ASQ American Society for Quality (ASQ) DC & MD Metro Software Special Interest Group (SSIG)
McLean, VA // Gaithersburg, MD // Eatontown, NJ

The Open Source Quality Paradox: How Can Code Done Bad Be So Good?

Terry Bollinger
The MITRE Corporation

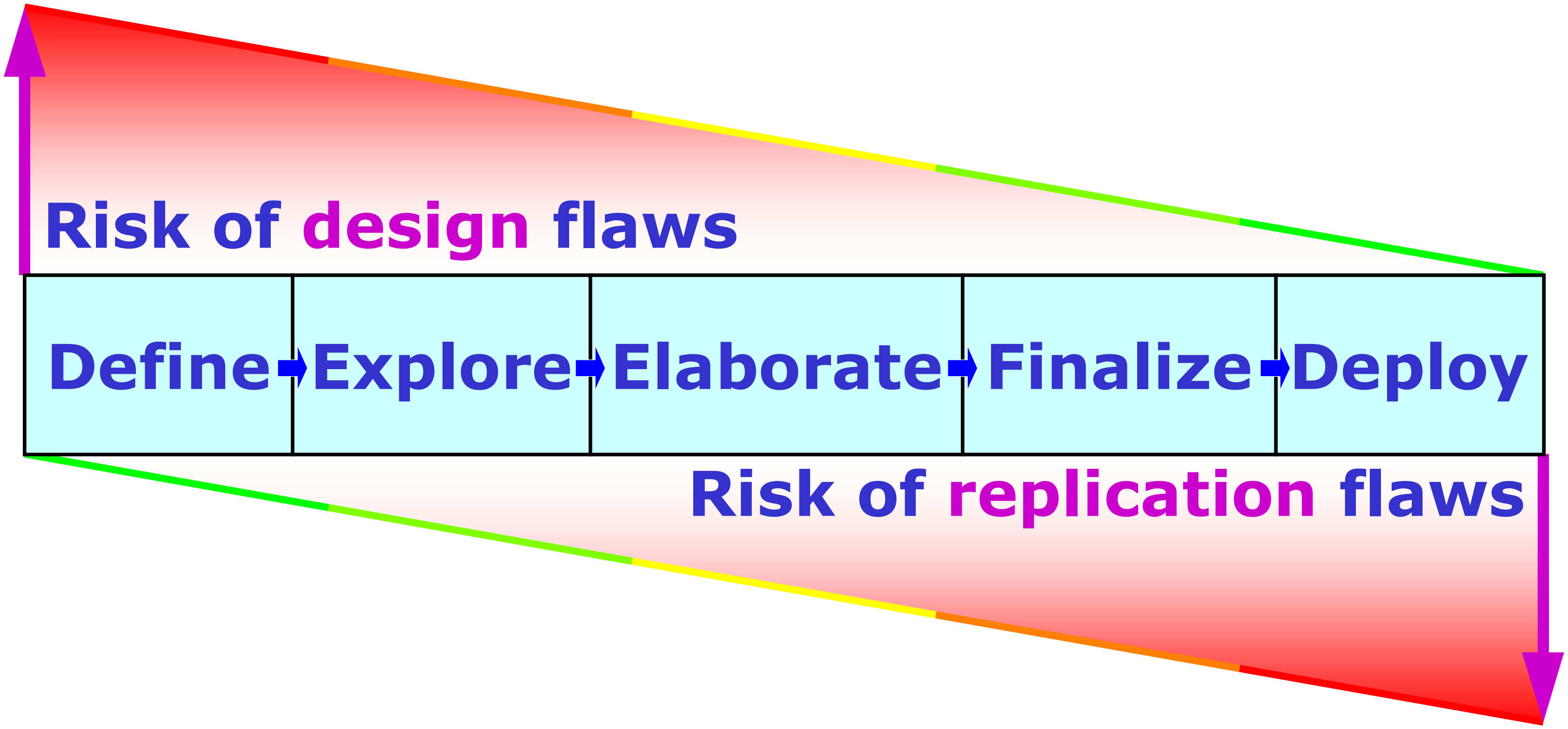
October 25, 2006

Note: *The concepts and materials in this presentation are the result of work performed by the author independently of The MITRE Corporation or its customers. The author's affiliation with The MITRE Corporation is listed here for identification purposes only, and does not imply MITRE concurrence with or support for any of the positions, opinions or viewpoints expressed in this presentation. It is released under a under a BLT 1.1 License.*

