Three Patterns
that help explain the development of
Software Engineering

Mary Shaw
Computer Science Department
Carnegie Mellon University
Pittsburgh PA 15213 USA

March 1997

The term "software engineering" came to prominence when it was used as the name of a NATO workshop in 1968 [NaRan69]. It was used then to draw attention to software development problems. It was then, as to a large extent it remains now, a phrase of aspiration, not of description.

In the intervening years, the focus of the academic community (though not so much the industrial software development community) has shifted from simply writing programs to analyzing and reasoning about large distributed systems of software and data the come from diverse sources. Figure 1 lays out the highlights of these shifts.

Figure 1: Highlights of academic attention in software engineering

	1960 ± 5 Programming-any-which-way	*1970 ± 5 Programming-in-the-small*	*1980 ± 5 Programming-in-the-large*	*1990 ± 5 Programming-in-the-world*
Specifications	**Mnemonics, precise use of prose**	**Simple input-output specifications**	**Systems with complex specifications**	**Distributed systems with open-ended, evolving specs**
Design Emphasis	**Emphasis on small programs**	**Emphasis on algorithms**	**Emphasis on system structure, management**	**Emphasis on subsystem interactions**
Data	**Representing structure, symbolic information**	**Data structures and types**	**Long-lived databases**	**Data & computation independently created, come and go**
Control	**Elementary understanding of control flow**	**Programs execute once and terminate**	**Program systems execute continually**	**Suites of independent processes cooperate**

I see three simple patterns that have guided this development. Each of these provides partial explanations, but none is either comprehensive enough or rich enough to be in and of itself a full model.

(1) Evolution of engineering disciplines.

Technologies evolve from craft through commercial practice before they integrate scientific knowledge and become true engineering disciplines. Figure 2 illustrates this pattern. Software engineering has been following this pattern; it helps to explain the role of software process improvement. [Shaw90]

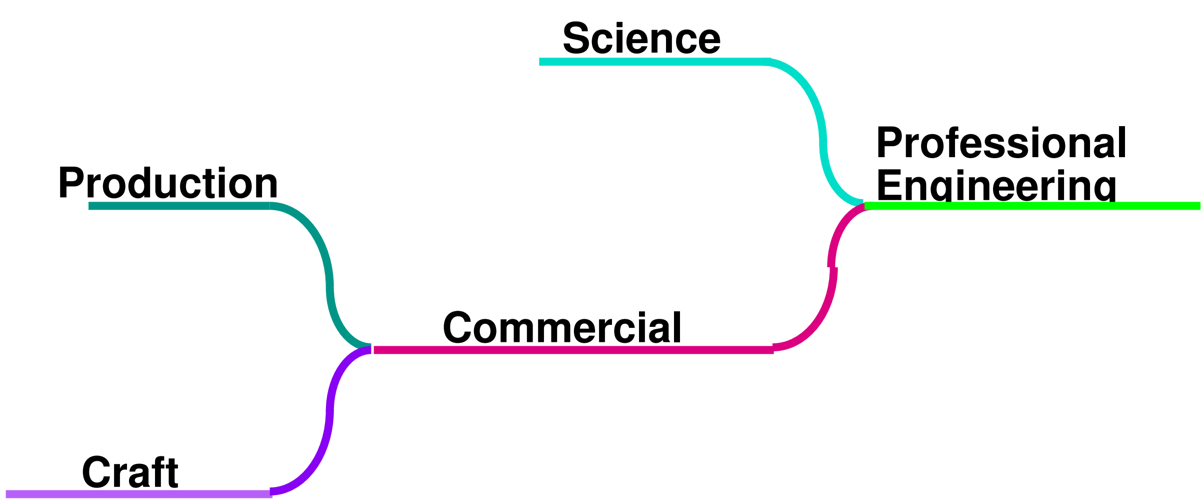
Exploitation of a technology begins with craftsmanship: practical problems are solved by talented amateurs and virtuosos, but no distinct professional group is dedicated to problems of this kind. Intuition and brute force are the primary problem-solving strategies. Progress is haphazard, and the transmission of knowledge is casual. Extravagant use of materials may be tolerated, and manufacture is often for personal or local use.

