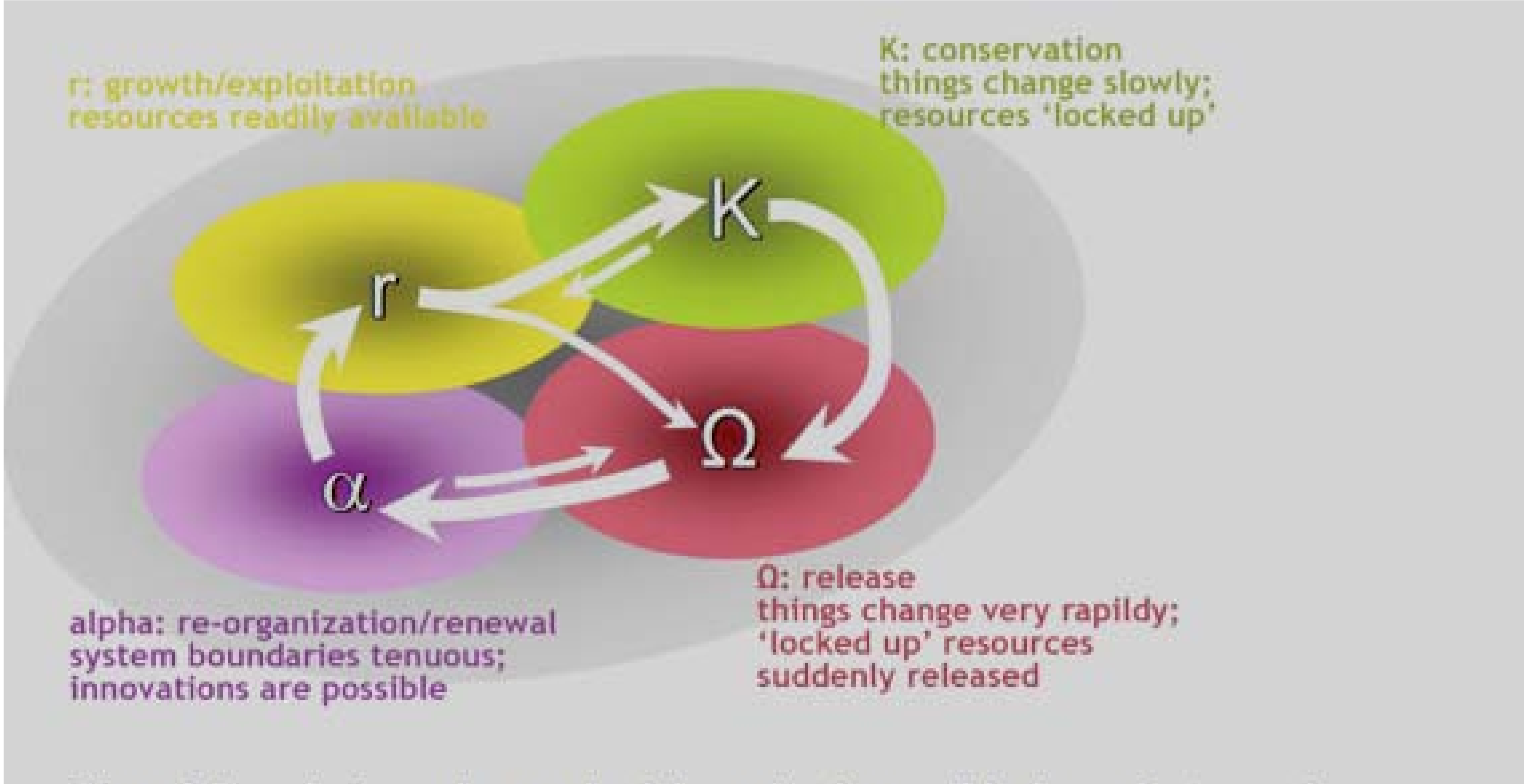


Figure 3. The adaptive cycle, as a simple loop, showing possible changes between phases.

http://www.resalliance.org/564.php

Adaptability:
Viewed as
Elementary or ***Complex*** concept..

Adaptability:

Type: **Elementary** Quality Requirement.

Scale: Time needed to adapt a defined [System] from a defined [**Initial State**] to another defined [**Final State**] using defined [**Means**].

© Alistair Boddy-Evans 2002

Adaptability:

Type: **Complex** Quality Requirement.

Includes: *{Flexibility, Upgradeability}.*

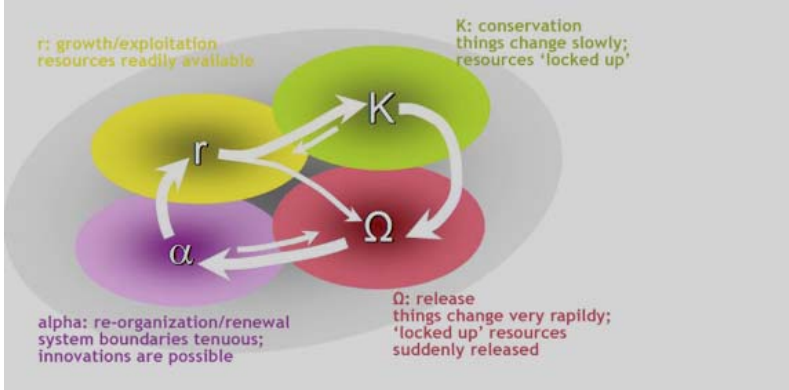


Figure 3. The adaptive cycle, as a simple loop, showing possible changes between phases.

“No system can be understood or managed by focusing on it at a *single* scale.”

Multiple scales and cross-scale effects - "Panarchy"No system can be understood or managed by focusing on it at a single scale.

