No Silver Bullet

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 5828 — Lecture 2 — 01/14/2010

© University of Colorado, 2010
Lecture Goals

- Introduce thesis of Fred Brook’s No Silver Bullet
  - Classic essay by Fred Brooks discussing “Why is SE so hard?”
No Silver Bullet

“There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.”

— Fred Brooks, 1986

i.e. There is no magical cure for the “software crisis”
Brooks divides the problems facing software engineering into two categories:

- essence: difficulties inherent in the nature of software
- accidents: difficulties related to the production of software

Brooks argues that most techniques attack the accidents of software engineering.
In order to improve software development by a factor of 10

- first, the accidents of software engineering would have to account for 90% of the overall effort
- second, tools would have to reduce accidental problems to zero

Brooks doesn't believe that the former is true…

- and the latter is nigh impossible because each new tool or technique solves some problems while introducing others
The Essence

Brooks divides the essence into four subcategories:
- complexity
- conformity
- changeability
- invisibility

Let's consider each in turn
Software entities are amazingly complex
  - No two parts (above statements) are alike
  - Contrast with materials in other domains

Large software systems have a huge number of states
  - Brooks claims they have an order of magnitude more states than computers (i.e. hardware) do

As the size of a system increases, both the number and types of parts increase exponentially
  - the latter increase is the most significant
You can't abstract away the complexity of the application domain. Consider:

- air traffic control, international banking, avionics software

These domains are intrinsically complex and this complexity will appear in the software system as designers attempt to model the domain.

- Complexity also comes from the numerous and tight relationships between heterogeneous software artifacts such as specs, docs, code, test cases, etc.
Complexity (III)

Problems resulting from complexity
- difficult team communication
- product flaws; cost overruns; schedule delays
- personnel turnover (loss of knowledge)
- unenumerated states (lots of them)
- lack of extensibility (complexity of structure)
- unanticipated states (security loopholes)
- project overview is difficult
A lot of complexity facing software engineers is arbitrary.

Consider designing a software system for an existing business process and a new VP arrives at the company.

The VP decides to “make a mark” on the company and changes the business process.

Our system must now conform to the (from our perspective) arbitrary changes imposed by the VP.
Other instances of conformity

- Having to integrate with a non-standard module interface
- Adapting to a pre-existing environment
  - and if the environment changes (for whatever reason), you can bet that software will be asked to change in response

Main Point: It is almost impossible to plan for arbitrary change;
- instead, you just have to wait for it to occur and deal with it when it happens
Software is constantly asked to change

Other things are too, however, manufactured things are rarely changed after they have been created

- instead, changes appear in later models
  - automobiles are recalled only infrequently
  - buildings are expensive to remodel
Changeability (II)

- With software, the pressure to change is greater
  - in a project, it is functionality that is often asked to change and software EQUALS functionality (plus its malleable)
  - clients of a software project often don't understand enough about software to understand when a change request requires significant rework of an existing system
  - Contrast with more tangible domains
    - Imagine asking for a new layout of a house after the foundation has been poured
Invisibility (I)

- Software is by its nature invisible; and it is difficult to design graphical displays of software that convey meaning to developers.
  - Contrast to blueprints: here geometry can be used to identify problems and help optimize the use of space.
- But with software, it's difficult to reduce it to diagrams.
  - UML contains 13 different diagram types (!):
    - to model class structure, object relationships, activities, event handling, software architecture, deployment, packages, etc.
Invisibility (II)

- Hard to get both a “big picture” view as well as details
  - Hard to convey just one issue on a single diagram
  - Instead multiple concerns crowd and/or clutter the diagram hindering understanding
- This lack of visualization deprives the engineer from using the brain's powerful visual skills
What about “X”?  

- Brooks argues that past breakthroughs solve accidental difficulties
  - High-level languages
  - Time-Sharing
  - Programming Environments
  - OO Analysis, Design, Programming
  - ...