At some point, the products of the technology gain commercial significance, and economies of manufacture become an issue. At this point, the resources required for systematic commercial manufacture are defines, and the expertise to organize exploitation of the technology is introduced. Capital is needed to acquire raw materials or invest in manufacture long before sale, so financial skills become important. Scale increases over time, and skilled practitioners are needed for continuity and consistency. Pragmatically-derived procedures are replicated carefully without necessarily having knowledge of why they work. Management and economic strategies may assume as large a role as the development of the technology. Nevertheless, problems with the technology often stimulate the development of an associated science.

When the associated science is mature enough to yield operational results --that is, results that are cast in the form of solutions to practical problems, not as abstract theories -- an engineering discipline can emerge. This allows technological development to pass limits previously imposed by relying on intuition; progress frequently becomes dependent on science as a forcing function.

Software engineering is in the process of moving from the craft to the commercial stage. It has only achieved the stature of a mature engineering discipline in isolated cases.

Figure 2: Evolution of engineering disciplines (after [Finch51])



(2) Abstraction and its coupling to specifications

The granularity of our abstractions -- the intellectual size of a chunk we treat as atomic -- increases over time. Abstractions are supported by formal specifications, but formal specifications will be used in practice only to the extent that they provide clear payoff in the near term. [Shaw80]

This pattern can be seen in the development of data types and type theory. In the early 1960's, type declarations were added to programming languages. Initially they were little more than comments to remind the programmer of the underlying machine representation. As compilers became able to perform syntactic validity checks the type declarations became more meaningful, but "specification" meant little more than "procedure header" until late in the decade. The early 1970s brought early work on abstract data types and the associated observation that their checkable redundancy provided a methodological advantage because they gave early warning of problems. At this time the purpose of types in programming languages was to enable a compile-time check that ensured that the actual parameters presented to a procedure at runtime would be acceptable. Through the 1980s type systems became richer, stimulated by the introduction of inheritance mechanisms. At the same time, theoretical computer scientists began developing rich theories to fully explain types. Now we see partial fusion of types-in-languages and types-as-theory in functional languages with type inference. We see in this history that theoretical elaboration relied on extensive experience with the phenomena, while at the same time practicing programmers are willing to write down specifications only to the extent that they are rewarded with analysis than simplifies their overall task.

