No silver Bullet: Essence and Accidents of Software Engineering.
Frederik Brooks - 1987

Discussed by Pablo Navarro.

What is a silver bullet?

Orders of magnitude

- Orders of magnitude are used to make approximate comparisons.
- If numbers differ by one order of magnitude, x is about ten times different in quantity than y. If values differ by two orders of magnitude, they differ by a factor of about 100.
- Two numbers of the same order of magnitude have roughly the same scale: the larger value is less than ten times the smaller value.
- Thus, 2300 is one **order of magnitude** larger than 230, which in turn is is one **order of magnitude** larger than 23.

What is our werewolf?

- Software costs
- Managers and scientists have been looking for a silver bullet to reduce software costs by an **order of magnitude.**
- Where does this idea of a silver bullet come from?

Software costs

Computer hardware has silver bullet like performance.

- Computer hardware consistently gets better performance and lower prices every year.
- Computer hardware has seen six **orders of magnitude** in performance price gain in 30 years.
- It is not that the software industry progress is slow.
- Computer hardware progress is very fast, no other technology since the beginning of civilization has seen this kind of progress.

We have no silver bullet.

- The author believes that we should not expect anything close to a silver bullet in the software industry.
- The absence of a silver bullet in the software industry is a direct consequence of the intrinsic nature of software engineering.

Building software is difficult.

Difficulties can be separated into two main categories:

- Essential difficulties
  - Essential part or property of an object.
- Accidental difficulties
  - Parts or properties of an object that it happens to have but that it could lack.

Example: Running a marathon is hard.

- Accidental difficulties:
  - Improper shoes or clothing.
  - Inadequate diet.
  - Bad training.
  - Illness.
  - Extreme weather during the race.
  - Many more.

- All of these can be removed and gain performance improvements, some of those gains could be an order of magnitude improvement.

Removing accidental difficulties in a marathon.

Running with an old diving suit: 5 days 8 hours

Running with good equipment : 4 hours 19 minutes avg.

Example: Running a marathon is hard.

- Essential difficulties
  - 42.1 km is a long distance for a human to run.
    - So far there's not much to be done about this.
- The essential difficulties are very hard to remove because they are an essential part of the problem.
- Like a well prepared marathon runner, software engineering has been working on removing the accidental difficulties.

Past Breakthroughs Solved Accidental Difficulties.

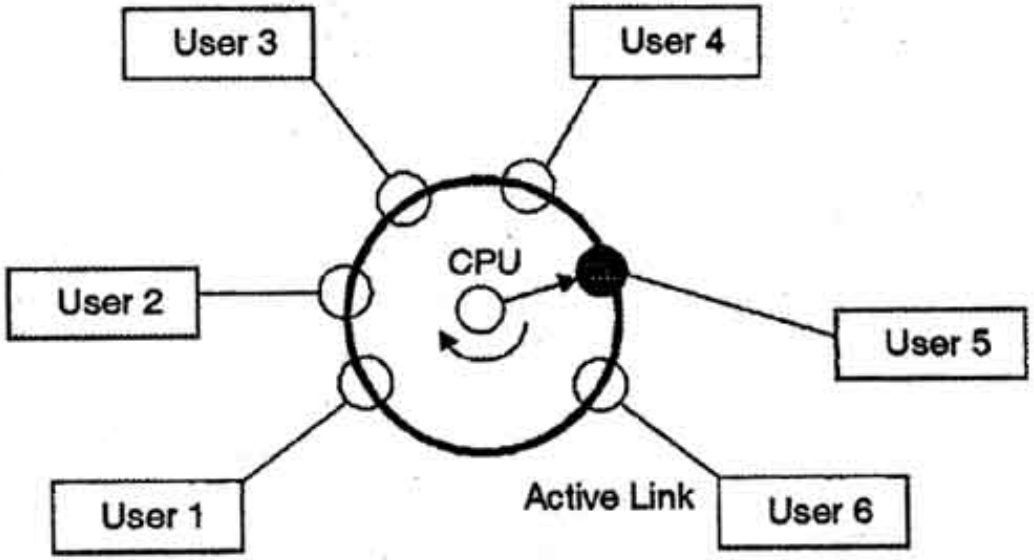
- Software engineering research teams have already achieved excellent results in reducing the accidental difficulties that increase the cost of producing new software.
- The author believes that these improvements have natural limits that prevents them from becoming silver bullet solutions.