Background & Goals

- **The author**
  - Makes no claim to be an expert in anything
  - Tries hard to ask really annoying questions
  - Always encourages questions on quantum mechanics
- **The goals of this talk**
  - Take a few pokes at what is meant by “software quality”
  - Discuss if-why-how open source software has quality
  - Introduce a few ideas about component-based quality
- **The approach**
  - Examine a few critical definitions and assumptions
  - Try to understand how open source relates to the past
  - Look for a synthesis and a way forward

Two Types of Risk



Risk of design flaws

Define ➡ Explore ➡ Elaborate ➡ Finalize ➡ Deploy

Risk of replication flaws

Design Risks

- **Design risks = What you *don't* know**
- **Design risks include:**
  - Omissions (missing features, missing checks)
  - Wrong understandings (wrong needs, wrong theories)
  - Deleterious emergent behaviors (<u>big</u> problem!)
    - Emergent performance issues
    - Emergent interface issues
    - Emergent reliability issues
    - Emergent security issues
    - … (one for every ility) …
- **Metrics**
  - Metrics <u>must</u> encourage searches for emergent issues
  - Analytical creativity and insight are vital for success

Replication Risks

- **Replication risks = Not following rules**
- **Replication risks include:**
  - Poor training (not enough information to perform well)
  - Sloppy execution (indifference to what should be done)
  - Poor memory (people always forget)
  - Inaccuracy (people are not machines)
  - Using groups or teams amplifies the risks enormously:
    - Well-trained individuals *do not* translate into well-trained groups
    - The huge "communications gap" between people masks training
    - The communications gap is even more damaging to execution
    - Forgetfulness easily becomes rampant at the group level
- **Metrics**
  - Must enforce machine-like memory and rote repetition

Matching Risks to Players

- **The two types of risk:**
  - Design risks (needed: creativity, prediction, insight)
  - Replication risks (needed: memory, precision, diligence)
- **The two types of players:**
  - Humans (sloppy, slow, forgetful… and oh yes, creative)
  - Computers (precise, incredibly fast, perfect memories)
- **The quandary: Who should do what?**
  - How does this one sound?
    - Train people to replicate better. Grade their performance not on creativity, but on how well they can overcome human frailties of bad memories and sloppiness… all in the worst-case team setting.
  - Here's an alternative:
    - Use <u>computers</u> to <u>replicate</u>; keep <u>people</u> focused on <u>creativity</u>.

