A View of 20\textsuperscript{th} and 21\textsuperscript{st} Century Software Engineering

Barry Boehm
ICSE 2006 Keynote Address
May 25, 2006
boehm@usc.edu
http://sunset.usc.edu
Outline

• Motivation and Context
• A 20th Century View
• A 21st Century View
• Conclusions
  – Timeless principles and aging practices
Motivation and Context

• Working definition of “software engineering”

• Importance of historical perspective

• We are losing our history

• Some historical perspectives
“Software Engineering” Definition
- Based on Webster definition of “engineering”

• The application of science and mathematics by which the properties of software are made useful to people

• Includes computer science and the sciences of making things useful to people
  - Behavioral sciences, economics, management sciences
Why Software Projects Fail
- Average overrun: 89.9% on cost, 121% on schedule, with 61% of content

352 companies - 8,000 software projects. Source: The Standish Group, 1995
Importance of Historical Perspective

• Santayana half-truth:
  – “Those who cannot remember the past are condemned to repeat it”

• Don’t remember failures?
  – Likely to repeat them

• Don’t remember successes?
  – Not likely to repeat them
We Are Losing Our History

• Great people are gone
  – Hopper, Mills, Royce, Dijkstra, Dahl, Nygaard, Weiser, ...

• Relative inaccessibility of hardcopy literature
  – Median ICSE 2005 paper has no reference before 1984-85
    • 77% have no references before 1980
Some Historical Perspectives

• 1996 Dagstuhl Workshop
  – Historians: Aspray, Bergin, Ceruzzi, Mahoney, Tomayko, …
  – Practitioners: Endres, Parnas, Randell, Ross, Shaw, …
  – Provided perspectives and insights adapted here

• 2001 Software Pioneers conference and book (Broy, Denert; Springer)

• NSF/ACM IMPACT Project: impact of research on practice
  – To date: Configuration Management, Programming Languages
  – Coming: Reviews, Testing and Analysis, Design, Reuse, Middleware, Resource Estimation

• A Hegelian View (drafted at Dagstuhl workshop)
  – Thesis, Antithesis, Synthesis
A Hegelian View of Software Engineering Evolution

Theses
- Engineer Software like Hardware
- Many defects
- Software as Craft
- Software Differences, Engineer Shortages

Syntheses
- Formality, Waterfall
- Productivity; Reuse; Objects; Peopleware
- Scalability, Risk Mgmt.
- Prototyping

Antitheses
- Compliance
- Plan-Driven Software Maturity Models
- Risk Management
- Agile Methods
- Time to Market, Rapid Change

1950's
1960's
1970's
1980's
1990's
2000's
2010's

Integrated Sw-Systems Engineering
Software Value-Add
COTS
Risk-Based Agile/Plan-Driven Hybrids; Model-Driven Development
Value-Based Methods; Collaboration; Global Development; Enterprise Architectures
Soft SysE

Autonomy; Bio-Computing

Process Overhead
Scalability
Global Systems of Systems

© USC-CSE
Outline

• Motivation and Context

→ A 20\textsuperscript{th} Century View

• A 21\textsuperscript{st} Century View

• Conclusions
  – Timeless principles and aging practices
1950’s Thesis: Engineer Software Like Hardware

• Hardware-oriented:
  – Software applications: airplanes, bridges, circuits
  – Economics: Boehm supervisor, 1955
    • “We’re paying $600/hour for that computer, and $2/hour for you, and I want you to act accordingly.”
  – Professional Societies: Association for Computing Machinery, IEEE Computer Society
  – Software Processes: SAGE (Semi-Automated Ground Environment)
    • 1 MLOC air defense system, real-time, user-intensive
    • Successful development of highly unprecedented system
    • Hardware-oriented waterfall-type process
The SAGE Software Development Process - (Benington, 1956)

“We were successful because we were all engineers.”
1960’s Antithesis: Software Is Not Like Hardware
- Four Brooks factors plus two

• Invisibility: like the Emperor’s Magic Cloth
• Complexity: Royce, “for a $5M procurement, need a 30-page spec for hardware, and a 1500-page spec for software”
• Conformity: executed by computers, not people
• Changeability: up to a point, then becomes difficult
• Doesn’t wear out: different reliability, maintenance phenomena
• Unconstrained: can program antigravity, time travel, interpenetration, …
1960’s Antithesis: Software Crafting

- Flexible materials, frequent changes as above
- SW demand exceeded supply of engineers
  - Music, history, art majors
  - Counter culture: question authority
  - Cowboy programmers as heroes
  - Code-and-fix process
  - Hacker culture (Levy, 1984)
    - Collective code ownership
    - Free software, data, computing access
    - Judge programmers by the elegance of their code
1960’s Progress and Problems

