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

**Tom Gilb**

Result Planning Limited
Tom.Gilb@INCOSE.org

For ACCU Oxford UK
Friday 4\(^{th}\) April 2008
1400 90 Minutes
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.
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.
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}.

April 14, 2008

www.gilb.com
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?

[Cazes-Valettes, 2001.](#)

[http://www.youtube.com/watch?v=X-JiKA1vTRo](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**}. 

April 14, 2008  www.gilb.com
Software Downtime:

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.

**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.
Testability:

Type: Software Quality Requirement.
Part of: Rock Solid Robustness
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.
5. All projects in project portfolio system.
6. Reduce development project spend on low priority work with better alignment between Technology and business demand.
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.

• 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

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.

April 14, 2008  www.gilb.com
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.

- 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.
Let us take a simplified tour of the method.

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.
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.
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 \textit{dramatic} architecture,
  - in an effort to meet the \textit{radically} improved requirement level.
- The reader need not take any design \textit{too} seriously.
- This is an example of trying to solve the problem, using engineering techniques (redundancy)
  - that have a solid scientific history.
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 **Analyst Selection**, 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.
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.
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.
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?

- **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.

April 14, 2008  www.gilb.com
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
## Design Impact Estimation Table:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Requirements</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Availability</td>
<td>50%</td>
<td>10%</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
<td>60%</td>
<td>185%</td>
</tr>
<tr>
<td>90% &lt;-&gt; 99.5% Up time</td>
<td>50%</td>
<td>5%</td>
<td>5-10%</td>
<td>0%</td>
<td>0%</td>
<td>200%</td>
<td>265%</td>
</tr>
<tr>
<td>Usability</td>
<td>50%</td>
<td>5-10%</td>
<td>5-10%</td>
<td>50%</td>
<td>0%</td>
<td>10%</td>
<td>130%</td>
</tr>
<tr>
<td>200 &lt;-&gt; 60 Requests by Users</td>
<td>50%</td>
<td>10%</td>
<td>90%</td>
<td>25%</td>
<td>5%</td>
<td>50%</td>
<td>180%</td>
</tr>
<tr>
<td>Responsiveness</td>
<td>50%</td>
<td>10%</td>
<td>90%</td>
<td>25%</td>
<td>5%</td>
<td>50%</td>
<td>180%</td>
</tr>
<tr>
<td>70% &lt;-&gt; ECP's on time</td>
<td>50%</td>
<td>10%</td>
<td>90%</td>
<td>25%</td>
<td>5%</td>
<td>50%</td>
<td>180%</td>
</tr>
<tr>
<td>Productivity</td>
<td>45%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>303%</td>
<td>303%</td>
</tr>
<tr>
<td>3:1 Return on Investment</td>
<td>45%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>303%</td>
<td>303%</td>
</tr>
<tr>
<td>Morale</td>
<td>50%</td>
<td>15%</td>
<td>15%</td>
<td>15%</td>
<td>15%</td>
<td>61%</td>
<td>251%</td>
</tr>
<tr>
<td>72 &lt;-&gt; 60 per month on Sick Leave</td>
<td>50%</td>
<td>15%</td>
<td>15%</td>
<td>15%</td>
<td>15%</td>
<td>61%</td>
<td>251%</td>
</tr>
<tr>
<td>Data Integrity</td>
<td>42%</td>
<td>10%</td>
<td>25%</td>
<td>5%</td>
<td>70%</td>
<td>25%</td>
<td>177%</td>
</tr>
<tr>
<td>88% &lt;-&gt; 97% Data Error %</td>
<td>42%</td>
<td>10%</td>
<td>25%</td>
<td>5%</td>
<td>70%</td>
<td>25%</td>
<td>177%</td>
</tr>
<tr>
<td>Technology Adaptability</td>
<td>5%</td>
<td>30%</td>
<td>5%</td>
<td>60%</td>
<td>0%</td>
<td>60%</td>
<td>160%</td>
</tr>
<tr>
<td>75% Adapt Technology</td>
<td>5%</td>
<td>30%</td>
<td>5%</td>
<td>60%</td>
<td>0%</td>
<td>60%</td>
<td>160%</td>
</tr>
<tr>
<td>Requirement Adaptability</td>
<td>80%</td>
<td>20%</td>
<td>60%</td>
<td>75%</td>
<td>20%</td>
<td>5%</td>
<td>260%</td>
</tr>
<tr>
<td>? &lt;-&gt; 2.6% Adapt to Change</td>
<td>80%</td>
<td>20%</td>
<td>60%</td>
<td>75%</td>
<td>20%</td>
<td>5%</td>
<td>260%</td>
</tr>
<tr>
<td>Resource Adaptability</td>
<td>10%</td>
<td>80%</td>
<td>5%</td>
<td>50%</td>
<td>50%</td>
<td>75%</td>
<td>270%</td>
</tr>
<tr>
<td>2.1M &lt;-&gt; ? Resource Change</td>
<td>10%</td>
<td>80%</td>
<td>5%</td>
<td>50%</td>
<td>50%</td>
<td>75%</td>
<td>270%</td>
</tr>
<tr>
<td>Cost Reduction</td>
<td>50%</td>
<td>40%</td>
<td>10%</td>
<td>40%</td>
<td>50%</td>
<td>50%</td>
<td>240%</td>
</tr>
<tr>
<td>FADS &lt;-&gt; 30% Total Funding</td>
<td>50%</td>
<td>40%</td>
<td>10%</td>
<td>40%</td>
<td>50%</td>
<td>50%</td>
<td>240%</td>
</tr>
<tr>
<td><strong>Sum of Performance</strong></td>
<td>482%</td>
<td>280%</td>
<td>305%</td>
<td>390%</td>
<td>315%</td>
<td>649%</td>
<td></td>
</tr>
<tr>
<td>Money % of total budget</td>
<td>15%</td>
<td>4%</td>
<td>3%</td>
<td>4%</td>
<td>6%</td>
<td>4%</td>
<td>36%</td>
</tr>
<tr>
<td>Time % total work months/year</td>
<td>15%</td>
<td>15%</td>
<td>20%</td>
<td>10%</td>
<td>20%</td>
<td>18%</td>
<td>98%</td>
</tr>
<tr>
<td><strong>Sum of Costs</strong></td>
<td>30</td>
<td>19</td>
<td>23</td>
<td>14</td>
<td>26</td>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>
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.
General ‘Change Attribute’ Tailoring