Past Breakthroughs: High level languages.

- Liberate the developer of the low level computer instructions (Registers, buffers, interruptions, clock cycles, cylinders).
- The developer can think in higher level structures like data types, operations or sequences without thinking in the low level concepts.
- It can create a tool-mastery burden that increases, not reduces the size of the intellectual task of the software construction.
- The most powerful improvement for software productivity.
- Observers credit an increase of at least a factor of 5 in productivity.

Past Breakthroughs: Time sharing.

- Multiple users sharing time slots for access to a CPU.
- Allows to maintain a mental overview of complexity by having faster responses from the CPU regarding compilation and execution.
- It is now called multitasking.
- The effects will be less and less noticeable as the response time approaches zero.
- Not as good improvement as with the higher level languages.



Past Breakthroughs: Unified programming environments

- Provide hierarchical organization and syntactical errors detection (accidental difficulties).
- Integrated programming knowledge databases can only improve the process marginally.
- Most of the possible improvements have already been gained by the current state of the art tools.

Hopes for the silver bullet.

- The author lists some technology improvements that some claim that could provide silver bullet like increase in cost efficiency for software engineering.

Past Breakthroughs: Object oriented programming.

- The author knew that OOP could improve productivity and reduce costs but the removed difficulty would only be accidental.
- Object oriented programming and Ada obtained non order of magnitude gains.
- They could have gained an order of magnitude if the accidental complexity removed was 9/10 of the actual code, the author believed that was not the case and he was right.

Artificial Intelligence.

- Some believed that speech recognition could be used to improve programming.
- The author believes that the hard part of defining software is deciding **what one wants to say about it**, not actually saying it. Because of this the improvements can not be very large.

Expert systems.

- Is a program that contains a generalized inference engine and a rule base, takes input data and assumptions, explores the inferences derivable from the rule base yields conclusion and advice.
- Requires knowledge from experts distilled in knowledge based rules.
  - It is very hard to obtain a meaningful set of rules.
- The most likely contribution from these experts systems would be help for the novice programmer in form of some automated checklists.
  - This is already present in many programming environments with results that are not of an order of magnitude.

Graphical programming.

- The solutions will all be too similar to flow charts that the author deems to be not very useful.
- Software is complex and multidimensional that is not easy to fully represent in just 2 or 3 dimensions.
- The size of the current screens is very small (not true anymore).

Program verification.

- Claims that the cost of verification is very high so it’s very unlikely to see cost savings due to the use of verification.
  - The author refers to **formal verification** (Mathematical proof) not dynamic verification like unit testing or integration testing.
- Verification doesn’t imply error free programs since verification can be faulty as well.
- Verification can only ensure that the program fulfills an specification but the hardest part of software is getting that **specification right**.

Workstations.

- The Moore’s law was providing faster hardware at a predictable rate.
- The improvement in the workstations of the speed of the workstations used by the development teams would remove accidental difficulty of the waiting time for running and compiling only small gains can be expected.
- More of the time of the developers would be used thinking about designing the solution.

Essential difficulties.

- The next slides will summarize the essential difficulties of Software engineering.
- Even when the previous list of accidental difficulties had changes with the passing of years, the essential difficulties remain the same and it will remain unchanged for the foreseeable future.

Essential difficulties: Complexity.

