No Silver Bullet
Essence and Accidents of Software Engineering
Frederick P. Brooks, Jr.

Software projects are similar to werewolves since they may unexpectedly become a monster of missed schedules and budgets, and unsatisfying products; there is no silver bullets to solve this problem in a magical way since there are essential difficulties.

1 Does it have to be hard?

The difficulties of software may be divided into essence, the ones inherent in the nature of software, i.e. complex conceptual constructs, and accidents that are not inherent, i.e. representing the constructs in programming language.

The essence of software is a construct of interlocking concepts: data sets, relationship among data items, algorithms, and invocations of functions. This essence is abstract in that the conceptual construct is the same under many different representations, but nonetheless highly precise and richly detailed. The specification, design, and testing of the construct, not the labor of representing it and testing fidelity of the representation are the hard parts.

The very nature of software eliminates the possibility that any magical advancement happens. One cannot expect software technologies keep pace with hardware since the latter is nothing less than an exception due to the transformation of computer manufacture from an assembly industry into a process industry.

The hardness of the essence of software may be embodied in the following aspects: complexity, conformity, changeability, and invisibility.

1.1 Complexity

Software is much more complex than any other human constructs in the sense that software may contain much more elements than any other thing people build, and with a linear increase of the
number of elements, the interaction between them however increases in some nonlinear fashion.

The complexity of software is essential, and leads to many of the classic problems of developing software products. We cannot abstract away its complexity without abstracting away its essence.

1.2 Conformity

Software involves arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which its interfaces must conform, the number of which is still swelling; this complexity cannot be simplified out by any redesign of the software alone.

1.3 Changeability

The software product is embodied in a cultural matrix of applications, users, laws, and machine vehicles. The changes of any of these aspects inexorably force software to change.

1.4 Invisibility

Software is invisible since it is not inherently embedded in space; moreover, we cannot always visualize it in hierarchical graphs, which fit the way people think.

2 Past breakthroughs solved accidental difficulties

2.1 High-level languages

The development of high-level languages is credited with at least a factor of five in productivity, and with concomitant gains in reliability, simplicity, and comprehensibility. It, however, eliminates only the complexity related to lower level constructs that are not inherent in software.

The level of our thinking about data structures, data types, and operations is steadily rising, but at an ever decreasing rate, and approaches closer and closer to the sophistication of users.

2.2 Time-sharing

Time-sharing eliminates the slow turnaround of batch programming, and helps programmers to keep fresh in mind the essence of a complex system, but only the accidental difficulties are attacked
here. What’s more, the benefit of time-sharing is to be boundary due to the human threshold of noticeability.

2.3 Unified programming environments

Unified programming environments enable related individual tools to work together in an automatic manner and thus free programmers from the burden of various manual operations, which are obviously not inherent in software products.

3 Hopes for the silver

3.1 Ada and other high-level language advances

Ada, one of the most touted recent development, not only reflects evolutionary improvements in language concepts, but indeed embodies features to encourage modern design and modularization. Nevertheless, it is just another high-level language and will not prove to be the silver bullet.

3.2 Object-oriented programming

Object-oriented programming is considered the most promising silver bullet candidate. Its two important concepts, abstract data types and hierarchical types, represent real advances in the art of building software. Nevertheless, they remove only accidental difficulties from the expression of the design, rather than the design itself.

3.3 Artificial Intelligence

Although artificial intelligence is credited by many people with much hope for revolutionary breakthrough in software productivity and quality, it unfortunately involves even terminology chaos. There is either no support for programming practices in sight from AI techniques used for speech and image recognition in the sense that the hard thing about building software is deciding what one wants to say, not saying it.
3.4 Expert Systems

An expert system 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 conclusions and advice, and offers to explain its results by retracting its reasoning for the user.

The advantages of expert systems over programmed algorithms are as follows:

- Inference-engine technology is developed in an application-independent way.
- The application-specific materials are encoded in the rule base in a uniform fashion, which regularizes the complexity of the application itself.

Such systems may be built to help inexperienced programmers based on the wisdom of experts by suggesting interface rules, advising on testing strategies, and offering optimization hints. To achieve this, there however exist many difficulties:

- Diagnostics: How to generate automatically the diagnostic rules from program-structure specification
- Prescription: How to extract expertise and distill it into rule bases

3.5 “Automatic” programming

Automatic programming is actually a euphemism for programming with a higher-level language than was presently available to programmers so that solutions could be given more easily. Although this technique has indeed succeeded in some special cases, it is hard to be generalized for the ordinary software systems.