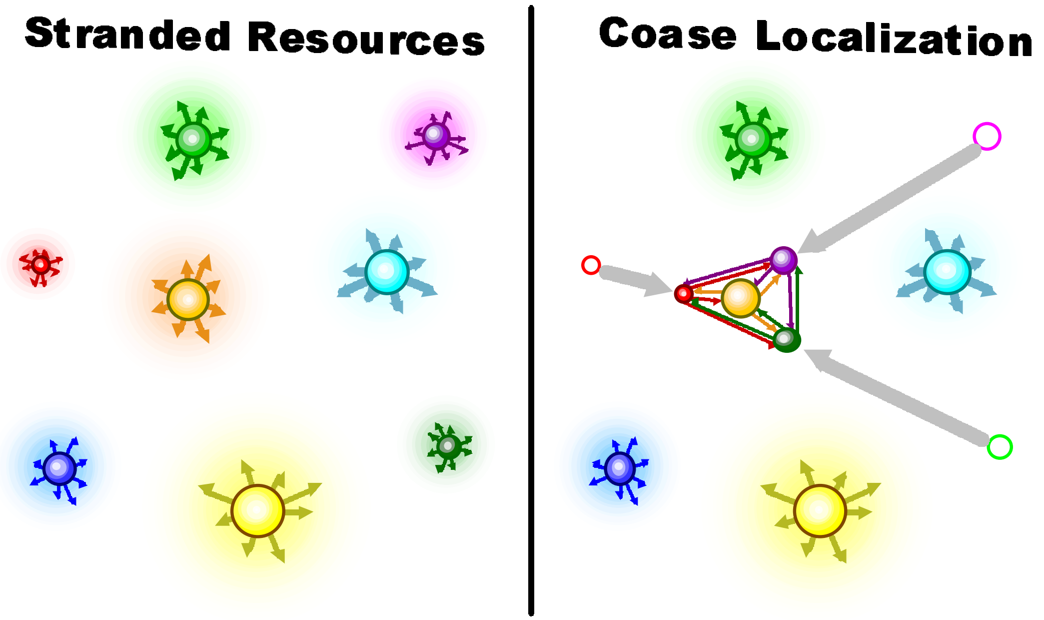
The Internet as a Global Replicator

- **Software as machinery:**
  - ☐ Like machinery, software performs useful functions
  - ☐ Like machinery, software must be designed and built
  - ☐ The difference: Software is machinery built out of pure information (no permanent physical resources needed)
- **For information machinery, transmission and remote storage *are* replication**
- **The Internet enables global replication:**
  - ☐ Millions of people can receive software tasks & products
  - ☐ The ease and richness of work product replication allows new modes of interaction between participants
  - ☐ *Self-selection* enables creativity in such interactions

Open Source Software and Creativity

1970-80s: Era of the Software Firm

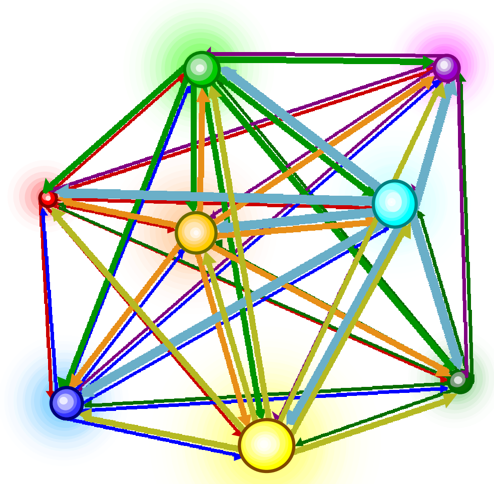
(costly data transport drives structure)



RESULT: Innovation is enabled, but "invisible hand" is limited

1990s-on: Free Market

(cheap transport dominates)