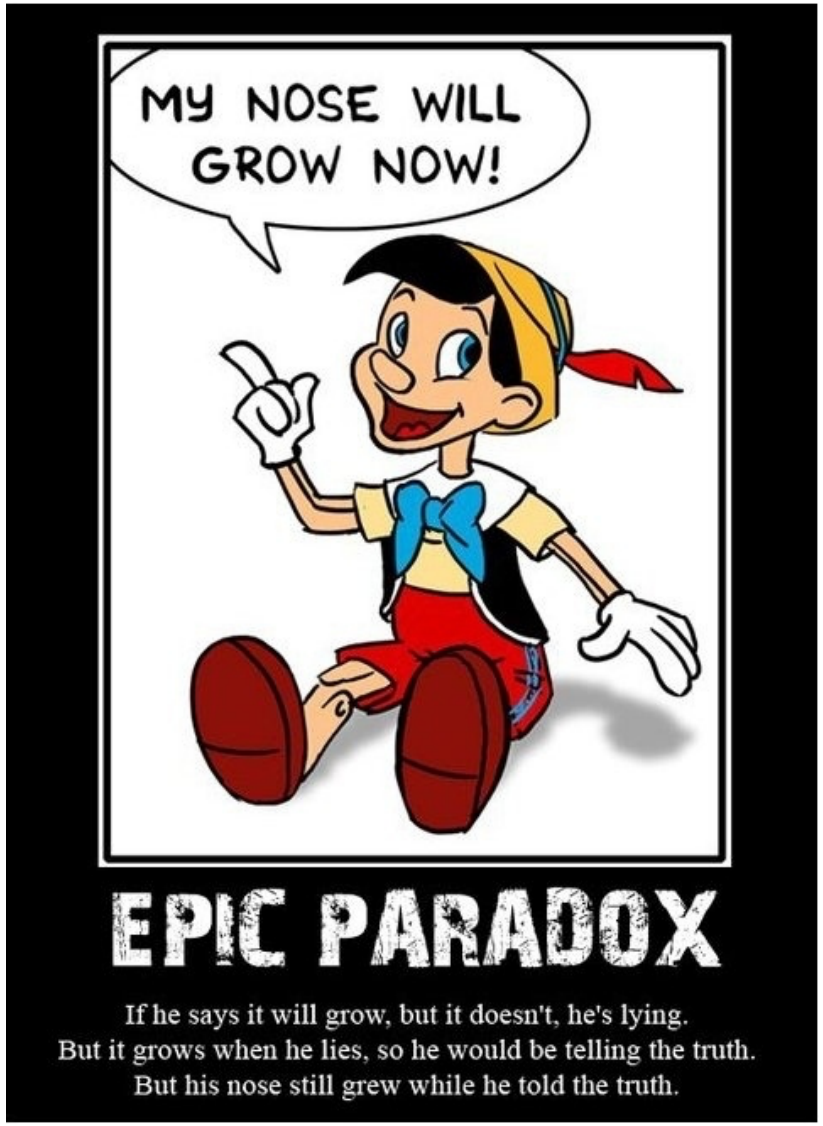
- Digital computers are very complex as they have a very large number of states.
  - Software tends to be much more complex and have **many orders of magnitude** more states that digital computers.
- Scaling up software is not just repetition of the same elements, it's an increase in the type of elements and also an increase in the relationships between elements, this increase is **nonlinear**.
  - It is not like making a building.
- The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often **abstract away its essence**.
  - There is no way to take the complexity out of the software.

Essential difficulties: Conformity.

- Software is always pressured to conform to **external interfaces**. (Policies, processes, standards, laws, personal wishes).
- In many cases, the software must conform because it is the most **recent arrival on the scene**.
  - Software is perceived as the most conformable thing.
- Most of the times the complexity is **arbitrary**, coming directly from many people not by natural laws like physics.
  - This makes software deal with inconsistencies and contradictions (catch 22).

Essential difficulties : Conformity.