- The following slides 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:

**Code Portability:**

**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.

April 14, 2008

www.gilb.com
A Generic Set of Performance measures – including several related to ‘change’

Figure 5.3
One decomposition possibility for performance attributes with emphasis on the detail of the quality attributes.
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.
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’).
The Adaptive Cycle

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

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

Adaptability:
Type: Complex Quality Requirement.
Includes: {Flexibility, Upgradeability}.
“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}.

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}. 

April 14, 2008 59 www.gilb.com
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.’

**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}.

**Installability:** ‘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.

**Portability:** ‘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}.

**Improveability:** ‘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..
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 **requirements**
  – and to **architecture**,
  – in order to **maintain** needed system **performance** characteristics.
Evaluating Maintainability Designs Using Impact Estimation

- See Powerpoint Notes for detailed written comment.
# Architecture Level Impact Estimation Table

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

- See PPT Notes

April 14, 2008  www.gilb.com  65
Engineering “Maintainability”: Green Week
Weekly ‘Refactoring’ at Confirmit

<table>
<thead>
<tr>
<th>Current Status</th>
<th>Improvement</th>
<th>Goals</th>
<th>Step 6 (week 14)</th>
<th>Step 7 (week 15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Units</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>0</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>Speed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>0</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>Maintainability.Docs.Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>0</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>Interviewer.Console</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>0</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>MiniTests</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>96</td>
<td>100</td>
</tr>
<tr>
<td>PeerTests</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>0</td>
<td>96</td>
<td>100</td>
</tr>
<tr>
<td>FlexCop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TestDirectorTests</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>0</td>
<td>96</td>
<td>100</td>
</tr>
<tr>
<td>Robustness.Correctness</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td>2.0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Robustness.BoundaryConditions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>Speed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>ResourceUsage.CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>0</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>Maintainability.Docs.Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>0</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>Synchronization.Status</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.0</td>
<td>100.0</td>
<td>0</td>
<td>80</td>
<td>100</td>
</tr>
</tbody>
</table>

---

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

© Ashleigh Brilliant www.ashleighbrilliant.com

April 14, 2008
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


Chapter 5: Scales of Measure:
Chapter 10: Evolutionary Project Management:

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

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.

- 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

April 14, 2008  www.gilb.com
References


Chapter 5: Scales of Measure:

Chapter 10: Evolutionary Project Management:

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


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

• **BIOGRAPHY**
• 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](http://www.Gilb.com).