- **All systems (and SESs especially) exist and function at multiple scales of space, time and social organization,**
  - **and the interactions across scales are fundamentally important in determining the dynamics of the system at any particular focal scale.**
  - **This interacting set of hierarchically structured scales has been termed a "panarchy" (Gunderson and Holling 2003).**

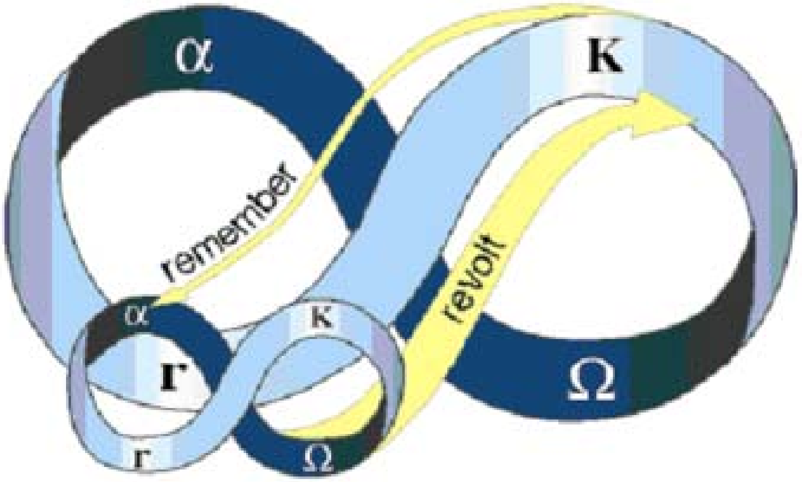


Figure 4. "Panarchy" - nested adaptive cycles, with influences between scales.

http://www.resalliance.org/564.php

Flexibility:

Gist: ‘Flexibility’ concerns the

‘in-built’ ability of the system

to adapt,

or to be adapted,

by its users,

to suit conditions

(without any fundamental system modification

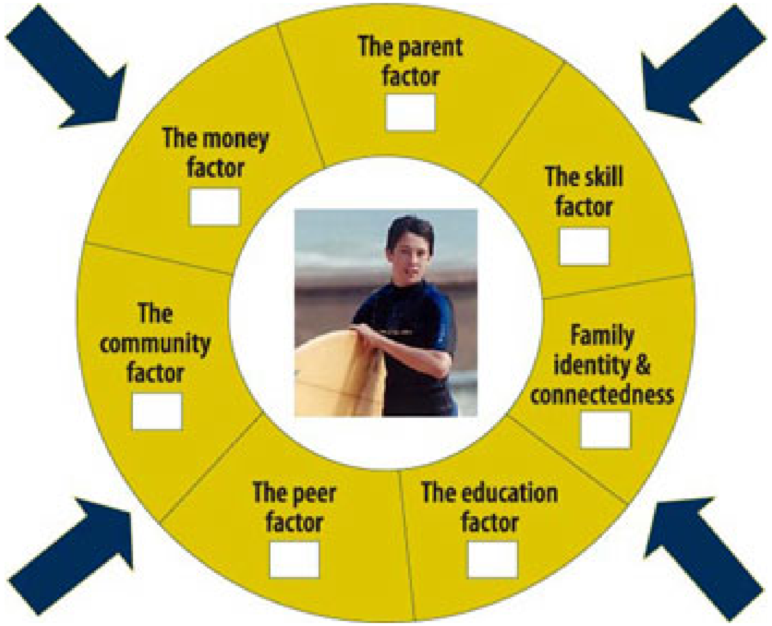
by system development).

Type: Complex Quality Requirement.

Includes: {Connectability, Tailorability}.

See next 2 slides!

Possible Synonyms: Resilience, Robustness



Connectability:

‘The cost to interconnect the system to its environment.’

Gist: The cost of connecting **one set of interfaces** to defined **environments** with **other interfaces**

Part Of: Flexibility.

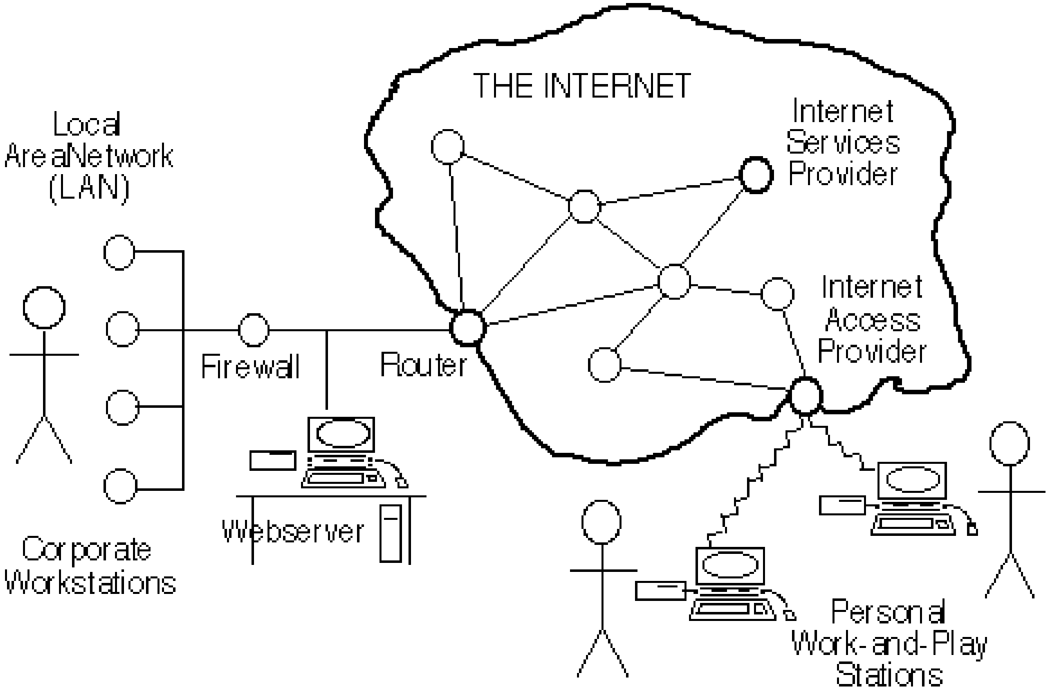
Scale: the **Effort** needed

to connect a defined [**Home Interface**]

to a defined [**Target Interface**]

using defined [**Methods**]

with minimum allowed system [**Degradation**].