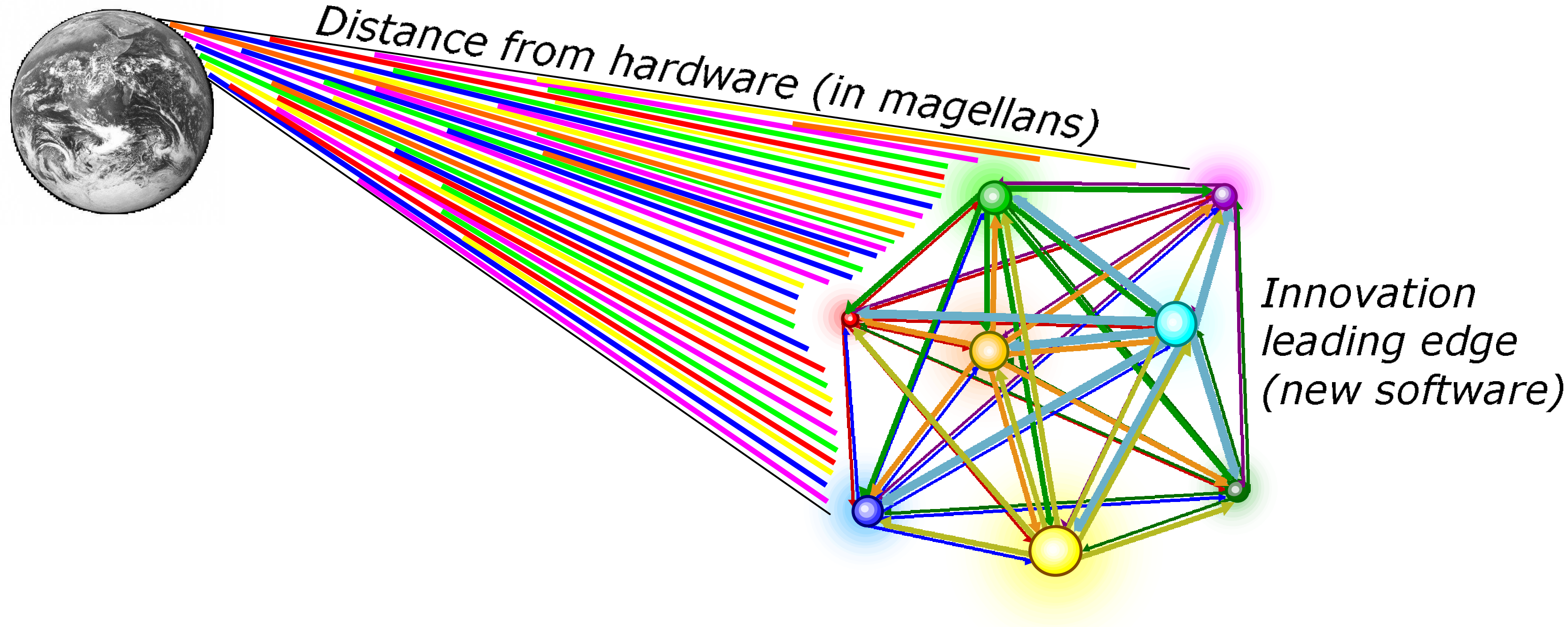


RESULT: "Invisible hand" is unleashed

Source: "Software Cooperatives" by Terry Bollinger (http://www.terrybollinger.com/)

