Chapter 1

Understanding Software Engineering

In order to understand software engineering, we first need to look at the projects that were reported in the early software engineering literature. One feature is immediately striking—the absence of reports on commercial applications. Most case studies are of either large defense projects or small scientific projects. In either case, the projects typically involved severe hardware and software challenges that are not relevant to most modern projects.

A typical example is the SAFEGUARD Ballistic Missile Defense System, which was developed from 1969 through 1975.[3] "The development and deployment of the SAFEGUARD System entailed the development of one of the largest, most complex software systems ever undertaken." The project took 5,407 staff-years, starting with 188 staff years in 1969 and peaking at 1,261 staff-years in 1972. Overall productivity was 418 instructions per staff-year.

SAFEGUARD was a very large software engineering project that challenged the state of the art at the time. Computer hardware was specially developed for the project. Although the programming was done in low-level languages, the Code and Unit Test activities

3. Stephenson, W. E., "An analysis of the resources used in the SAFEGUARD system software development." In Donald J. Reefer, *Tutorial: Software Management,* IEEE Computer Society, 1981.

required less than 20% of the overall effort. System Engineering (requirements) and Design each consumed 20% of the effort, with the remainder (more than 40%) being accounted for by Integration Testing.

The Paradox of Software Engineering

In trying to understand software engineering, we need to keep two points in mind:

- Projects the size of SAFEGUARD are extremely rare.
- These very large projects (1,000-plus staff-years) helped to define software engineering.

Similarly, *The Mythical Man-Month*[4] by Fred Brooks was based on IBM's experiences when developing the OS/360 operating system. Even though Brooks wrote about the fact that *large programming projects suffer management problems that are different from the problems encountered by small ones due to the division of labor*, his book is nevertheless still used to support the ideas behind software engineering.

These really large projects are really *systems engineering* projects. They are combined hardware and software projects in which the hardware is being developed in conjunction with the software. A defining characteristic of this type of project is that *initially the software developers have to wait for the hardware, and then by the end of the project the hardware people are waiting for the software.* Software engineering grew up out of this paradox.

What Did Developers Do While Waiting for the Hardware?

Early in the typical software engineering project, there was plenty of time. The hardware was still being invented or designed, so the software people had plenty of time to investigate the requirements and produce detailed design specifications for the software. There

4. Brooks, Frederick P., *The Mythical Man-Month,* 20th Anniversary Edition, Addison-Wesley, 1995.

was no point in starting to write the code early, because the programmers lacked hardware on which to run the code (and in many early examples, the compilers and loaders for the code were not ready either). In some cases, the programming language wasn't even chosen until late in the project. So, even if some design specifications were complete, it was pointless to start coding early.

In that context, it made sense to define a rigorous requirements process with the goal of producing a detailed requirements specification that could be reviewed and signed off. Once the requirements were complete, this documentation could be handed off to a design team, which could then produce an exquisitely detailed design specification. Detailed design reviews were a natural part of this process, as there was plenty of time to get the design right while waiting for the development of the hardware to advance to the point where an engineering prototype could be made available to the software team.

How Did Developers Speed Up Software Delivery Once the Hardware Became Available?

The short answer is, "Throw lots of bodies at the problem." This was the "human wave" approach that Steven Levy described and that can be seen in the manpower figures reported from the SAFEGUARD project. As soon as the hardware became available, it made sense to start converting the detailed design specifications into code. For optimum efficiency, the code was reviewed to ensure that it conformed to the detailed design specification, because any deviation could lead to integration problems downstream.

Lots of people were needed at this stage because the project was waiting for the software to be written and tested. So, the faster the designs could be converted into tested code, the better. Early software engineering projects tended to use lots of programmers, but later on the emphasis shifted toward the automatic generation of code from the designs through the use of CASE tools. This shift occurred because project teams faced many problems in making the overall system work after it had been coded. If the code could be generated from the design specifications, then projects would be completed faster, and there would be fewer problems during integration.

Implications for the Development Process

Software engineering projects require lots of documentation. During the course of a project, three different skill sets are needed:

- Analysts to document the requirements
- Designers to create the design specifications
- Programmers to write the code

At every stage, the authors of each document must add extra detail because they do not know who will subsequently be reading the document. Without being able to assume a certain, common background knowledge, the only safe course is to add every bit of detail and cross-referencing that the author knows. The reviewers must then go through the document to confirm that it is complete and unambiguous.