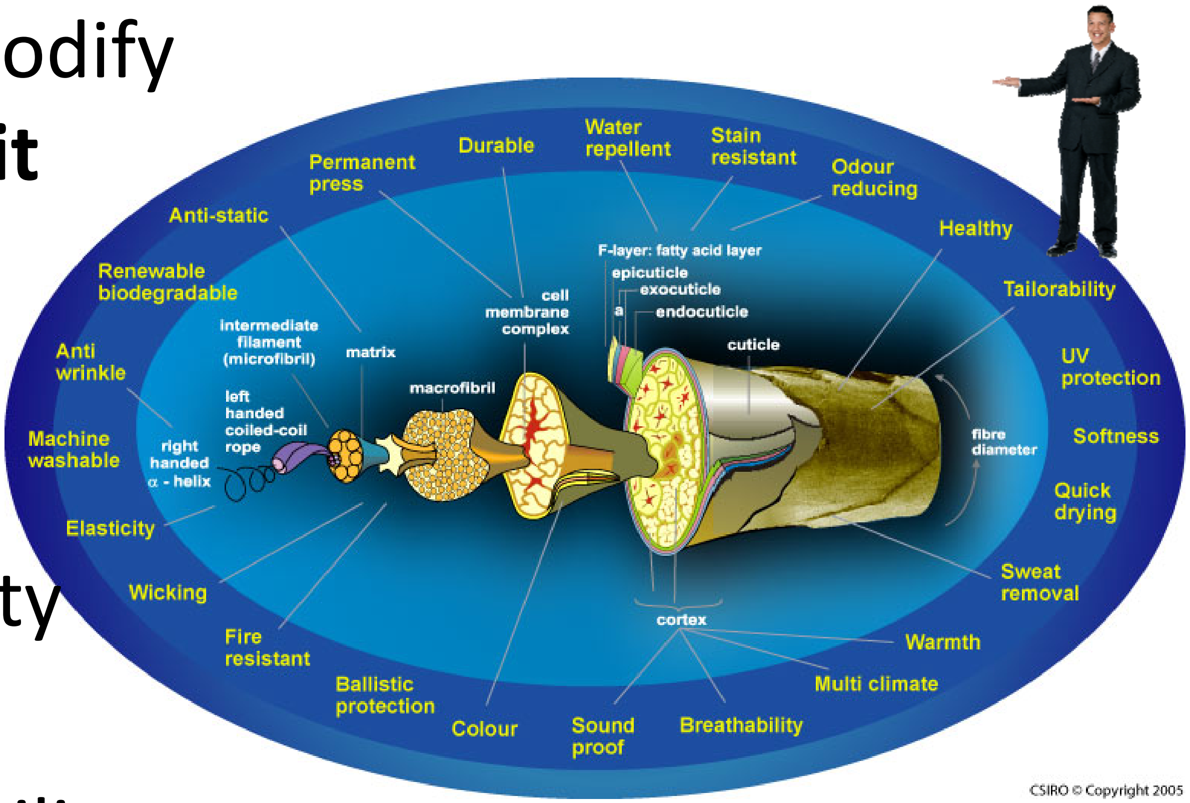
Tailorability:

Gist: The **cost** to modify the system to **suit** defined **future** conditions.

Part Of: Flexibility.

Type: *Complex* Quality Requirement.

Includes: {Extendibility, Interchangeability}.



CSIRO © Copyright 2005

Multiple Attributes of Wool Fiber !

Extendibility: Scalability

Extendibility:

Part Of: Tailorability.

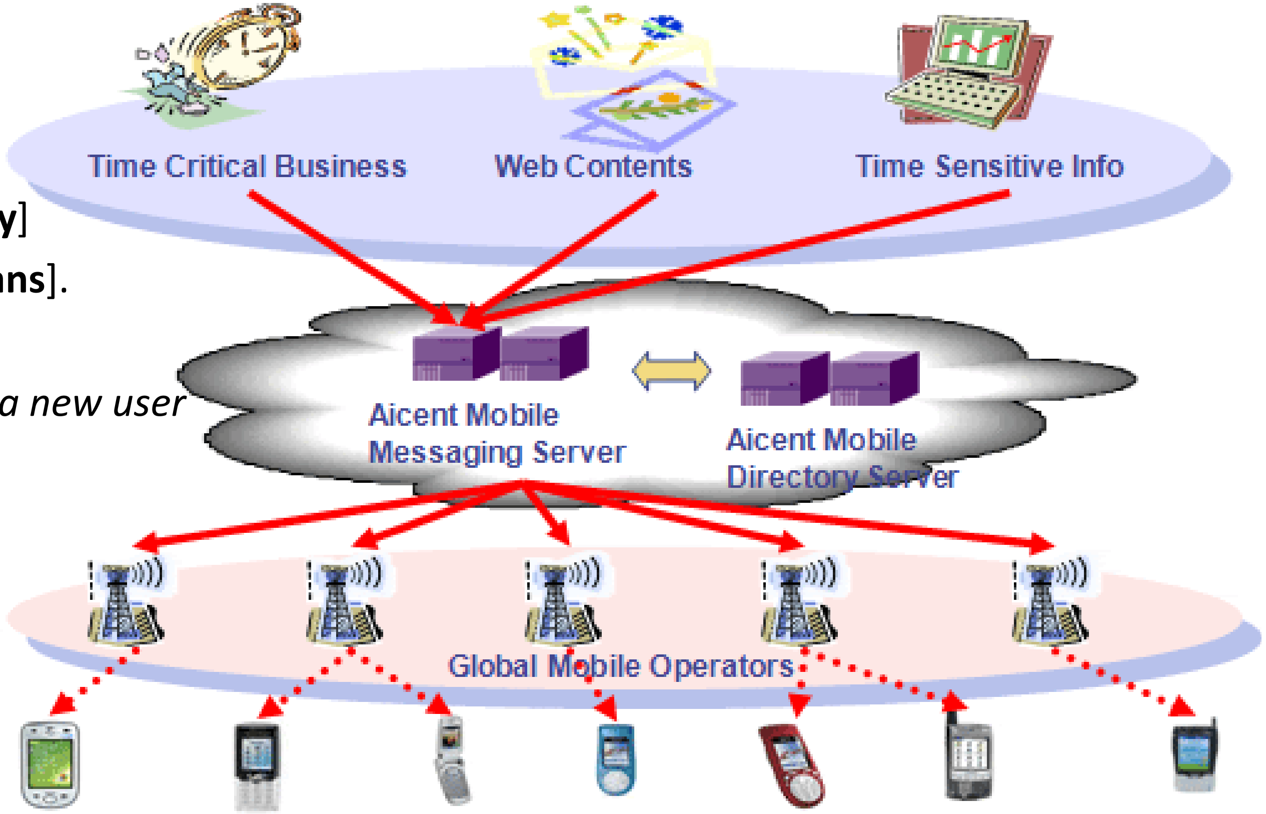
Synonym: Scalability.