• Better infrastructure: OS, compilers, utilities
• Computer Science Departments
• Product Families: OS-360, CAD/CAM, math/statistics libraries
• Some large successes: Apollo, ESS, BofA check processing
• Problems: 1968, 1969 NATO Reports
  – Failure of most large systems
  – Unmaintainable spaghetti code
  – Unreliable, undiagnosable systems
1970’s Antithesis: Formal and Waterfall Approaches

• **Structured Methods**
  – Structured programming (Bohm-Jacopini: GO TO unnecessary)
    • Formal programming calculus: Dijkstra, Hoare, Floyd
    • Formalized Top-Down SP: Mills, Baker

• **Waterfall Methods**
  – Code and fix too expensive (100:1 for large systems)
  – Precede code by design (De Marco SD, Jackson JSD/JSP)
  – Precede design by requirements (PSL/PSA, SA, SREM)
The Royce Waterfall Model (1970)
- Explicit feedback
- “Do it twice”
Increase in Software Cost-to-fix vs. Phase (1976)

- Larger Software Projects
  - IBM-SSD
  - GTE
  - Median (TRW Survey)
  - SAFEGUARD

- Smaller Software Projects

Phase in Which defect was fixed:
- Requirements
- Design
- Code
- Development test
- Acceptance test
- Operation

Relative cost to fix defect:
- 1
- 2
- 5
- 10
- 20
- 50
- 100
- 200
- 500
- 1000

05/25/06
1970’s: Problems with Formal Methods

• Successful for small, critical programs
• Largest proven programs around 10 KSLOC
• Proofs show presence of defects, not absence
  – Defects in specification, proofs happen
• Scalability of programmer community
  – Techniques require math expertise, $500/SLOC
  – Average coder in 1975 survey:
    • 2 years of college, SW experience
    • Familiar with 2 languages, applications
    • Sloppy, inflexible, in over his head, and undermanaged
1970’s: Problems with Waterfall Model

• Overly literal interpretation of sequential milestones
  – Prototyping is coding before Critical Design Review
  – Mismatch to user-intensive systems

• Heavyweight documentation hard to review, maintain
  – 7 year project with 318% requirements change
  – Milestones passed but not validated

• Mismatch to COTS, reuse, legacy SW
  – Bottom-up vs. top-down processes

• Scalability, cycle time, and obsolescence
  – Months = $5 \times 3^{3/4} \sqrt{KSLOC}$ for sequential development
  – 3000 KSLOC $\Rightarrow 5 \times 14.4 = 72$ months : 4 computer generations
A Hegelian View of Software Engineering Evolution
1980’s Synthesis: Productivity, Reuse, Objects

• Worldwide concern with productivity, competitiveness
  – Japanese example: autos, electronics, Toshiba SW reuse

• Major SW productivity enhancers
  – Working faster: tools and environments
  – Working smarter: processes and methods
  – Work avoidance: reuse, simplicity; objects
  – Technology silver bullets: AI, transformations, DWIM, PBE
    • Do what I mean; programming by example
Tools, Environments, and Process

• Overfocus on programming: IPSEs to SEEs
  – Requirements, design, planning and control, office support
  – Formatted vs. formal specifications

• Process-driven SEEs
  – Use process knowledge as tool integration framework
  – Some overkill in locking people into roles

• Process execution support: “SW processes are SW too”
  – What’s good for products is good for processes
  – Reuse, prototyping, architecture, programming

• Process compliance support: Standards and CMMs
Reuse and Object Orientation

- 1950’s: Math routines, utilities
- 1960’s: McIlroy component marketplace, Simula – 67
- 1970’s: Abstract data types, Parnas program families
- 1980’s: Smalltalk, Eiffel, C++, OO methods, reuse libraries
- 1990’s: Domain engineering, product lines, UML, pub-sub architectures
- 2000’s: Model driven development, service oriented architectures
HP Product Line Reuse Investment and Payoff

Calendar Months to Release

Month and Year Released

Jan-86  Mar-87  Jul-87  Aug-89  Sep-90  Apr-91  Nov-91  Feb-92
No Silver Bullet: Brooks

• Automated solutions are good for “accidental” software problems
  – Simple inconsistencies, noncompliance, inferences

• They do not do well on “essential” software problems
  – Changeability: adapting themselves to unanticipated changes
  – Conformity: working out everything the computer needs to “know”
    • Devoid of intuition, commonsense reasoning
  – Complexity: integrating multiple already- complex programs
  – Invisibility: communicating their likely behavior to humans

• Closest thing to silver bullet: great designers
People: The Most Important Factor