Promising Attacks on the Essence

- **Buy vs. Build**
  - Don't develop software when you can avoid it
- **Rapid Prototyping**
  - Use to clarify requirements
- **Incremental Development**
  - Don't build software, grow it
- **Great designers**
  - Be on the look out for them, when you find them, don't let go!
Brooks reflects on No Silver Bullet, ten years later

Lots of people have argued that their methodology, technique, or tool is the silver bullet for software engineering

If so, they didn't meet the deadline of 10 years or the target of a 10 times improvement in the production of software

Others misunderstood what Brooks calls “obscure writing”

e.g., “accidental” did not mean “occurring by chance”;

instead, he meant that the use of technique A for benefit B unfortunately introduced problem C into the process of software development
Some people misunderstood his point with the 90% figure. Brooks doesn't actually think that accidental effort is 90% of the job. It's much smaller than that. As a result, reducing it to zero (which is effectively impossible) will not give you an order of magnitude improvement.
Some people interpreted Brooks as saying that the essence could never be attacked.

That's not his point; he said that no single technique could produce an order of magnitude increase by itself.

He argued that several techniques in tandem could achieve it but that requires industry-wide enforcement and discipline.

Brooks states:

“We will surely make substantial progress over the next 40 years; an order of magnitude improvement over 40 years is hardly magical…”
Quiz Yourself

- **Essence or Accident?**
  - A bug in a financial system is discovered that came from a conflict in state/federal regulations on one type of transaction.
  - A program developed in two weeks using a whiz bang new application framework is unable to handle multiple threads since the framework is not thread safe.
  - A new version of a compiler generates code that crashes on 32-bit architectures; the previous version did not.
  - A fickle customer submits 10 change requests per week after receiving the first usable version of a software system.
Returning to SE Intro

- Lets continue our “Overview of Software Engineering” that was started in Lecture 1
  - This draws on material from *Software Engineering: Theory and Practice* by Pfleeger and Atlee
    - As such, some material is copyright © 2006 Pearson/Prentice Hall.
What is Software Engineering?

Simply Put: It’s about solving problems with software-based systems

- Design and development of these systems require
  - Analysis
    - decomposing large problems into smaller, understandable pieces
      - abstraction is the key
  - Synthesis
    - building large software systems from smaller building blocks
      - composition is challenging
To aid us in solving problems, we apply

- **techniques**: a formal “recipe” for accomplishing a goal that is typically independent of the tools used
  - procedure for thickening a sauce without causing it to curdle
- **tools**: an instrument or automated system for accomplishing something in a better way, where “better” can mean more efficient, more accurate, faster, etc.
To aid us in solving problems, we apply

- **procedures**: a combination of tools and techniques that, in concert, produce a particular product

- **paradigms**: a particular philosophy or approach for building a product

  - Think: “cooking style”: may share procedures, tools, and techniques with other styles but apply them in different ways
  
  - Example: OO approach to development vs. the structured approach
Software engineering has helped to produce systems that improve our lives in numerous ways:

- helping us to perform tasks more quickly and effectively
- supporting advances in medicine, agriculture, transportation, and other industries

Indeed, software-based systems are now ubiquitous:

- How many computers do you have in your home?
- How many times do you interact with a software-based system each day?
Software is not without its problems

- Systems function, but not in the way we expect
- Or systems crash, make mistakes, etc.
- Or the process for producing a system is riddled with problems leading to a failure to produce the entire system
  - many projects get cancelled without ever producing a system

One study in the late 80s found that in a survey of 600 firms, more than 35% reported having a runaway development project. A runaway project is one in which the budget and schedule are completely out of control.
CHAOS Report from Standish Group

- Has studied over 40,000 industry software development projects over the course of 1994 to 2004.
- Success rates (projects completed on-time, within budget) in 2004 was 34%, up from 16.2% in 1994
- Failure rates (projects cancelled before completion) in 2004 was 15%, down from 31% in 1994.
- In 2004, “challenged” projects made up 51% of the projects included in the survey.
  - A challenged project is one that was over time, over budget and/or missing critical functionality
Most challenged projects in 2004 had a cost overrun of under 20% of the budget, compared to 60% in 1994.

The average cost overrun in 2004 was 43% versus an average cost overrun of 180% in 1994.

In 2004, total U.S. project waste was 55 billion dollars with 17 billion of that in cost overruns; Total project spending in 2004 was 255 billion.
In 1994, total U.S. project waste was 140 billion (80 billion from failed projects) out of a total of 250 billion in project spending.

So, things are getting better (attributed to better project management skills industry wide), but we've still got a long way to go!