Scale: The **cost** to add to
a defined [**System**]
a defined [**Extension Class**]
and defined [**Extension Quantity**]
using a defined [**Extension Means**].

"In other words, add such things as a new user or a new node."

Type: *Complex* Quality Attribute.

Includes: {Node Addability, Connection Addability, Application Addability, Subscriber Addability}.



Interchangeability:

‘The cost to modify use of system components.’

Interchangeability

Gist: This is concerned with the ability to modify the system, to switch from using a certain set of system components, to using another set.

Part Of: Tailorability.

Type: Elementary Quality Attribute.

“For example, this could be a daily occurrence switching system mode from day to night use.”

Scale: the Effort needed to
Successfully,
without Intolerable Side Effects,
replace a defined [Initial Set] of components,
with a defined [Replacement Set] of system components,
using defined [Means].

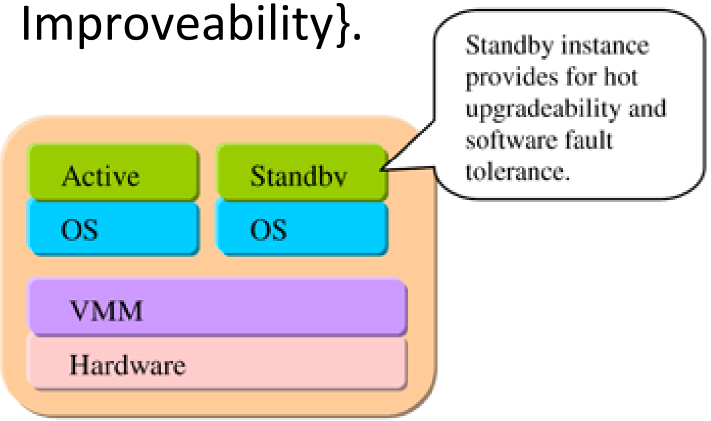
Upgradeability: ‘The cost to modify the system fundamentally; either to install it, or to change out system components.’

<u>Upgradeability</u>:

Gist: This concerns the ability of the system to be modified by the system developers or system support in planned stages (as opposed to unplanned maintenance or tailoring the system).

Type: *Complex* Quality Requirement.

Includes: {Installability, Portability, Improveability}.



Standby instance provides for hot upgradeability and software fault tolerance.

<u>Installability</u>: ‘The cost to install in defined conditions.’

Pattern: This concerns installing the system code and also, installing it in new locations to extend the system coverage. Could include conditions such as the installation being carried out by a customer or, by an IT professional on-site.

<u>Portability</u>: ‘The cost to move from location to location.’

Scale: The cost to transport a defined [System] from a defined [Initial Environment] to a defined [Target Environment] using defined [Means].

Type: Complex Quality Requirement.

Includes: {Data Portability,
Logic Portability,
Command Portability,
Media Portability}.

<u>Improveability</u>: ‘The cost to enhance the system.’

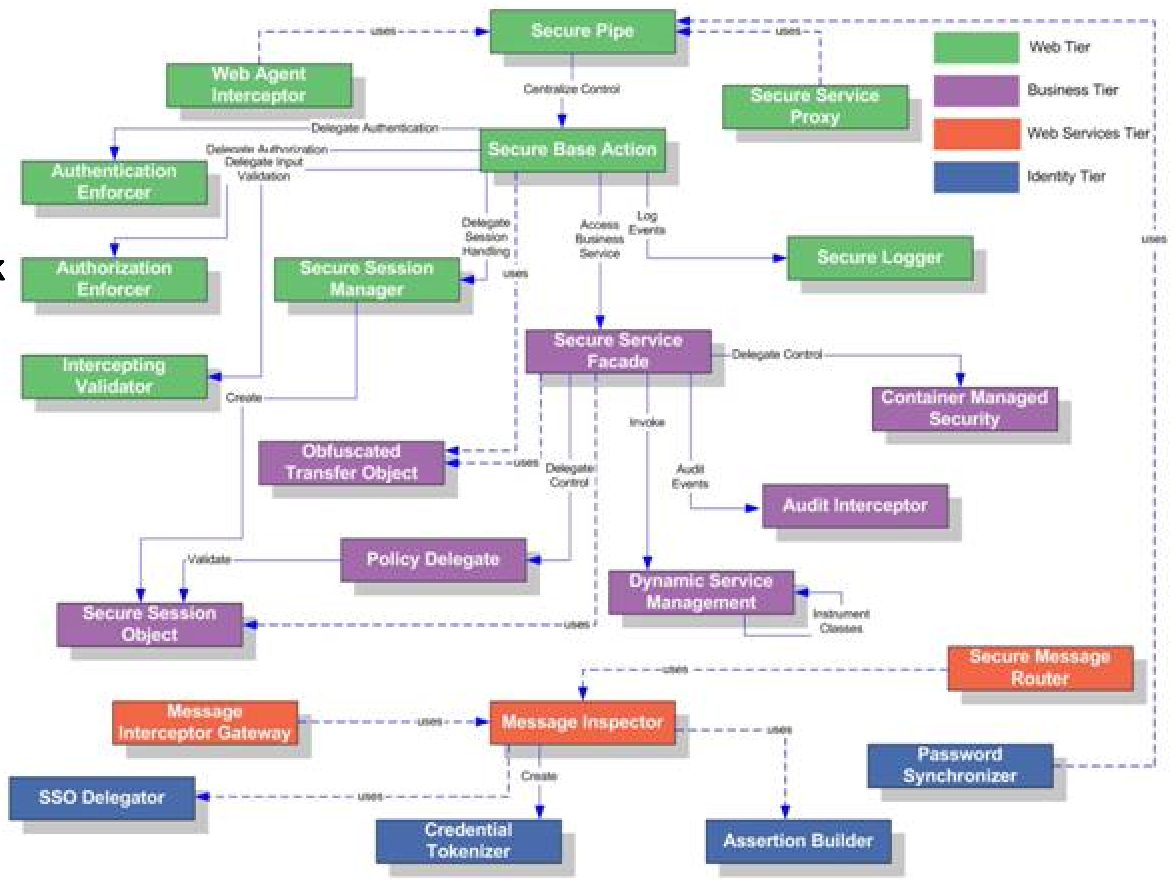
Gist: The ability to replace system components with others, which possesses improved (function, performance, cost and/or design) attributes.