Figure 3: Language constructs and phases of software engineering development

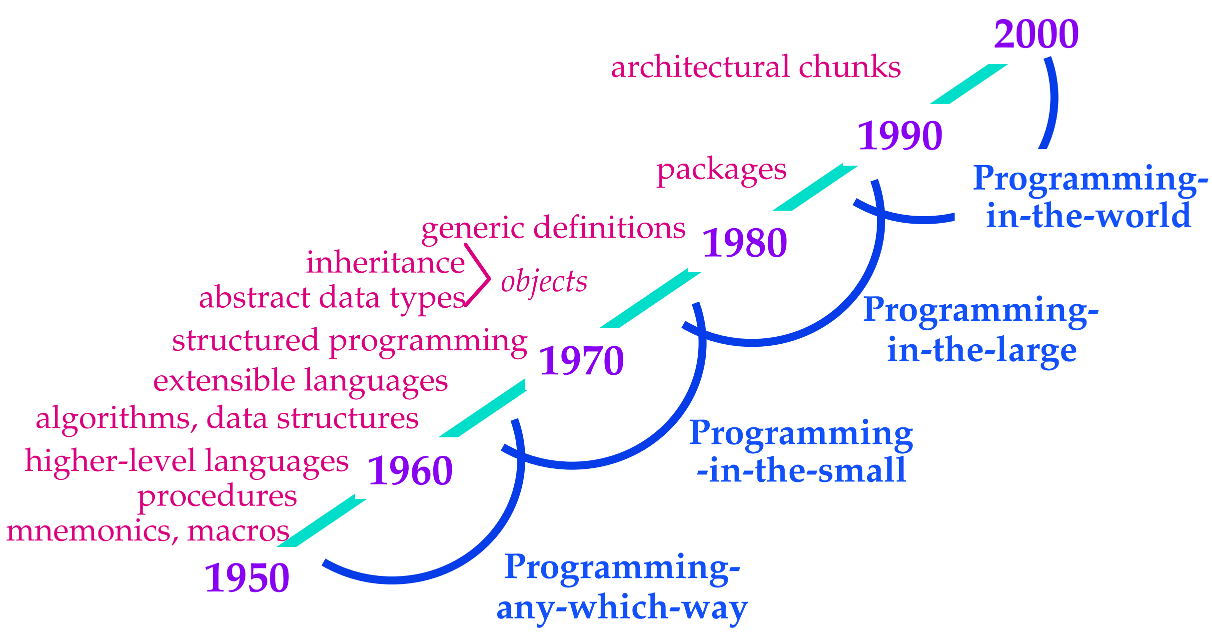
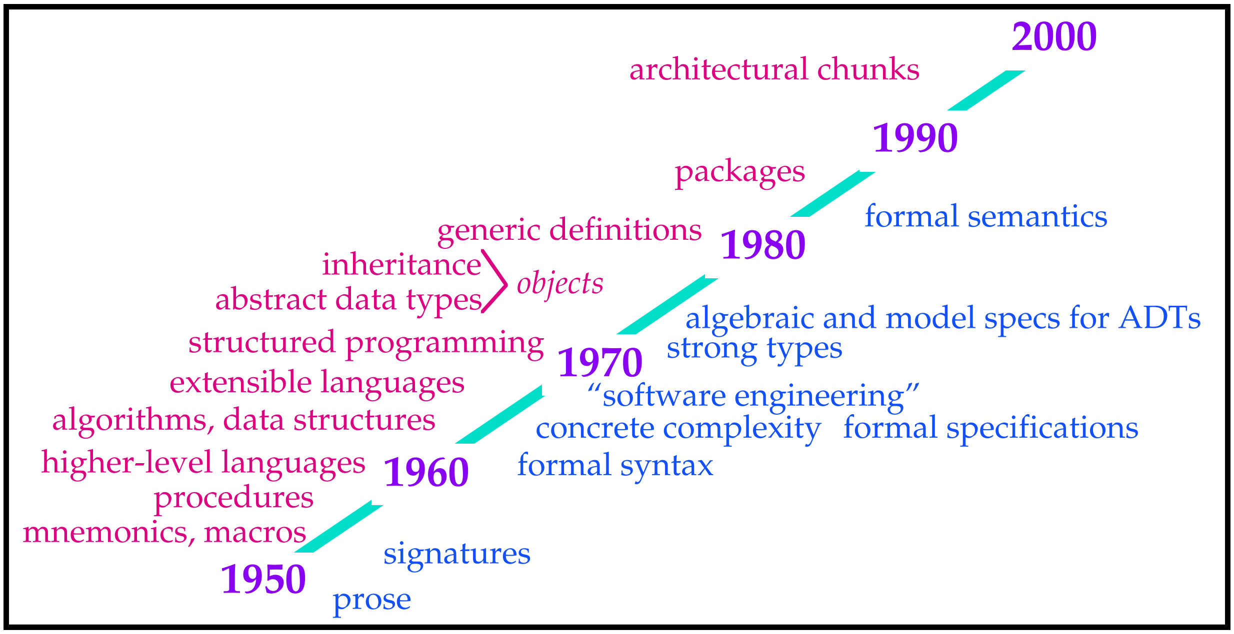