- Software developers have to conform with things like this.

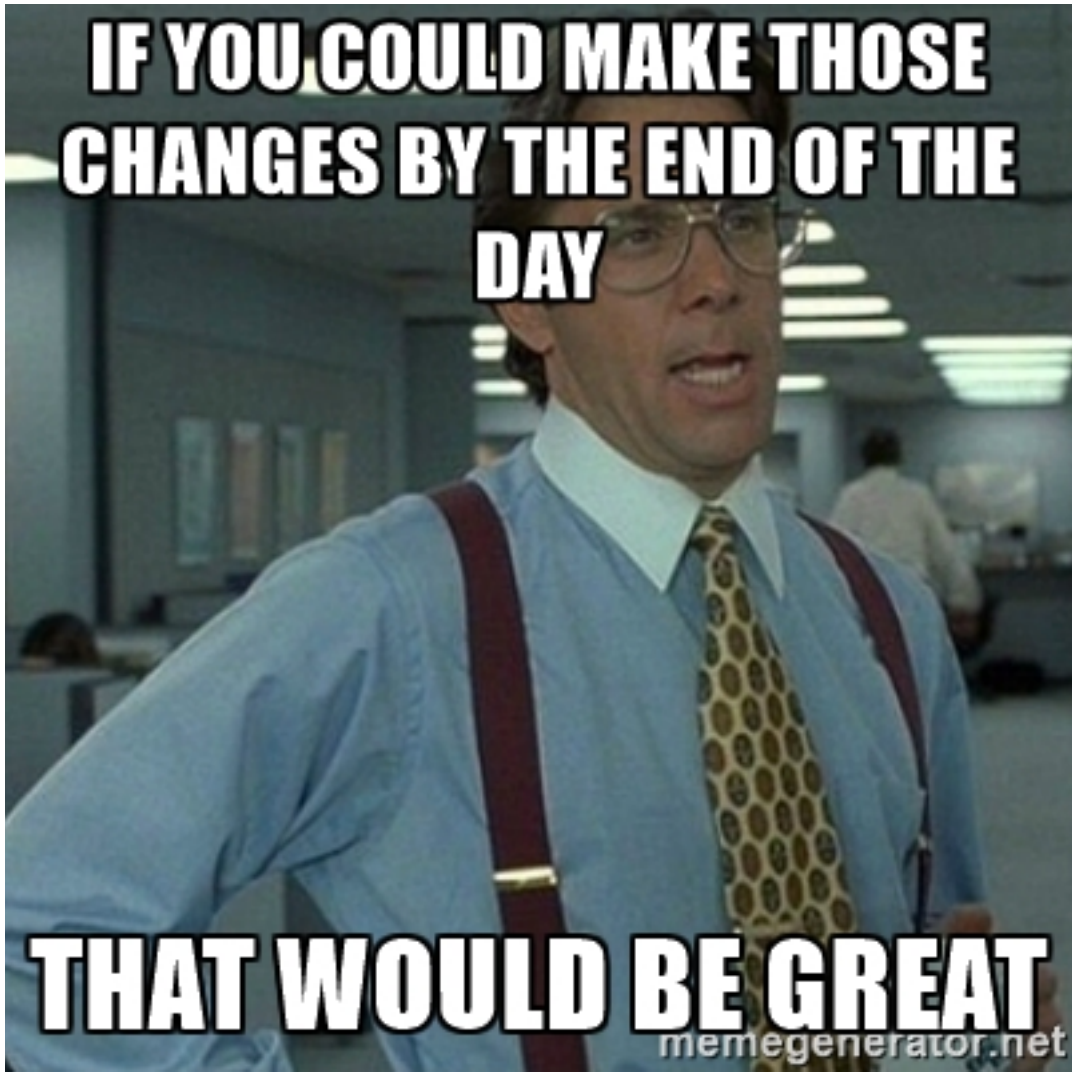


Essential difficulties: Changeability.