Scale: The cost to add to a defined [System] a defined [Improvement] using a defined [Means].

This Basic 'Adaptability' Pattern Was Successfully Applied

- **Hopefully this set of patterns**
  - **gives you a departure point**
  - **for defining those maintenance attributes**
  - **you might want to control, quantitatively.**

- **The above adaptability definition**
  - **was use to co-ordinate the work**
    - **of 5,000 software engineers,**
    - **and 5,000 hardware engineers,**
    - **in UK,**
    - **in bringing out a new product line at a computer manufacturer.**
    - **Where 'Adaptability' was the Number One Product Characteristic**
  - **The Company became profitable for the next 14 years..**



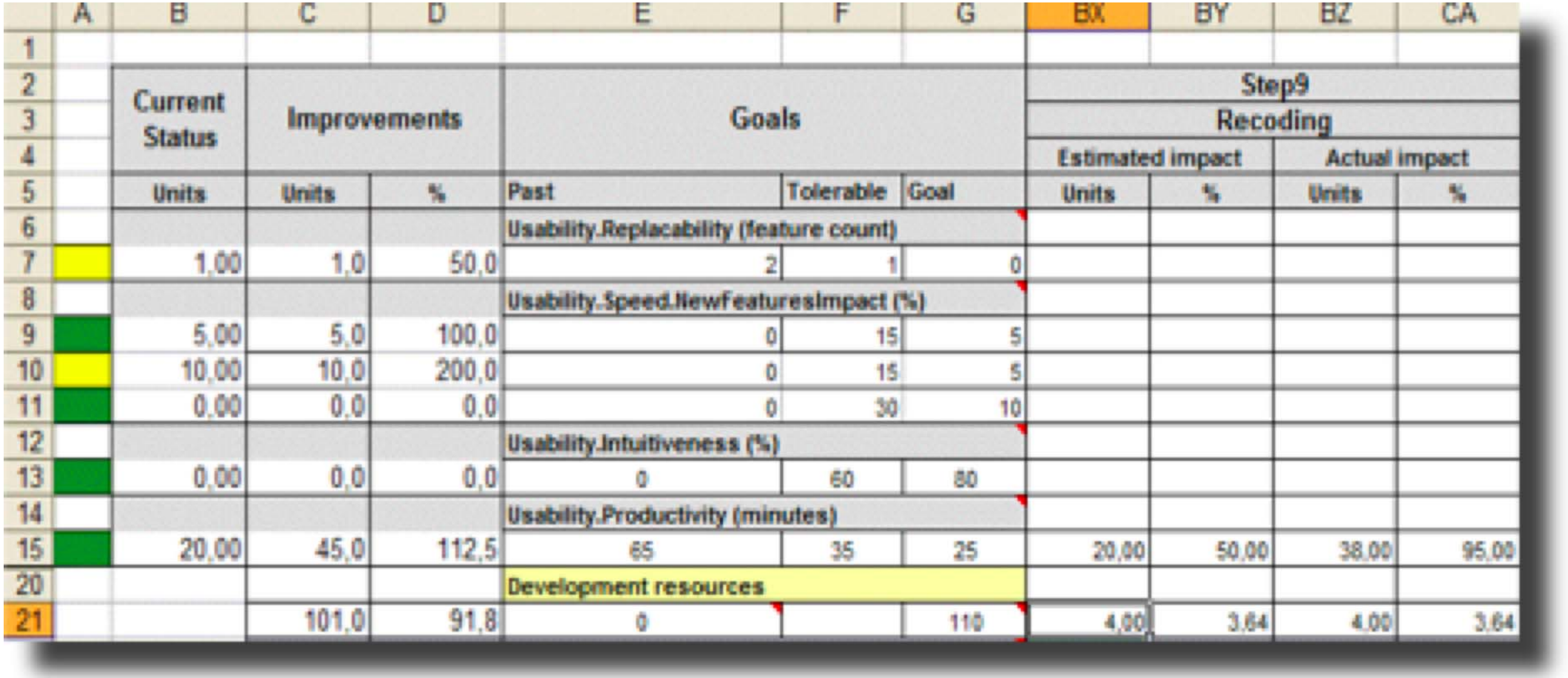
Security Patterns

The Software Architect Role in Maintainability

The role of the software architect is:

- to participate in **clarification of the requirements** that will be used as inputs to their architecture process.
- to insist that the requirements are **testably clear**: that means with defined and agreed scales of measure, and defined required levels of performance.
- to then **discover appropriate architecture**,
  - capable of delivering those levels of performance, hopefully within resource constraints, and
- **estimate** the probable **impact** of the architecture,
  - on the requirements (Impact Estimation)
- **define** the architecture in such **detail**
  - that the intent **cannot be misunderstood** by implementers,
  - and the desired **effects** are bound to be **delivered**.
- **monitor** the developing system as the architecture is applied in practice,
- and **make** necessary **adjustments**.
- finally **monitor** the **performance characteristics** throughout the lifetime of the system,
  - and make necessary **adjustments** to <u>requirements</u>
  - and to <u>architecture</u>,
  - in order to **maintain** needed system **performance** characteristics.