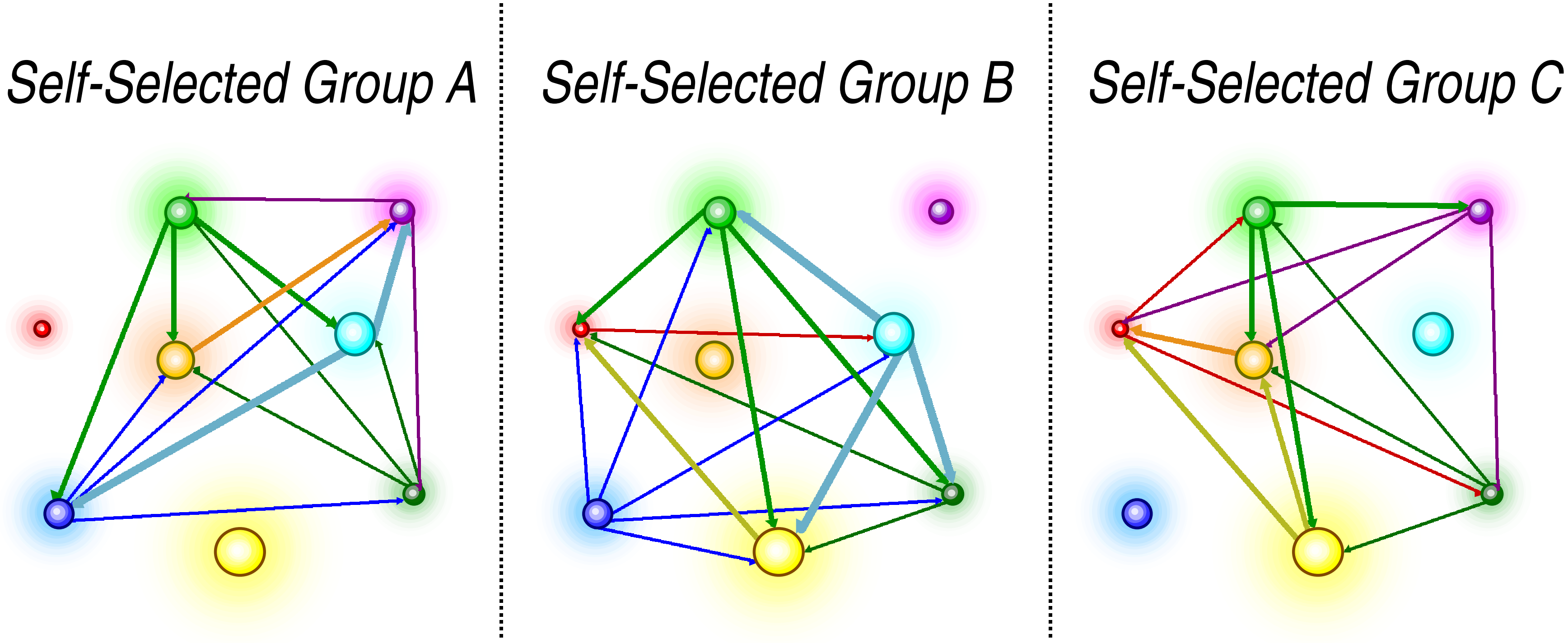
The Emergence of Open Source Software

- **Open source software exists because:**
  - ☐ A low-cost global internet enables synergistic sharing
  - ☐ The distance from hardware to apps has grown too large
  - ☐ Cost of rebuilding software infrastructure is too high
  - ☐ Economic attraction of composability rises over time



What are the Quality Consequences?

- **Self-selecting groups** **with high *internal cohesion* can emerge**
- **Group communications improve** **through shared knowledge**
- **Focused testing** **by a group raises reliability levels over time**



Self-Selected Group A

Self-Selected Group B

Self-Selected Group C

IMPLICATION: ***Self-selection of groups can directly benefit quality***

Nice Theory — Any Proof?

- **At least a quarter of federal web servers now run on open source technology**
  - Source: Keith Thurston (GSA) in “The real cost of open source,” FCW, Nov 22, 2004.

- **U.S. DoD views open source software as indistinguishable from proprietary**
  - Source: John P. Stenbit, DoD CIO, in a May 28, 2003 memo “Open Source Software (OSS) in the Department of Defense (DoD)”

- **The military uses open source software in high-value & security-critical situations**
  - Source: “Use of Free and Open Source Software (**FOSS**) in the U.S. Department of Defense,” Jan 2, 2003.

Summary of DoD Survey Results (2003)

- 115 FOSS applications identified
- 251 typical examples of FOSS use found
  - Highly diverse & “tip of the iceberg”
  - *Not* just Linux; many different applications & uses
- Some surprises:
  - Many DoD intranets depend heavily on FOSS
  - Software development makes heavy use of FOSS
  - FOSS is used extensively in security applications (!)
  - Research uses FOSS to exchange ideas & cut costs
  - Cost is seldom the only reason for using FOSS
  - FOSS and proprietary can be used together

How Does the U.S. DoD Use FOSS?

- **Infrastructure Support**
  - Network support, especially Internet technologies
- **Software Development**
  - C, C++, Ada, Perl, Python, debugging, code control
- **Security**
  - Operating systems, auditing, cyberattack responses
- **Research**
  - Math tools, low-cost supercomputing, display tools

