16
No Silver Bullet – Essence and Accident in Software Engineering

No Silver Bullet

- Abstract
- Introduction
- Does It Have to Be Hard? - Essential Difficulties
- Past Breakthroughs Solved Accidental Difficulties
- Hopes for the Silver
- Promising Attacks on the Conceptual Essence

Abstract

- Essential task
  - Construction of the complex conceptual structures that form the abstract software entity
- Accidental tasks
  - Representation of the abstract software entities in programming languages
- Prior gains in software productivity have come from removing obstacles in accidental tasks

Abstract

- Large gains cannot be achieved by simply reducing the effort required for accidental tasks
- Essential tasks should be addressed
- Brooks suggests the following
  - Exploiting the mass market to avoid constructing what can be bought
  - Using rapid prototyping
  - Growing software *organically*, adding more and more function to systems as they are run, used and tested
  - Identifying and developing great conceptual designers of the rising generations

Introduction

- Problems usually associated with software projects
  - Missed schedules, blown budgets, and flawed products
- No single development in technology or management technique can improve the productivity by an order of magnitude
- However, many encouraging innovations going on.
- Disciplined and consistent effort required to improve the productivity

Essential Difficulties

- Essential nature of software makes it difficult to have a *silver bullet* for increasing the productivity
- Progress of computer hardware is very fast – six orders of magnitude price-performance gain in last 30 years
- Difficulties in progress of software
  - Essence : inherent in the nature of software
  - Accidents : not inherent in the nature of software

Essence

- Essence of a software entity is the construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocation of functions
- Highly precise and detailed
- Brooks says that the difficult parts of building software are specification, design, and testing of the conceptual construct

Essence : Complexity

- Complexity of software is an essential property
- Software systems have a very large number of states making it perhaps more complex for its size than any human construct
- Scaling up : Requires increase in number of elements and not merely repetition of existing elements
- Complexity leads to the difficulty of communication among team members leading to product flaws, cost overruns, and schedule delays
- Leads to technical problems and management problems

Essence : Conformity

- Much of the complexity in software systems is arbitrary unlike physical sciences, where there is a unifying principle
- Software must conform to various interfaces, which is because different interfaces were designed by different people
- This complexity cannot be simplified by simply redesigning the software

Essence : Changeability

- Software is constantly under the pressure for change
- Other systems such as cars, buildings are also subject to change but very infrequently as compared to software
- Partly because software can be changed easily with much less cost as compared to cars or buildings
- Also because software in a system is the functionality of the system and therefore is subject to pressure for change

Essence : Changeability

- Almost all software products changes over time
  - Extended, new usage beyond the original domain etc.
- Successful software also survives beyond the normal life of the system which it is originally written for
  - A change in the system forces change in the software

Essence: Invisibility

- Geometric abstractions help in the design of any systems as it helps in visualization of the system
  - E.g. floor plan of a building, diagrams of silicon chips etc.
- This is inherently not possible with software systems
- Usually represented in terms of several directed graphs, superimposed one upon another
- These represent the flow of control, flow of data, patterns of dependency, time sequence, name-space relationships.

Essence: Invisibility

- These are usually not even planar and also not hierarchical leading to conceptual difficulties
- In spite of simplifying the structures of software, they inherently remain un-visualizable

Past breakthroughs for accidental difficulties

- High-level languages
- One of the most important innovation for increasing the software productivity
- Improvement of a factor of ~5 in software productivity
- Frees a program from much of accidental complexity
- However, most a high-level language can do is furnish all the constructs that a programmer imagines in the abstract program

High-level languages

- Level of sophistication in thinking about data structures, data types, and operations are steadily rising, but at a decreasing rate
  - Language development has kept pace with this sophistication
- Further, at some point the difficult constructs of a language becomes burden

Time sharing

- Major improvement due to time sharing but not as much as due to high-level languages
- Time sharing preserves immediacy and hence enables us to maintain an overview of complexity
- Slow turnaround of batch programs leads to loss in the continuity of thinking – leads to loss of time
- Slow turnaround is an accidental difficulty
- The principal effect of time sharing is to shorten the system response time
- Beneficial only until the threshold of human perception, ~100 milliseconds

Unified programming environments

- Integrated programming environments like Unix attack accidental difficulties by using programs together
- Provides integrated libraries, unified file formats and pipes and filters
- Conceptual structures can interact with each other easily
- Led to development of whole toolbenches, such that each new tool can be applied to any programs using standard formats

Hopes for the silver

- Advances in high-level languages
- Object-oriented programming
- Artificial intelligence
- Expert Systems
- *Automatic* Programming
- Graphical Programming
- Program Verification
- Environments and tools
- *Workstations*

Advances in high-level languages

- Advances in the features supported by high-level languages would increase the software productivity
- But still it would not prove to be the silver bullet

Object Oriented programming

- Object oriented programming has been a great advancement in the *art* of software development
- Removes the accidental difficulty from the process allowing the designer to express the essence of his design without having to express large amount of the syntactic material
- However this removes only accidental difficulties, complexity of the design remains

Artificial Intelligence

- Brooks does not think that advancements in AI will lead to an order of magnitude improvement in software productivity
- Parnas clarifies two definitions of AI:
  - AI-1: Use of computers to solve problems that could only be solved by applying human intelligence
  - AI-2: The use of specific set of programming techniques known as heuristic or rule based programming