3.6 Graphical programming

Computer graphics, which has been applied successfully in other fields, could not be a silver bullet for software design based on the following observations:

- The flowchart, once considered as the ideal program-design medium, is actually a very poor abstraction of software structure.
- The screens of today are too small to show detailed software diagram.
More fundamentally, software is very difficult to visualize.

3.7 Program verification

Although program verification seems promising to avoid immense effort upon implementing and testing by eliminating errors in the design phase, there is no magic because:

- Verifications are so much work that only a few programs have been verified.
- Verification cannot eliminate errors totally since mathematical proofs can be faulty too.
- Specification, the baseline of verification, is usually incomplete and inconsistent.

3.8 Workstations

More powerful computers surely facilitate software development, but nowadays time of thinking, instead of waiting for computers’ response, is the dominant activity of programmers. Magical enhancement thus cannot be expected.

4 Promising attacks on the conceptual essence

Since the conceptual components of the software development task are now taking most of the time, we must consider those attacks that address the essence of the software problem. Fortunately, some attacks are very promising.

4.1 Buy versus build

The most radical possible solution for constructing software is not to construct it at all, which is already practised by many organizations by purchasing off-the-shelf products. This is made possible by the following reasons:

- The personal-computer revolution has created many mass markets for software, which, together with zero replication cost of software, stirred the motivation for software companies to produce more and better software products.
- Applicability of software is enhanced with the generalization of software tools and the decreasing hardware/software cost ratio.
4.2 Requirements refinement and rapid prototyping

The hardest part of building a software system is deciding detailed technical requirements. Unfortunately even the clients themselves do not exactly know what they want, so iterative extraction and refinement of product requirements are necessary.

Furthermore, it is really impossible for a client to specify completely, precisely, and correctly the exact requirements of a modern software product before trying some versions of the product. Rapid prototyping may give clients a first-hand feel of what the product will be and a check for consistency and usability.

4.3 Incremental development

The conceptual software structures we construct today are too complicated to be specified accurately in advance, and too complex to be built faultlessly. A different approach must be taken.

Enlightened by the development of human brain, incremental development, or growing software, has the following advantages over building straightforward:

- The approach necessitates top-down design, thus allowing easy backtracking and detecting fundamental defects as early as possible.

- An always working system stirs enthusiasm.

4.4 Greater designers

People are the key factor for solving problems. Although steps have been taken to raise the level of our practice from poor to good, the gain is still limited; great designers, however, create state-of-the-art works, the benefits of which are order-of-magnitude. More should be done to grow design masters.
5 Questions

1. What is process industry?

2. I think the specification, design, and testing of conceptual constructs are also presentations, although they are in a higher level other than the detailed implementation.

3. Why the complexity of software is an essential property? It is possible to attack the complexity at a variety of levels. We may split a system into several subsystems, whose details are invisible to the outside environment, which is actually information hiding.

4. The development of software may be conceived as implementing a black box, whose input and output are respectively the capabilities of computer (+, -, ×, ÷ at a lower level; or loop, condition, or nest structures at a higher level), and the functions of software people expect. As for mathematics and physics, the same thing holds. People in these two fields try to figure out what the black boxes are between small systems and large ones so that the latter ones show the behaviors we see in the real world. Why do they have different fates? I think mathematics and physics always make progress in a stepwise fashion since they always try to solve problems that immediately follows. Things are different for software development since the input and output are clear but there are still a big gap between them. Why do mathematics and physical sciences seem make great strides? How to measure those strides if they are? One thing needs to mention is that even if only one advancement is achieved, the possible combinations of this advancement with other known ones nonetheless allot, which is actually what we see, and is looked on as the reason we say ”great strides”. Do we consider the establishment of genome of a variety of species one advancement or a lot?

5. Although the complexity of possible interaction between system elements increases in a non-linear fashion with the increase of system size, it is still possible to limit interaction to a local range so that the system may be constructed hierarchically and the details of subsystems be abstracted away in the higher levels, which is the case for physical science. Physical science however deals with objects that exist in a 3D environment.

6. Brooks said that much of the complexity that the software engineer must master is arbitrary complexity, forced by the many human institutions and systems to which his interfaces must
conform. First, complexity is arbitrary? Think about the fate of the "goto" statement in programming languages. Maybe the high complexity of present software is just because we haven’t found a better way to describe it. Second, the conformity is actually limited since the software we proposed are not universal software, and are always bounded in some extent.