Fifty Years of Progress in Software Engineering

L. B. S. Raccoon
raccoon@lbsr.com
PROG34.WPD - NOV 14, 1996

INTRODUCTION

In this paper, I describe a new outlook on the history of Software Engineering. I portray large-scale structures within Software Engineering to give a better understanding of the flow of history. I use these large-scale structures to reveal the steady, ongoing evolution of concepts, and show how they relate to the myriad whorls and eddies of change. I also have four smaller, more specific purposes in writing this paper.

First, I want to point out that old ideas do not die. In The Mythical Man-Month after 20 Years, Brooks claims "the Waterfall Model is Wrong." But if the Waterfall model were wrong, we would stop arguing over it. Though the Waterfall model may not describe the whole truth, it describes an interesting structure that occurs in many well-defined projects and it will continue to describe this truth for a long time to come. I expect the Waterfall model will live on for the next one hundred years and more.

Second, I want to show that the Chaos model, Chaos life cycle, Complexity Gap, and Chaos strategy are part of the natural evolution of Software Engineering. The Chaos model and strategy supersede, but do not contradict, the Waterfall and Spiral models, and the Stepwise Refinement strategy. They are more up to date because they express contemporary issues more effectively, and fit our contemporary situations better. The Chaos model, life cycle, and strategy are equally as important, but not better than, other concepts.

Third, I compare the Chaos model, life cycle, and strategy to other models, life cycles, and strategies. This paper can be considered a comparison of the ideas presented in my papers about chaos with other ideas in the field. I avoided comparisons in my other papers because I wanted to define those ideas in their own terms and the comparisons did not further the new ideas.

Fourth, I make a few predictions about the next ten years of Software Engineering. The large-scale structures described in this history provide a stronger base for understanding how software engineering will evolve in the future.

This paper is laid out as follows. In the first section, I use the flow of water as a metaphor to describe the flow of progress in Software Engineering. I use the Water metaphor to show some of the structures within Software Engineering. The current work builds on top of the historical work, and future work will build on top of current work. In the remaining sections, I describe the waves, streams, and tides that portray the evolution of concepts and technologies in Software Engineering.

A WATER METAPHOR OF THE EVOLUTION OF SOFTWARE ENGINEERING

In this section, I compare the flow of water with the flow of progress in Software Engineering. I use waves, streams, and tides to describe units of progress. Waves represent individual technical developments. Streams represent sequences of waves or the evolution of technical developments on one theme. Tides represent the simultaneous interest in a group of different technologies, or a generation of technologies. The Water metaphor shows that software development does not advance steadily, but progresses like the slow ebb and flow of a tide.

The Wave Shape of Interest in a Technical Development

Each technical development wave evolves through four stages of interest: innovation, growth, maturity, and convention, as shown by the wave in Figure 1. Waves represent the level of interest in a technical development by the whole community of software engineers. Waves are defined in terms of the level of interest, rather than in terms of the amount of work because the two terms represent distinct concepts. For example, in the 1990s, we still spend fortunes to write and maintain Cobol and Assembler programs, even though few people consider either technology interesting. Waves represent the level of interest by the whole field, rather than the personal convictions of any individual.

Innovation: During the Innovation stage, a neat, new technical development slowly acquires a group of supporters. The technology may require ten or more years to become fully developed and widely understood. It takes a while to spread the word about any new technical development. Some technologies, such as Fortran, took only five or ten years to become popular. Others, such as Object-oriented programming, took more than twenty years to catch on. During the Innovation stage, businesses may take the new technology seriously, but are not yet willing spend money to acquire new tools or to train their employees.

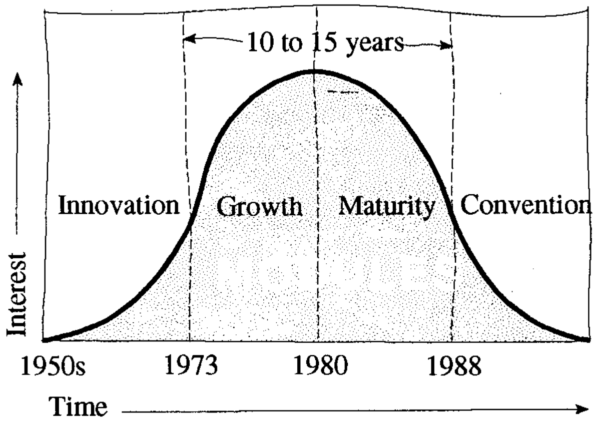

Figure 1: The Wave of Interest in Modules.

Growth and Maturity: During the Growth and Maturity stages we consider the technical development important. In the Growth stage, the technology has been proved out, and it becomes increasingly popular as supporters emphasize its possibilities. In the Maturity stage, interest in the technology levels off, as uniform enthusiasm gives way to a balanced understanding. We finally push the technical development to its limits and find its flaws. We may perceive a backlash against the technology, because detractors emphasize its limitations. Popularity wanes and the technical development begins to burn out. The combined Growth and Maturity stages often last ten to fifteen years. During Growth and Maturity stages, we often confuse technical developments with the projects that implement them. Thus during the 1960s, we did not speak of "function languages," but rather of "Fortran" and "Algol." During the Growth and Maturity stages, businesses will spend money to acquire new tools and train employees to use the technology.

