Nathan Quirk  
Article Review 1  
“No Silver Bullet” – Frederick P. Brooks, Jr.

In “No Silver Bullet,” author Frederick Brooks explains why the process of engineering software will never be an easy one. Written in 1986 at a time when computer hardware saw (and continues to see) exponential increases in speed and performance along with similar decreases in cost, Brooks illustrates why the very nature of software programming put limits on growth.

Brooks identifies the essential difficulty in building software to be “the specification, design, and testing,” not the logic or programming. This, he says, means that software engineering will always be difficult. Brooks identifies four areas that make efforts to streamline this process an uphill battle: *complexity, conformity, changeability, and invisibility*.

By complexity, Brooks is referring to the complex nature of software systems compared to other engineering constructs. When software projects expand, their constituent parts are different, and each brings the potential for errors as the number of states within the system increase exponentially. This complexity gives rise to the classic issues surrounding development such as bugs in the code. Additionally, because accounting for all the possible states of a program is difficult, the product can be unreliable. Because programs can behave in unexpected ways, using the programs can be difficult and thus unwieldy, and adapting the program to new use cases often creates side effects that dictate the need to go back to the drawing board.

Brooks also asserts that software is often the most malleable part of any given project, and as such it is subjected to the arbitrary whims of the people and processes of any particular situation. The need for conformity, as Brooks call it, forces the software developer to jump through many artificial hoops in addition to the other challenges they face.

In “Silver Bullet,” changeability refers to the inevitable need for any successful software implementation to be changed to fit growing use cases or updates in hardware technology. In this sense, good software engineers are their own worst enemy, because a product that becomes widely used is likely to be tested against edge cases and demand increases for the scope of the software to widen. Similarly, if software is used long enough, changes to the hardware landscape dictate that new versions of the software be released.

Another intrinsic aspect of software is that it defies attempts to be represented visually. Here Brooks points to blueprints that may aid a builder, or the molecular models that can help a chemist. Software engineers have no such ability. Elements of code may exist simultaneously on different planes of space, with many layers needed to represent the myriad of connections, names, flows, and dependencies. Thus, it is not easy to take a step back and see where an omission or oversight has been made.

While some progress has been made, the author submits that these improvements have done nothing to mitigate the conceptual complexity of software design, only some of the difficulties that happen to accompany the building of software. These difficulties are costly and are worth addressing, but remain distinct from the fundamental and inalienable challenges at the heart of software engineering.

Some of the attempts to resolve these lesser hindrances include: high level languages, time-sharing, and unified programming environments. While Brooks doesn’t discount the help that these advances provide, he maintains that they have diminishing returns and don’t attack the complexities at the core of the issue.

In looking to the future, the author points to some up-and-coming advancements – some which have materialized in the decades since the article was written and some not. These include higher-level languages, object-oriented programming, artificial intelligence, expert systems (which we have today to some degree as part of our IDEs), program verification, and more powerful personal workstations.

The author *does* see some hope though. He first notes that it may be better to buy existing solutions rather than re-invent the wheel every time a piece of code is needed. (Brooks, who is 87 today, is almost certainly excited to see how the open-source movement has transformed thinking!) Brooks goes on to say that a paradigm shift is necessary in how customers and engineers interface. Specifically, that software development should be an iterative process and that rapid prototyping should bring the developers and clients to the table often to reflect and revise their goals and expectations. As Brooks puts it: “…it is really 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.” By *iterative*, the author suggests that good software should be grown, not built. In other words, rather than trying to build code as it will exist in the finished version of the product, build a minimum working model, than expand upon that, using ongoing input from client as a guide. Finally, he stresses the importance of identifying and nurturing talented designers.