Artificial Intelligence

- Parnas criticizes these by saying that it is difficult to identify a body of technology which is unique to this. Brooks agrees with this
- Brooks questions that how any image recognition technique will make any appreciable difference to programming practice
- Difficult part in software development is deciding what to express and not how to express it

Expert Systems

- Most advanced part of the artificial intelligence
- Contains a generalized inference engine and a rule base, designed to take input data and assumptions and explore the logical consequences through inferences derivable from rule base
- Inference engine can typically deal with fuzzy or probabilistic data, in addition to purely deterministic data

Expert Systems

- Advantages over programmed algorithms
  - Inference engine technology is developed in an application independent way
  - The changeable parts of the application dependent materials are encoded in the rule base in a uniform fashion, and tools are provided for developing, changing, testing and documenting the rule base

Expert Systems

- Leads to separation of application complexity with from the program
- Application to software systems
  - Suggesting interface rules, advising on testing strategies, remembering bug-type frequencies etc.
- Generation of diagnostic rules will be done while creating the test cases
  - If done suitably, will help in maintenance

Expert Systems

- Difficulties in realizing the system
  - Development of easy ways to get from the program specification to the automatic or semiautomatic generation of diagnostic rules
  - Finding articulate self-analytical experts who know *why* they do things
  - Developing efficient techniques for extracting what they know and distilling it into rule bases

Expert Systems

- Most powerful contribution would be the help to the inexperienced programmer, by providing the experience of the best programmers

Automatic Programming

- Generation of a program for solving a problem from a statement of the problem specifications
- Parnas thinks that this is just an euphemism for programming with a higher-level language
- In most cases solution method and not the problem specification has to be given
- Favorable properties of these applications
  - Problems are readily characterized by relatively few parameters
  - There are many known methods of solution to provide a library of alternatives
  - Extensive analysis has led to explicit rules for selecting solution techniques, given problem parameters

Graphical Programming

- Application of computer graphics to software design
  - Considering program as flowcharts and providing facilities to construct them
- Brooks suggest that this has nothing to offer for gains in software productivity because
  - Flow chart is a poor abstraction of software structure
  - Software is difficult to visualize

Program Verification

- Large part of total effort spent in testing and fixing of bugs
- Gaining on productivity here will provide sufficient improvement in overall productivity
- Brooks thinks that effort cannot be reduced here
- A perfect program verification can only establish that a program meets its specification

Program Verification

- The hardest part of the software task is arriving at a complete and consistent specification
- Much of the essence of building a program is in fact debugging of the specification

Environment and tools

- Major accidental difficulties have already been addressed
  - E.g. File formats, generalized tools, editors etc.
- Use of integrated database systems to keep track of all details for use by the programmers
- Further improvements will lead to only marginal increase in overall productivity

Workstations

- Speed and memory is not a concern nowadays
- Enhancements in these will not lead to significant increase in productivity

Promising attacks on conceptual essence

- Technological attacks on the accidents of the software process are fundamentally limited by the productivity equation

  Time of Task = Σ Frequency$_i$ x Time$_i$
- Conceptual components of the task are taking most of the time
  - Reducing the time on the *expression* of these concepts would not improve productivity much

Buy versus Build

- Radical solution for constructing software is not to construct it at all
- Becoming easier as more and more vendors are offering software products for variety of applications
- Many software tools and environments can be bought off-the-shelf
- It is much cheaper to buy than to build afresh any such product
  - Also better maintained and documented than homegrown software

Buy versus Build

- Mass market for such software reduces the cost
  - Sharing of a software by *n* users effectively multiplies the productivity of its developers by *n*
- Key issue : applicability
  - Can the software be used for the task
- Earlier many users would be reluctant to use these off-the-shelf packages, whereas now they are quite popular

Buy versus Build

- This is primarily because of the tremendous advances in hardware
- Earlier, the cost of the hardware was huge, so that the cost of developing a customized solution was too small as compared to the cost of the hardware
- Now the cost of hardware has reduced so much that customized solutions would prove to be too expensive
- Of course, there are exceptions to above

Requirements refinement and rapid prototyping

- Hardest part of building a software is deciding *precisely,* what to build; establishing the detailed technical requirements
- This is because the clients do not clearly know the product that they want
- Extensive iteration required between the clients and the designers to arrive at a final system definition

Requirements refinement and rapid prototyping

- Further, it is almost impossible for a client to fully specify a system without having used a version of the product
- Rapid prototyping would be very useful for this as a part of the iterative specification of requirements
- Prototype software systems simulates the important interfaces and the main functions of the intended system

Incremental development

- Conceptual concepts and structures are too complex to be built without faults
- Harlan Mills proposed that a software system should be *grown* by incremental development, and not built
- System should be made to run with dummy subprograms and then subprograms being developed incrementally
- This necessitates a top-down design
- Brooks says that the results of this approach is very encouraging

Great Designers

- Software development is *essentially* a creative process
- Great designs are different from simply good designs
- Great designs are faster, simpler, smaller, cleaner and produced with smaller effort
- Usually great designs have been produced by one or few persons, great designers
  - E.g. Unix, APL etc.

Great Designers

- Any software organization should identify and grow great designers
- Brooks thinks that usually companies ignore the task of finding and developing great designers
- Brooks suggests some of the methods by which a great designer can be developed in a company
  - Identifying top designers as early as possible
  - Assigning a career mentor to be responsible for development of the prospect
  - Devising and maintaining a career development plan for each prospect
  - Providing opportunities for growing designers to interact with and stimulate each other