115 FOSS Applications (DoD, 2003)

A	ACE	ACE ORB (TAO)	ACID	AMANDA	Apache	Autoconf	Automake
B	bash	Bastille	BIND				
C	C++ Boost	CIS Benchmarks	Colt	Condor	COPS	Crack	CVS
	CVW	Cygwin					
D	DDD	DjVuLibre					
E	EADSIM	Emacs	eTrust	Expect			
F	FreeBSD						
G	GateD	gawk (awk)	GCC	GDB	Ghostscript	GNAT	GnuPG
	gnuplot	grep					
H	h2n	HOSTS					
I	ImageMagick						
J	JADE	Jakarta	Jaxen	JBoss	JDOM	Jikes	jSIP
K	Kaffe						
L	LaTeX	Linux	Linux (Red Hat)	Linux firewalls	Lsof		
M	m4	Majordomo	make	Maxima	MIMEsweeper	MRTG	MTR
	MySQL						
N	Nessus	NetBSD	NetSaint	nload	Nmap	ntop	NTP
O	Octave	OpenBSD	OpenMap	OpenOffice	OpenSSH	OpenSSL	
P	Perl	Perl CGI scripts	PerLDAP	PHP	PingScan	Procmail	
Q	Qmail						
R	R	RealSecure	RRDtool	RTLinux	RWhois	RXVT	
S	Samba	SARA	SATAN	Saxon	SCA	sed	SELinux
	Sendmail	SNARE	Snort	Squid			
T	Tcl/Tk	TCP Wrappers	Tomcat	Top	Tripwire		
U							
V	VisAD	VOCAL	VTK				
W	Webmin	WebTAS	Weka	WU-FTPD			
X	Xalan	Xerces	XFree86	XGobi	Xpatch		
Y							
Z	zlib	Zope					

However: Why you *don't* want free source

- **A generous gift?**
  - Imagine: A major vendor offers to give you all of their operating system source code, with *no* strings attached!
  - Well, actually, two strings: You are fully responsible for maintaining your source copy; and you cannot resell it.
- **Factors to consider**
  - Expert support staff needed… 10, 100, 1000… more?
  - Where is your revenue stream? You cannot resell it!
  - Will you grow isolated? What if you lose compatibility?
- **Do the comparison**
  - Consider: How does the above differ from grabbing open source code to customize for internal-only use?

An Alternative: Share Support Costs

- **Why share the software burden?**
  - Your profits don’t come from selling software
  - You and your competitors are tired of high support costs
  - You form a consortium to share software support costs
- **Factors to consider**
  - What will your company’s total costs and duties be?
  - Which works best: Just a few members, or very many?
  - Can you reduce total costs enough to get a net benefit?
- **Do the comparison**
  - Consider: Consortia maximize their benefits if they are: (1) global, (2) “funded” via direct sharing of resources
  - Question: *How is that different from open source?*

Choose What You Share!