66% of the surveyed projects in 2004 did not succeed!
Software Engineering: The Ugly (I)

- Loss of NASA’s Mars Climate Observer
  - due to mismatch of English and Metric units!
  - even worse: problem was known but politics between JPL and Houston prevented fix from being deployed
- Leap-year bug
  - A supermarket was fined $1000 for having meat around 1 day too long on Feb. 29, 1988
- Denver International Airport
  - Luggage system: 16 months late, 3.2 billion dollars over budget!
IRS hired Sperry Corporation to build an automated federal income tax form processing process

- An extra $90 M was needed to enhance the original $103 M product
- IRS lost $40.2 M on interest and $22.3 M in overtime wages because refunds were not returned on time
Therac-25 (safety critical system: failure poses threat to life or health)

- Machine had two modes:
  - “electron beam” and “megavolt x-ray”
  - “megavolt” mode delivered x-rays to a patient by colliding high energy electrons into a “target”

- Patients died when a “race condition” in the software allowed the megavolt mode to engage when the target was not in position

- Related to a race between a “type ahead” feature in the user interface and the process for rotating the target into position
An **error** is a mistake made by a human

A **fault** is the manifestation of the error in a software artifact

A **failure** is a departure from a system’s expected behavior
What is Good Software?

“Good” is often associated with some definition of quality. The higher the quality, the better the software.

The problem? Many different definitions of quality!

- **Transcendental:** where quality is something we can recognize but not define (“I know it when I see it”)
- **User:** where quality is determined by evaluating the fitness of a system for a particular purpose or task (or set of tasks)
- **Manufacturing:** quality is conformance to a specification
What is Good Software?

- Many different definitions of quality!
  - **Product**: quality is determined by internal characteristics (e.g. number of bugs, complexity of modules, etc.)
  - **Value**: quality depends on the amount customers are willing to pay
    - customers adopt “user view”; developers adopt “manufacturing view”; researchers adopt “product view”; “value view” can help to tie these together
What is Good Software?

Good software engineering must always include a strategy for producing high quality software.

Three common ways that SE considers quality:

- The quality of the product (product view)
- The quality of the process (manufacturing view)
- The quality of the product in the context of a business environment (user view)

The results of the first two are termed the “technical value of a system”; The latter is the “business value of a system”
The Quality of the Product

- Users judge a system on external characteristics
  - correct functionality, number of failures, types of failures
- Developers judge the system on internal characteristics
  - types of faults, reliability, efficiency, etc.
- Quality models can be used to relate these two views
- An example is McCall’s quality model
  - This model can be useful to developers: want to increase “reliability” examine your system’s “consistency, accuracy, and error tolerance”
The Quality of the Process (I)

- Quality of the development and maintenance process is as important as the product quality
- The development process needs to be modeled
The Quality of the Process (II)

Modeling will address questions such as

- What steps are needed and in what order?
- Where in the process is effective for finding a particular kind of fault?
- How can you shape the process to find faults earlier?
- How can you shape the process to build fault tolerance into a system?
The Quality of the Process (III)

- Models for Process Improvement
  - SEI’s Capability Maturity Model (CMM)
  - ISO 9000
  - Software Process Improvement and Capability Determination (SPICE)
Business Environment
Quality (I)

The business value being generated by the software system
- Is it helping the business do things faster or with less people?
  - Does it increase productivity?

To be useful, business value must be quantified.
A common approach is to use “return on investment” (ROI).

Problem: Different stakeholders define ROI in different ways!

- Business schools: “what is given up for other purposes”
- U.S. Government: “in terms of dollars, reducing costs, predicting savings”
- U.S. Industry: “in terms of effort rather than cost or dollars; saving time, using fewer people”
Differences in definition means that one organization’s ROI cannot be compared with another organization’s ROI without careful analysis.
SE-Related Sites/Blogs

- slashdot.org; joelonsoftware.com
- http://www.tbray.org/ongoing/
- stackoverflow.com; loudthinking.com, semat.org

Humor:
- xkcd.org, The Order of the Stick, thedailywtf.com

Please send me others that you find useful
Coming Up Next

- Lecture 3: Pleasing Your Customer
  - Chapter 1 of Pilone & Miles
- Lecture 4: Introduction to Concurrency
  - Chapter 1 of Breshears