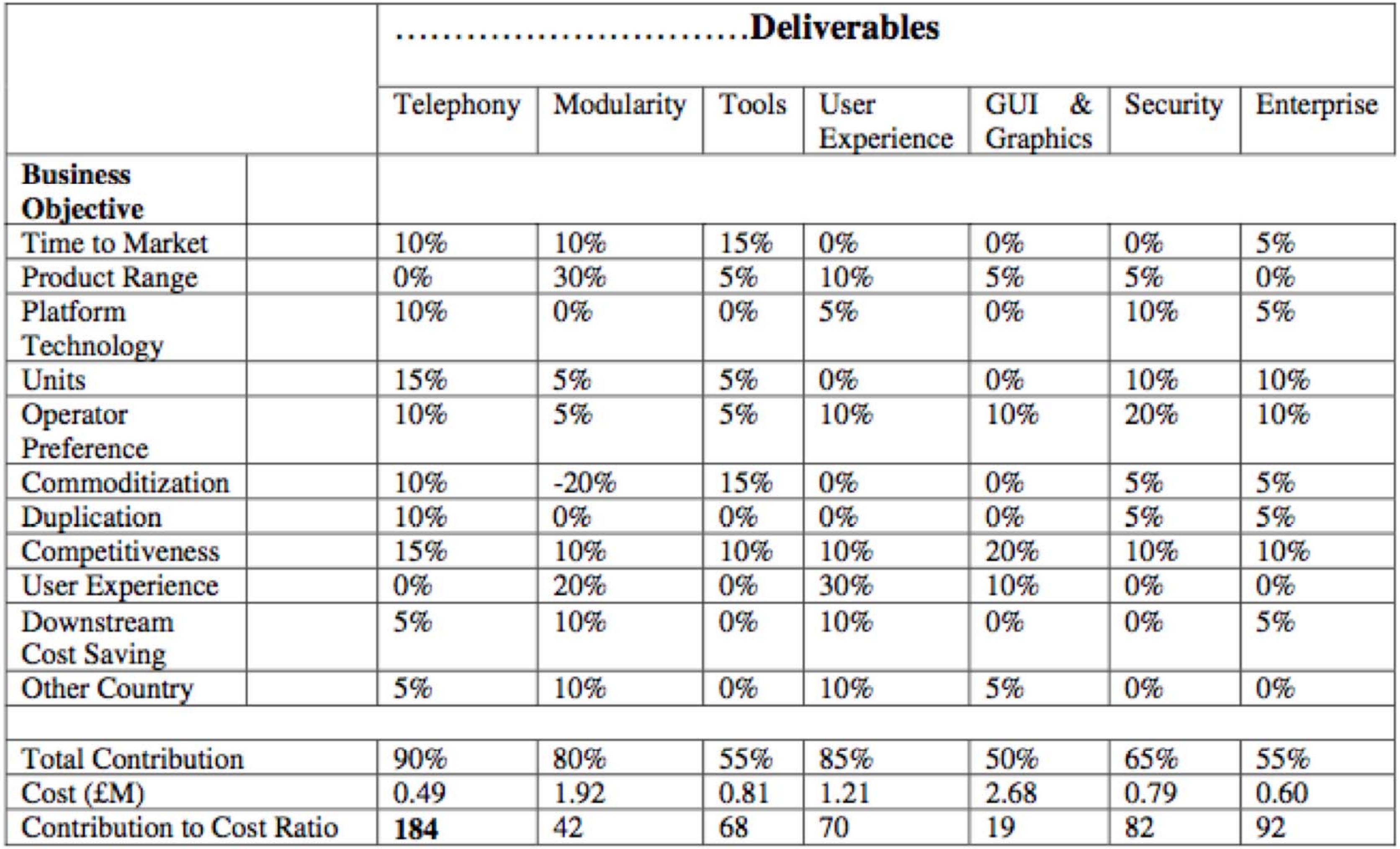
Evaluating Maintainability Designs Using Impact Estimation



	A	B	C	D	E	F	G	BX	BY	BZ	CA
1											
2		Current Status	Improvements		Goals			Step9			
3								Recoding			
4								Estimated impact		Actual impact	
5		Units	Units	%	Past	Tolerable	Goal	Units	%	Units	%
6					Usability.Replacability (feature count)						
7	🟨	1,00	1,0	50,0	2	1	0				
8					Usability.Speed.NewFeaturesImpact (%)						
9	🟩	5,00	5,0	100,0	0	15	5				
10	🟨	10,00	10,0	200,0	0	15	5				
11	🟩	0,00	0,0	0,0	0	30	10				
12					Usability.Intuitiveness (%)						
13	🟩	0,00	0,0	0,0	0	60	80				
14					Usability.Productivity (minutes)						
15	🟩	20,00	45,0	112,5	65	35	25	20,00	50,00	38,00	95,00
20					Development resources						
21			101,0	91,8	0		110	4,00	3,64	4,00	3,64