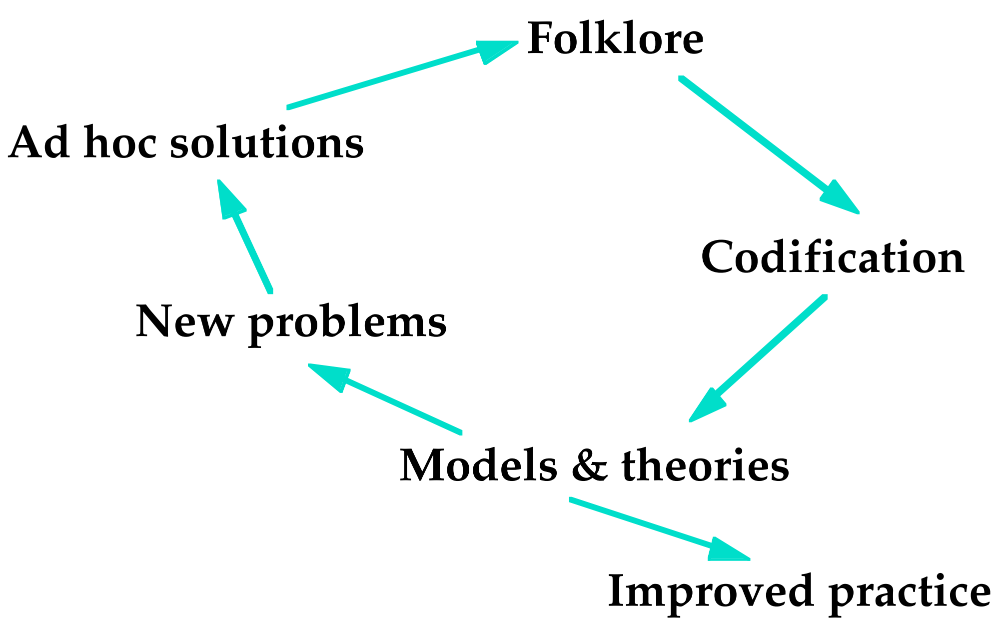
Figure 4: Coupled development of abstraction and specification



(3) Progressive codification

Specification techniques evolve in parallel with our understanding of the phenomena they specify. [ShGar95] We begin by solving problems any way we can manage. After some time we discover in the ad hoc solutions some things that usually work well. Those enter our folklore; as they become more systematic we codify them as heuristics and rules of procedure. Eventually the codification becomes crisp enough to support models and theories. These help to improve practice; they also allow us to address new problems that were previously unthinkable.

Figure 5: Cycle of progressive codification



Thus, as some aspect of software development comes to be better understood, more powerful specification mechanisms become available, and they yield better rewards for the specification effort invested. We can characterize some of the levels of specification power:

> Ad hoc:	implement any way you can
> Capture:	simply retain and explain a definition
> Construction:	explain how to build an instance from parts
> Composition:	explain how to compose parts and their specifications
> Selection:	guide designer's choices for design or implementation
> Verification:	determine whether an implementation matches specification
> Analysis:	determine the implications of the specification
> Automation:	construct an instance from an external specification

When describing, selecting, or designing a specification mechanism, either formal or informal, it is useful to be explicit about which level it supports. Failure to do so leads to mismatches between user expectations and specification power.

Brooks proposes recognizing three kinds of results, together with criteria for judging the quality of those results [Brooks88]:

findings	well-established scientific truths	truthfulness and rigor
observations	reports on actual phenomena	interestingness
rules-of-thumb	generalizations, signed by an author but perhaps not fully supported by data	usefulness
all three		freshness

Bibliography

[Brooks88] Frederick P. Brooks, Jr. Grasping Reality Through Illusion -- Interactive Graphics Serving Science. *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference*, May 1988, pp. 1-11.

[Finch51] James Kip Finch. Engineering and Western Civilization. McGraw-Hill 1951.

[NaRan69] Peter Naur and Brian Randell (eds). Software Engineering: report on a conference sponsored by the NATO Science Committee, Garmisch Germany 1968. NATO 1969.

[Shaw80] Mary Shaw. The Impact of Abstraction Concerns on Modern Programming Languages. *Proc IEEE*, September 1980, also *IEEE Software* Oct 1984.

[Shaw90] Mary Shaw. Prospects for an Engineering Discipline of Software. *IEEE Software*, November 1990.

[ShGar95] Mary Shaw and David Garlan. Formulations and Formalisms in Software Architecture. Computer Science Today (LNCS 1000), Jan van Leeuwen (ed), Springer-Verlag 1995.