- Software is pure thought-stuff **infinitely malleable** always pressured to change.
  - All things are pressured to change, but manufactured things are rarely changed after manufacture because of costs and convenience.
- Software is perceived as **easy** to change.
- Software is the representation of its own **function** and function is always pressured to change.

Essential difficulties: Changeability.

- Software changes can sometimes be handled like a trivial task.



Essential difficulties: Invisibility.

- Software is invisible and unvisualizable, it is not inherently embedded in space. Hence, it has no ready geometric representation.
- To have a full software representation it is necessary to have many kinds of diagrams linked and superimposed between them.
- Each diagram deals with a particular abstraction of the software, leaving some of its complexities out, because of this there is no definitive graphical representation of software.
- The human mind works very well by visualizing things, but this is not entirely possible in software.

Focusing on the essence of difficulties.

- The author believes that in the future the most important gains can be obtained by attacking the essential difficulties of software development.
- Most of the time spent developing software is now spent on dealing with the essential difficulties.
  - Thus the really important gains in cost and productivity can only come from focusing on the essential difficulties.
- He mentions the most promising attacks on essential complexity.

Buy versus build

- Buy software don’t build it.
- The mass pc market has broadened the market for software, now building custom software is not the only option.
  - Spreadsheets are tipped to be very productive software alternatives by the author and he was right.
- Selling the software to more than one client automatically multiplies productivity and divides costs.
- This also attacks the conformity difficulty with many users opting to conform to the software interfaces, instead of the other way around.
  - This is specially true with Tax or Point of Sale software that is sold in mass.

Incremental development

- The hardest and most critical part of building software is deciding **what to build**.
- Clients don’t know well enough what they want and have almost never **thought of the problem in the detail** necessary for specification.
- It is almost impossible for a client, even working with a software engineer, to specify completely, precisely, and correctly the exact requirements of a modern software product before trying some **versions of the product.**

Incremental development

- The assumption that a system can be satisfactory specified, designed, implemented, tested and installed is wrong (Waterfall model).
  - The software we build today is too complex to be specified in advance and too complicated to be built with no errors.
- Rapid prototyping can help the client and developers to understand and define the problem.
- Iterative development: building, testing and delivering to the end user in many iterations attacks the essential difficulties of complexity and visibility.

Great designers

- Great designers will have great productivity.
  - The difference between great designers and average designers in productivity is close to an order of magnitude.
- Teaching a better curricula with the best practices is necessary but will not necessarily foster great designers.
- Great designers should be identified quickly, fostered and mentored by some other experienced great designers.
- Great designers are born not made.

Still no silver bullet

- This publication is still discussed in the present days and people still believe that finding a silver bullet for software costs is still not a possibility.
- Most of the claims that are made by the author are still accurate in the present day.
- The biggest gains in software productivity should be pursued by attacking the essential difficulties.
- Silver bullets are very rare to find in almost all industries.
  - Some simple examples for silver bullets through history are: the printing press for book making and photography for portrait making.

References

- Brooks, F. *No silver bullet*. April, 1987.
- Esther Derby, http://www.estherderby.com/2011/03/still-no-silver-bullets.html
- Adrian Colyer, https://blog.acolyer.org/2016/09/06/no-silver-bullet-essence-and-accident-in-software-engineering/
- Dinesh Thakur, http://ecomputernotes.com/fundamental/disk-operating-system/time-sharing-operating-system
- Teresa Robertson, Philip Atkins, http://plato.stanford.edu/entries/essential-accidental/
- https://en.wikipedia.org/wiki/Software_verification
- https://en.wikipedia.org/wiki/Order_of_magnitude

Questions

- Can you think of some other accidental difficulties that software developers face?
- Can you think of a novel approach to attack the essential difficulties of software developing?
- Are there parts of the paper that you think are not true or applicable for the present times ?
- Comments in general.