- *See Powerpoint Notes for detailed written comment.*
-

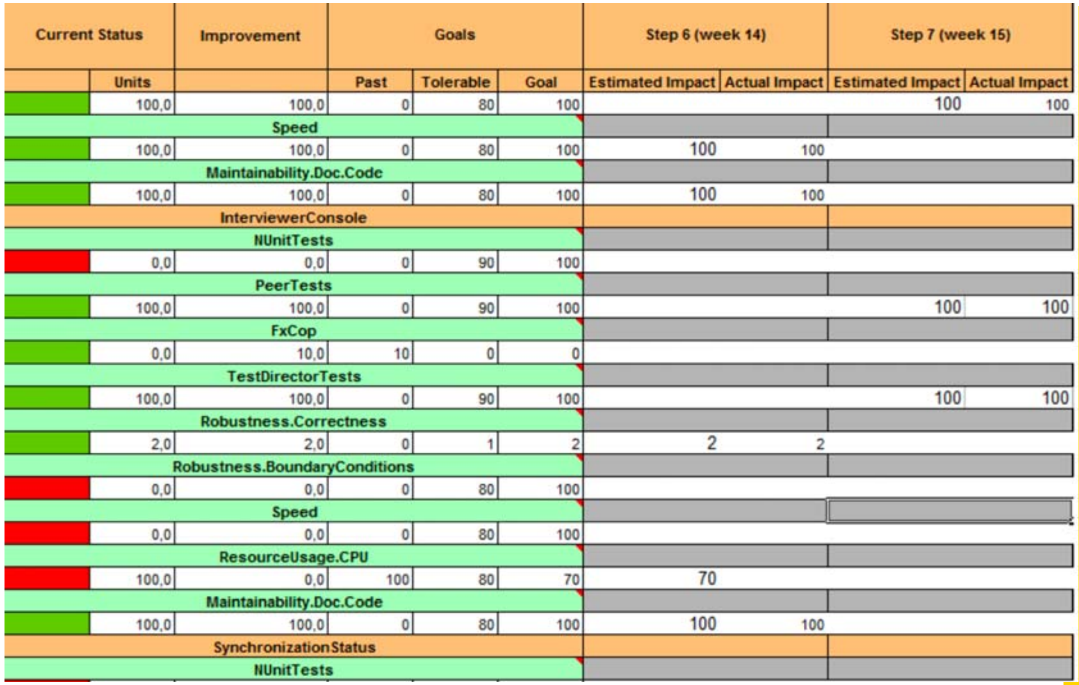
Architecture Level Impact Estimation Table



	..............................**Deliverables**						
	Telephony	Modularity	Tools	User Experience	GUI & Graphics	Security	Enterprise
Business Objective							
Time to Market	10%	10%	15%	0%	0%	0%	5%
Product Range	0%	30%	5%	10%	5%	5%	0%
Platform Technology	10%	0%	0%	5%	0%	10%	5%
Units	15%	5%	5%	0%	0%	10%	10%
Operator Preference	10%	5%	5%	10%	10%	20%	10%
Commoditization	10%	-20%	15%	0%	0%	5%	5%
Duplication	10%	0%	0%	0%	0%	5%	5%
Competitiveness	15%	10%	10%	10%	20%	10%	10%
User Experience	0%	20%	0%	30%	10%	0%	0%
Downstream Cost Saving	5%	10%	0%	10%	0%	0%	5%
Other Country	5%	10%	0%	10%	5%	0%	0%
Total Contribution	90%	80%	55%	85%	50%	65%	55%
Cost (£M)	0.49	1.92	0.81	1.21	2.68	0.79	0.60
Contribution to Cost Ratio	**184**	42	68	70	19	82	92

- See PPT Notes

Engineering "Maintainability": Green Week Weekly 'Refactoring' at Confirmit



Current Status		Improvement	Goals			Step 6 (week 14)		Step 7 (week 15)	
	Units		Past	Tolerable	Goal	Estimated Impact	Actual Impact	Estimated Impact	Actual Impact
🟩	100,0	100,0	0	80	100			100	100
Speed									
🟩	100,0	100,0	0	80	100	100	100		
Maintainability.Doc.Code									
🟩	100,0	100,0	0	80	100	100	100		
InterviewerConsole									
NUnitTests									
🟥	0,0	0,0	0	90	100				
PeerTests									
🟩	100,0	100,0	0	90	100			100	100
FxCop									
🟩	0,0	10,0	10	0	0				
TestDirectorTests									
🟩	100,0	100,0	0	90	100			100	100
Robustness.Correctness									
🟩	2,0	2,0	0	1	2	2	2		
Robustness.BoundaryConditions									
🟥	0,0	0,0	0	80	100				
Speed									
🟥	0,0	0,0	0	80	100				
ResourceUsage.CPU									
🟥	100,0	0,0	100	80	70	70			
Maintainability.Doc.Code									
🟩	100,0	100,0	0	80	100	100	100		
SynchronizationStatus									
NUnitTests									