- SW engineering is of the people, by the people, and for the people

• 1970’s: Weinberg *Psychology of Computer Programming*
• 1980’s: COCOMO factor-of-10, Scandinavian Participatory Design, DeMarco-Lister *Peopleware*
• 1990’s – 2000’s: Importance emphasized in both Agile and CMM cultures
  - Individuals and interactions over process and tools
  - People CMM, Personal Software Process
• Overall migration from Reductionism toward Postmodernism (Toulmin)
  - Universal towards Local
  - General towards Particular
  - Timeless towards Timely
  - Written towards Oral
Dual 1990’s – Early 2000’s Antithesis:  
- Maturity Models and Agile Methods

• Predictability and Control: Maturity Models
  – Reliance on explicit documented knowledge
  – Heavyweight but verifiable, scalable

• Time to Market and Rapid Change: Agile Methods
  – Reliance on interpersonal tacit knowledge
  – Lightweight, adaptable, not very scalable
Agile and Plan-Driven Home Grounds: Five Critical Decision Factors

- Size, Criticality, Dynamism, Personnel, Culture

**Criticality**
(Loss due to impact of defects)
- a: Many Lives
- b: Single Life
- c: Essential Funds
- d: Discretionary Funds
- e: Comfort

**Dynamism**
(% Requirements – change/month)

**Size**
(# of personnel)

**Personnel**
(% Level 1B) (% Level 2&3)

**Culture**
(% thriving on chaos vs. order)

- Agile
- Plan-driven
Other 1990’s – Early 2000’s Developments

- Y2K and reverse engineering
- Risk-driven concurrent engineering
  - Win-Win spiral with anchor points; Rational Unified Process
- Nature and importance of software architecture
- COTS, open source, and legacy software
- Software as the primary competitive discriminator
  - 80% of aircraft functionality
Spiral Model and Concurrent Engineering

Phases:
- Inception
- Elaboration
- Construction
- Transition

Disciplines:
- Business Modeling
- Requirements
- Analysis & Design
- Implementation
- Test
- Deployment
- Configuration & Change Mgmt
- Project Management
- Environment

Time:
- LCO
- LCA
- IOC

Iterations:
- Initial
- Elab #1
- Elab #2
- Const #1
- Const #2
- Const #N
- Tran #1
- Tran #2
COTS: The Future Is Here

- Escalate COTS priorities for research, staffing, education
  - Software is not “all about programming” anymore
  - New processes required

CBA Growth Trend in USC e-Services Projects

- CBA: COTS-Based Application
  * Standish Group CHAOS 2000 (54%)
A Hegelian View of Software Engineering Evolution

Theses
- Engineer Software like Hardware
- Many defects
- Formality, Waterfall
- Productivity; Reuse; Objects; Peopleware
- Compliance
- Plan-Driven Software Maturity Models
- Risk-Based Agile/Plan-Driven Hybrids; Model-Driven Development
- Software Value-Add
- Integrated Sw-Systems Engineering

Syntheses
- Scalability, Risk Mgmt.
- Process Overhead
- Domain Engr.
- Risk Mgmt.
- Scalability

Antitheses
- Software Differences, Engineer Shortages
- Software as Craft
- Prototyping
- Time to Market, Rapid Change
- Agile Methods


© USC-CSE
Mid-2000’s Synthesis: Risk-Driven Hybrid Products and Process

• Increasing integration of systems engineering and SW engineering
  – Increasing trend toward “soft systems engineering”

• Increasing focus on usability and value
  – Fit the software and hardware to the people, not vice versa

• Model-driven development and service-oriented architectures

• Emergent vs. prespecifiable requirements

• Hybrid agile and plan-driven product and process architectures
  – Encapsulate parts with rapid, unpredictable change
  – Concurrent build-to-spec, V&V, agile rebaselining
MDA Adoption Thermometer
- Gartner Associates, 2003

Key: Time to Plateau
- Less than two years
- Two to five years
- Five to 10 years
- More than 10 years

Acronym Key
ARAD  architected, rapid application development
SCCM  software change and configuration management
SODA  service-oriented development of applications

As of May 2003
Risk-Driven Scalable Spiral Model: Increment View

- **Rapid Change**
  - Short Development Increments
  - Foreseeable Change (Plan)

- **High Assurance**
  - Stable Development Increments

- **Increment N Baseline**
  - Short, Stabilized Development of Increment N
  - Increment N Transition/O&M
Risk-Driven Scalable Spiral Model: Increment View

- **Rapid Change**
  - Unforeseeable Change (Adapt)
  - Short Development Increments
  - Foreseeable Change (Plan)

- **High Assurance**
  - Stable Development Increments
  - Current V&V Resources
  - Continuous V&V

- **Increment N Baseline**
  - Agile Rebaselining for Future Increments
  - Deferrals

- **Short, Stabilized Development of Increment N**
  - Artifacts
  - Concerns

- **Future Increment Baselines**
  - Increment N Transition/O&M

- **V&V of Increment N**
  - Future V&V Resources
Outline

• Motivation and Context

• A 20\textsuperscript{th} Century View

→ A 21\textsuperscript{st} Century View

• Conclusions
  – Timeless principles and aging practices
The Future of Systems and Software