- **Which sharing strategies sound right?**

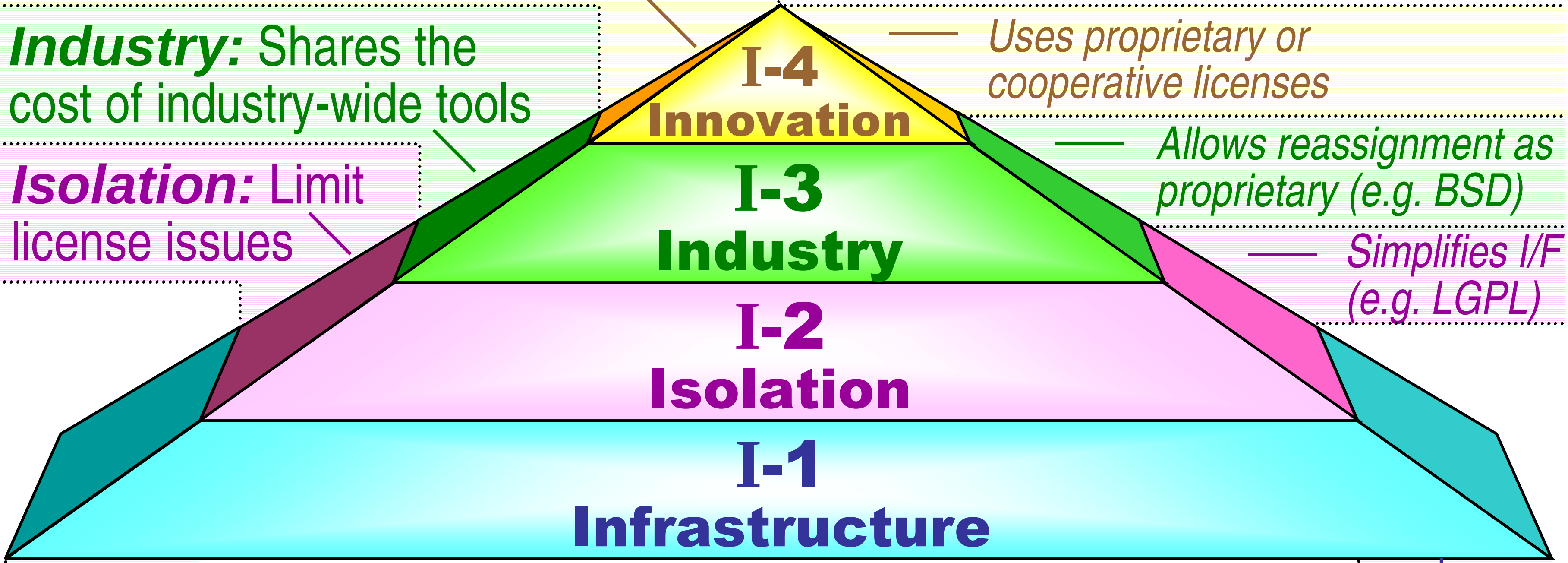
  Your company donates time, money, and expert personnel to a consortium that:

  A. Ensures that your generic operating system kernels and communications packages are reliable, efficient, and fully capable of supporting your company's needs

  B. Builds tools that are unique, but well understood overall, within your particular industry

  C. Asks members to share all forms of software, even if that includes your most market-critical innovations

The Open Source Pyramid: An Architecture for Maximizing Open Source Cost Benefits



Innovation: Software that solves a hard problem, or is unexpected

Uses proprietary or cooperative licenses

Industry: Shares the cost of industry-wide tools

Allows reassignment as proprietary (e.g. BSD)

Isolation: Limit license issues

Simplifies I/F (e.g. LGPL)

Broad mix of cooperative and consortia licenses (e.g., GPL, LGPL, BSD)

Infrastructure: Software whose value increases when it is shared more

Open Source Adoption Errors: Relying on “FOSS Magic” to Replace QA

- **“Let’s make this government software package open source.”**
  - Problem 1: No one will understand it (internal “dialects”)
  - Problem 2: For many apps, very few people will care
  - Problem 3: The modularity is all wrong
  - Problem 4: You just made your bugs a lot more visible
- **Better solutions:**
  - Don’t release it!
  - Freeze and encapsulate the application in new FOSS
  - Look for FOSS combinations that do the same job
  - Create & support a new FOSS effort to replace it
  - Look for a COTS solution

Open Source Adoption Errors: Assuming Others will Work for You for Free

- **Scenario: You create a license that:**
  - Opens up your code for anyone to see, and
  - Retains 100% ownership for your organization
- **Question:**
  - Will FOSS developers flock to help you?
- **Answer:**
  - Of course not!
  - Reason: Software cooperatives only work when *everyone* benefits comparably from the resulting product
  - Only when licenses give users equality of ownership do the cooperative incentives truly “kick in”
  - A good sanity check: Rural electric cooperatives

Open Source Adoption Errors: Disallowing All Proprietary Ownership

- **Scenario:**
    - A government group hires FOSS experts to create new software that will “glue together” FOSS applications
    - Without really thinking it through, the group insists that the new paid-for, fully custom code also be made FOSS
- **Likely outcomes:**
    - The contractors produce a quick prototype, then refuse to have anything more to do with it after funds run out
    - The contractors try to resell the code (which is perfectly legal), but also get very upset when the government group sends out copies of their code under FOSS rules
- **A Better Solution (others are possible):**
    - Use a timed proprietary license that converts into FOSS