Complete documentation brings with it another challenge: Namely, team members must ensure that the documents remain consistent in the face of changes in requirements and design changes made during implementation. Software engineering projects tackle this challenge by making sure that there is complete traceability from requirements through to implemented code. This ensures that whenever a change must be made, all of the affected documents and components can be identified and updated.

This document-driven approach affects the way that the people on the project work together. Designers are reluctant to question the analysts, and the programmers may be encouraged not to question the design nor to suggest "improvements" to the design. Changes are very expensive with all of the documents, so they must be controlled.

A great way to control changes from the bottom is to define a project hierarchy that puts the analysts at the top, with the designers below them, and the programmers at the bottom of the heap. This structure is maintained by promoting good programmers to become designers and allowing good designers to undertake the analysts' role.

The Modern Definition of Software Engineering

Over the last 30 years, the software engineering community has followed the path of applying mechanical metaphors to the software development process. Software engineering is now an accepted academic subject and an active research field for universities. The focus for software engineering projects is on a defined, repeatable approach as exemplified by the IEEE definition:

> *Software engineering is the application of a systematic, disciplined, quantifiable approach to development, operation, and maintenance of software; that is, the application of engineering to software.*[5]

This systematic, disciplined, and quantifiable approach to software development has proved to be very effective at developing safety critical systems. The team that writes the software for the space shuttle, for example, used this approach and has managed to achieve an admirable defect rate.

> *The last three versions of the program—each 420,000 lines long—had just one error each. The last 11 versions of this software had a total of 17 errors. Commercial programs of equivalent complexity would have 5,000 errors.*[6]

In the process, however, other process constraints had to be relaxed.

> *Money is not the critical constraint: The group's $35 million per year budget is a trivial slice of the NASA pie, but on a dollars-per-line basis, it makes the group among the nation's most expensive software organizations.*[7]

This is an appropriate engineering trade-off. When lives are at stake, it makes sense to use whatever resources are needed to ensure that nothing goes wrong. But what about software development when the consequence of error is lower?

---

5. *IEEE Standard Computer Dictionary,* ISBN 1-55937-079-3, IEEE, 1990.

6. "They Write the Right Stuff," *Fast Company,* http://www.fastcompany.com/online/06/writestuff.html.

7. "They Write the Right Stuff."

Good Enough Software—Software Engineering for the Masses

For some software, rapid development of feature-rich applications is what matters. The idea is that users will put up with errors in programs because they have so many useful features that are unobtainable elsewhere. As Edward Yourdon[8] put it, "I'm going to deliver a system to you in six months that will have 5,000 bugs in it—and you're going to be *very* happy!"

Good enough software is a logical extension of the ideas of software engineering. It represents the engineering trade-off between resources, schedule, features, and defects. The space shuttle software is safety-critical, so it has to minimize defects, accepting the resulting schedule and resource demands. Commercial shrink-wrapped applications like word processors and Web browsers need lots of features that must be developed quickly. Resources are constrained by the need to make a profit, so the engineering trade-off is made to shrink the schedule by spending less time removing known defects. The idea is that for some kinds of known defects, it is not economic to take the time to remove them.

Is Software Engineering a Good Choice for Your Project?

Systems engineering projects that involve the development of new hardware and software are a natural fit for software engineering. Many defense and aerospace projects fit within this category. When I'm a passenger in a "fly by wire" aircraft, I want to know that a *systematic, disciplined, and quantifiable approach* was taken to the development and verification of the flight control software. After all, it would not be very comforting to know that the software "was developed by the lowest bidder."

If your organization develops large, shrink-wrapped consumer software applications and is good at making appropriate engineering trade-offs, you might be able to use the *good enough software*

8. Yourdon, Edward, *Rise and Resurrection of the American Programmer*, Prentice-Hall, 1996.

approach. The key to success with this type of software engineering is volume. You need to be selling millions of units in a competitive market where customers buy on the basis of reviews and marketing rather than on detailed, side-by-side comparisons of products.

In all other cases, you should be looking for alternatives to software engineering.