• **Eight surprise-free trends**
  1. Increasing integration of SysE and SwE
  2. User/Value focus
  3. Software Criticality and Dependability
  4. Rapid, Accelerating Change
  5. Distribution, Mobility, Interoperability, Globalization
  6. Complex Systems of Systems
  7. COTS, Open Source, Reuse, Legacy Integration
  8. Computational Plenty

• **Two wild-card trends**
  1. Autonomy Software
  2. Combinations of Biology and Computing
Pareto 80-20 distribution of test case value [Bullock, 2000]

*Usual SwE assumption for all requirements, objects, defects,
Business Case for Value-Based Testing
Globalization: “The World is Flat”
- Friedman, 2005

• Information processing functions can be performed almost anywhere in the world
  – Low-cost global fiber-optic communications
  – Overnight global delivery services

• Significant advantages in outsourcing to low-cost suppliers
  – But significant risks also

• Competitive success involves pro-actively pursuing advantages
  – While keeping risks manageable
What does a SISOS look like?

- Network-Centric Air Traffic Control

[Diagram showing network-centric air traffic control systems involving satellite navigation, SAT NAV Monitor (WAAS), terminal radar systems, ARTCC (HOCSR, SDP, VDL, DSR), en route radar systems, and associated functionality such as ranging, time, connection, and integrity.]
Integrated Enterprise Architectures

Federal Enterprise Architectural Framework (FEAF)

DOD Architectural Framework (DODAF)

Zachman Framework

Scope

Business Model

System Model

Technology Model/Detailed Representations

Data View

Function View

Network View

People View

Time View

Motivation View
Persistence of Legacy Systems

• Before establishing new-system increments
  – Determine how to undo legacy system

1939’s Science Fiction World of 2000

Actual World of 2000
Computational Plenty: Process Implications

• New platforms: smart dust, human prosthetics (physical, mental)
  – New applications: sensor networks, nanotechnology

• Enable powerful self-monitoring software
  – Assertion checking, trend analysis, intrusion detection, proof-carrying code, perpetual testing

• Enable higher levels of abstraction
  – Pattern programming, programming by example with dialogue
  – Simpler brute-force solutions: exhaustive case analysis

• Enable more powerful software tools
  – Based on domain, programming, management knowledge
  – Show-and-tell documentation
  – Game-oriented software engineering education
Wild Cards: Autonomy and Bio-Computing

• Great potential for good
  – Robot labor; human shortfall compensation
    • 5 Senses, healing, life span, self-actualization
  – Adaptive control of the environment
  – Redesigning the world for higher quality of life
    • Physically, biologically, informationally

• Great potential for harm
  – Loss of human primacy: computers propose, humans decide
  – Overempowerment of humans
    • Accidents, terrorism, Enron California brownouts
  – New failure modes: adaptive control instability, self-modifying software, commonsense reasoning, bio-computer mismatches
  – V&V difficulties: cooperating autonomous agents, biocomputing

• Forms and timing of new capabilities still unclear
Outline

• Motivation and Context
• A 20\textsuperscript{th} Century View
• A 21\textsuperscript{st} Century View

\rightarrow Conclusions

– Timeless principles and aging practices
Timeless Principles (+) and Aging Practices (-)

• From the 1950’s
  + Don’t neglect the sciences
  + Look before you leap (avoid premature commitments)
    - Avoid inflexible sequential processes

• From the 1960’s
  + Think outside the box
  + Respect software’s differences
    - Avoid cowboy programming
Timeless Principles (+) and Aging Practices (-)

• From the 1970’s
  + Eliminate errors early
  + Determine the system’s purpose
  - Avoid top-down development and reductionism

• From the 1980’s
  + These are many roads to increased productivity
  + What’s good for products is good for processes
  - Be skeptical about silver bullets
Timeless Principles (+) and Aging Practices (-)

• From the 1990’s
  + Time is money and value to people
  + Make software useful to people
  - Be quick, but don’t hurry

• From the 2000’s
  + If change is rapid, adaptability trumps repeatability
  + Consider and satisfice all of your stakeholders’ value propositions
  - Avoid falling in love with your slogans (e.g. YAGNI)
Timeless Principles (+) and Aging Practices (-)

• For the 2010’s
  + Keep your reach within your grasp
  + Have an exit strategy
  - Don’t believe everything you read
    - “It’s true because I read it on the Internet”
Future Challenges for SW Engineering Education
- Student careers go through 2050's

• Keeping courseware continually up-to-date
• Anticipating future trends and preparing students for them
• Separating timeless principles from aging practices
• Making small student projects relevant to large industry practices
• Participating in research; incorporating results in courses
• Helping students learn how to learn
• Offering lifelong learning to practitioners