Open Source Adoption Errors: Going it Alone (Source Code Cloning)

- **Scenario**
  - A government project begins a complex effort using all FOSS resources
  - The effort includes adaptation and extension of several of the FOSS applications
  - The effort is done mostly in-house, since the needs addressed are unique to the government organization
- **Results**
  - After initially low startup costs, development costs escalate and begin to resemble proprietary development
  - The problem: FOSS cost savings only occur for the *cooperatively* maintained parts of your infrastructure

  - **Solution:** *Always* minimize unique adaptations of FOSS

Open Source Adoption Errors: "All FOSS Applications Are the Same"

- **Scenario**
  - FOSS applications are chosen based solely on whether the have the right functionality
- **Results**
  - Some work, and some are dismal failures
- **Solution**
  - Like proprietary apps, FOSS varies hugely in maturity
  - Choose solutions based on maturity of:
    - FOSS product itself (e.g., Apache is very mature)
    - FOSS project (e.g., the support group for Apache is very mature)
  - A good resource:
    - *Succeeding with Open Source,* by Bernard Golden: http://www.navicasoft.com/

FOSS Quality: Better or Worse?

- **Overall, is FOSS higher or lower in quality than proprietary software?**
  - It varies! Very good (and very bad) apps exist for both
  - Biggest Open Source issues: Bad designs; lack of $$
  - Biggest Proprietary issues: Tendency to rely on on comforting (but highly deceptive) replication metrics
- **How does FOSS use impact total cost of ownership?**
  - Potentially quite favorably, *if* the products have strong community support *and* you join that community fully
  - Potentially quite badly, *if* you are foolish enough to clone source code and try to support it without any help

A Component-Oriented Approach to QA

- **QA helps *select, validate,* and *integrate* components of the Open Source Pyramid**
  - Use maturity models to assess FOSS projects/products
  - Share validations as broadly as possible
  - *Always* look at final-product performance, as judged by technically astute users with hands-on experience:
    - Any “mature” process that produces buggy products is… a charade
    - Any “certification” that produces buggy products is… a charade
- **Minimize development of new code:**
  - Take the component model to heart
  - Think in terms of *inheriting* both quality and quality support, rather than creating it new each time.
- **Look for automation! (e.g., Coverity)**

A Vector Addition Model of QA (Preliminary!)

- **Component-base design requires new interpretations of the QA problem. E.g.:**
  - Assign quality metrics to components based on prior assessments (local, community, or both)
  - Represent the quality of that component as a vector (arrow) pointing to the right, with a length that is initially proportional to the the product of (a) capabilities it provides and (b) the prior assessment of its quality
  - Shorten the vector based on how much additional new code is required to make the component functional. Weigh “distant” changes higher than “local” changes.
  - The shortening process can cancel or even reverse the initial quality vector for the component.
  - An interesting result: High-quality individual components and applications can end up at *negative* overall quality.

Bollinger Literary Terms (BLT)

Copyright 2004 by Terry Bollinger. Bollinger Literary Terms (BLT) License v1.1 applies: (1) Any reader of this work, including any incorporated entity, is granted the right to unlimited redistribution of this work in any medium or language, provided only that the meaning of the entire work and of this license remain complete and accurate in the best judgment of the redistributor. (2) Readers may at their own discretion redistribute this work either for profit or free. (3) If this work is bundled with other materials, the redistribution rights granted by the BLT license apply only to this work. (4) If brief, fair-use quotations of text or figures from this work are clearly attributed to this work by title and author name, the redistribution rights of the BLT license do not apply to them and do not need to be mentioned. (5) The original author of this work retains the right to repackage this work and subsets of this work under non-BLT licenses, with the provision that such secondary author licenses must not attempt to remove or diminish the redistribution rights granted to readers by the original BLT version of this work. Secondary author licenses that attempt to remove such rights are invalid and cannot be enforced.

Copyright 2006 by Terry Bollinger. BLT 1.1 applies (see final slide).