Speed

Maintainability

Nunit Tests

PeerTests

TestDirectorTests

Robustness.Correctness

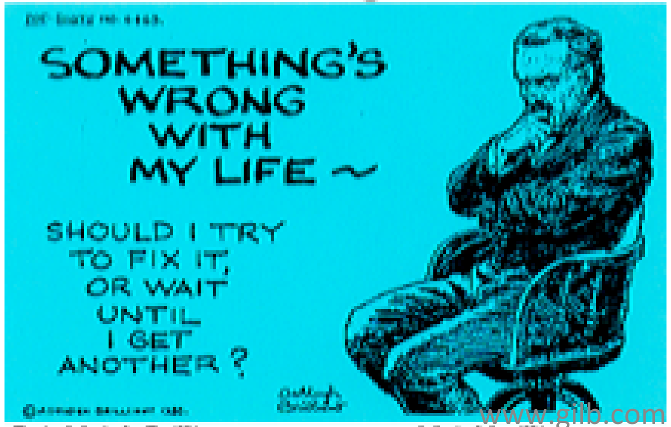
Robustness.Boundary Conditions

ResourceUsage.CPU

Maintainability.DocCode

SynchronizationStatus

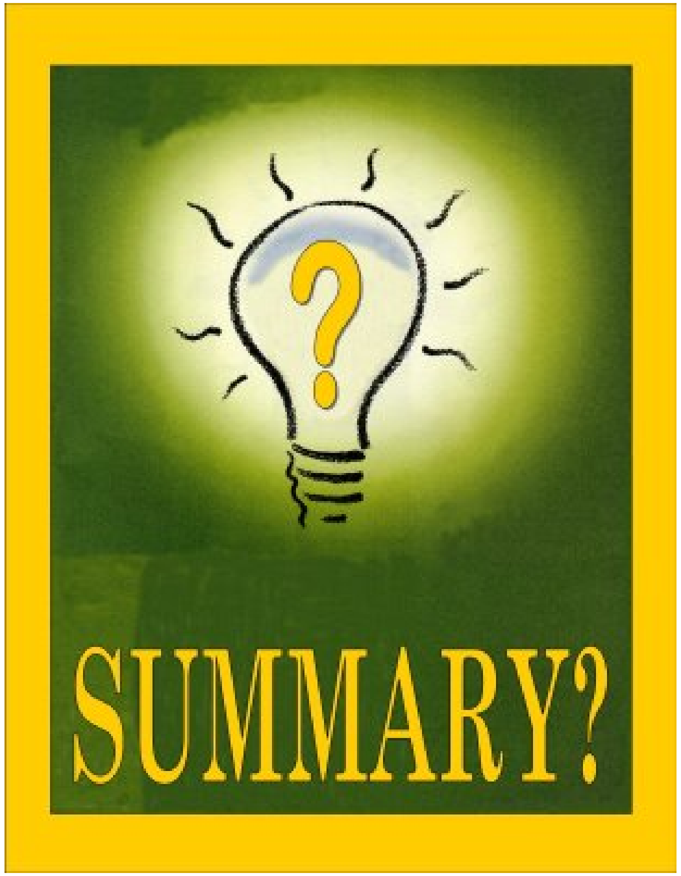
POT-SHOTS — Brilliant Thoughts in 17 words or less


SOMETHING'S WRONG WITH MY LIFE ~
SHOULD I TRY TO FIX IT, OR WAIT UNTIL I GET ANOTHER ?

© Ashleigh Brilliant www.ashleighbrilliant.com

Lecture Summary

- The **many** types of maintainability – ease of change – characteristics needed in large scale or critical software,
  - can be **architected**
  - and **engineered** using **numeric** measurement
  - and sound engineering **principles**,
  - instead of conventional small scale programming culture intuition.
- **Real** systems engineers will move towards this mode of 'real' software engineering.
- We cannot continue to have the craft of programming culture, dominate our systems engineering practices –
  - because software has become too critical a component of every major system.
  - The real engineers have to take **control**.
  - The **programmers will not wake up** without encouragement from real engineers.



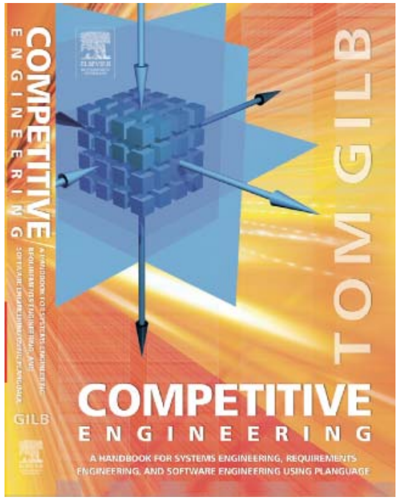
References

References

Gilb, Tom, Competitive Engineering, 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

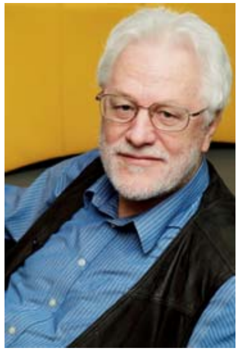

TOM GILB
COMPETITIVE ENGINEERING
A HANDBOOK FOR SYSTEMS ENGINEERING, REQUIREMENTS ENGINEERING, AND SOFTWARE ENGINEERING USING PLANGUAGE

INCOSE Systems Engineering Handbook v. 3
INCOSE-TP-2003-002-03, June 2006 , www.INCOSE.org

[Dart 93] Susan Dart , Alan M. Christie , Alan W Brown
A Case Study in Software Maintenance, Technical Report CMU/SEI-93-TR-8 ,
ESC-TR-93-185 , June 1993

Chris Inacio: Software Fault Tolerance**,** Carnegie Mellon University
18-849b Dependable Embedded Systems, Spring 1998
http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/
Google N-Version Software for more information on distinct software and N-version software.

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, and many smaller and lesser known others. See www.Gilb.com . He can be reached at: Planguage@mac.com

References

Gilb, Tom, Competitive Engineering, 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.

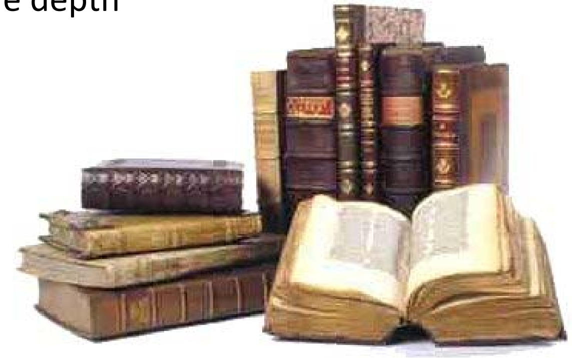
Chapter 5: Scales of Measure:

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



INCOSE Systems Engineering Handbook v. 3

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

[Dart 93] Susan Dart , Alan M. Christie , Alan W Brown

A Case Study in Software Maintenance, Technical Report CMU/SEI-93-TR-8 ,

ESC-TR-93-185 , June 1993

Chris Inacio: Software Fault Tolerance**,** Carnegie Mellon University

18-849b Dependable Embedded Systems, Spring 1998

http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/

Google N-Version Software for more information on distinct software and N-version software.

Last Slide

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, and many smaller and lesser known others. See www.Gilb.com .
-
-
-
-
-