Convention: During the Convention stage, everyone in the field assumes that the core idea of the technical development is valid and the core idea becomes part of the genre or the background information. We know the limitations of the technical development and we address them by moving on and developing new technologies. Modules moved into the Convention stage in 1988, when developers realized that modules do not give much control over allocation and deallocation of resources or the ability to create multiple instances of modules. We are now addressing these issues with objects. As the technical development becomes more conventional, we notice it less and less. The embodying projects die out while the technical development lives on. Thus, the languages Fortran 2 and Algol, which originally embodied functions, have nearly died, even though functions live on as a vital part of almost all contemporary languages. During the Convention stage, businesses often require that employees can proficiently use the technology.

Extending the Water Metaphor (Structures in the Flow)

In this section, I portray large structures within the flow of progress in Software Engineering. Waves combine sequentially to form streams and combine side-by-side to form tides.

Streams: Streams are sequences of waves that represent the evolution of one theme. Waves in a given stream share a focus on a specific problem. Each wave picks up where the last wave leaves off. For example, the Organization stream describes the evolution of organization structures. Figure 2 shows the sequence of organization waves, from the Statement "wave" to the Function "wave" to the Module "wave" to the Object "wave" to the Framework "wave."

In this paper, I focus on eleven loosely interdependent streams. Some streams are technical, others are conceptual. The streams are not necessarily equal in value or force. Some streams are more important than others. We control some streams, such as how we view process. Other streams are out of our control, such as the economics of hardware.

Streams can be very closely related. Sometimes one stream divides into two related streams. When the Waterfall model was introduced, we thought of the terms "model" and "life cycle" as synonyms. I distinguish between these terms because I distinguish between the structure of a project and the sequence of events within a project, respectively. Sometimes one stream strongly influences another. I believe that strategies invariably encourage developers to use current technologies well, so as technologies change, strategies need to keep up. I also argue that models and strategies share a symbiotic relationship.

Tides: Tides are a confluence of many waves that share the same zeitgeist (spirit of the times). For example, the Object tide reflects the increasing importance of object-oriented languages and tools, as well as many collaborating changes, including usability, the Object-Oriented Design strategy, Booch's distinction between the Macro-process and the Micro-process, and the emphasis placed on the design of larger and more sophisticated systems, particularly ones with graphical user interfaces. All of these changes together represent a complete change of worldview. Tides represent popular movements. Each tide overwhelms the field as it comes in, bringing a new set of issues to the surface, and then recedes, leaving old issues scattered or buried. Tides seldom match any particular wave exactly.

In this paper, I focus on six tides. The two best known tides are the Structured Programming tide which lasted from 1967 to 1978 and the Object tide which I expect will last from 1989 to about 2000.

I believe that tides are often related in pairs: a technological change over ten years often inspires a conceptual change in the following ten years. During the first tide of a pair we learn to use a new technology. During the second tide of the pair we learn to use the technology *well*. For example, the Function tide emphasizes using functions while the Structured Programming tide emphasizes using functions *well*. And, the Object tide emphasizes using objects while the Patterned Programming tide will emphasize using objects *well*.

Related Concepts of Scientific and Artistic Progress

The Water metaphor closely resembles the model of artistic progress described by Bowness in *The Conditions of Success*. Bowness points out that successful artists, such as Hockney and Manet, took five or ten years to earn the acceptance of their peers and about twenty five years to earn the full acceptance of the art community. And further, once these artists achieve their breakthroughs, they have "ten (or even five) good years" when they make their greatest impact on the field. Afterwards, their careers gradually taper off, when they do good work, but no longer influence the field as much.

The Water metaphor also resembles the models of scientific progress described by Bauer, Beveridge, Cohen, Ferguson, and Kuhn. Cohen argues that the "scientific revolution" never happened, that scientific progress began long before 1600 and has accelerated steadily ever since. Bauer and Ferguson emphasize slow, long-term progress, arguing that many technical developments follow decades (or even centuries) of steady improvement. Beveridge points out that many important scientific observations precede the "official" scientific discovery by one or more decades. Kuhn emphasizes rapid, short-term progress, arguing that communities undergo revolutions, called "paradigm shifts," when they adopt new technologies.

I believe the Water metaphor encompasses all of these points of view. Like Cohen, I believe that a "software engineering revolution" never occurred, because we addressed the issues of productivity and quality decades before 1968. Most streams in software engineering began long ago. Like Bauer, Beveridge, and Ferguson, I believe that most technical developments follow years of evolution. The waves and tides show how technologies may undergo decades of improvement, then enjoy ten to fifteen years of popularity, and then fade away slowly. Like Kuhn, I believe that technologies sometimes appear revolutionary. Kuhn's revolution corresponds to the rapid growth of interest at the transition from the Innovation stage to the Growth stage.

Which scale is best? I believe that each scale reveals a part of the truth, which is a very complex interplay of all scales. While all of these scales matter, I want to emphasize the medium- and long-term progress.

Dating a Technical Development

Dating the rise and fall of a technology is almost always difficult. Identifying when a technology shifts from the Innovation to the Growth stage is particularly tricky. If we are lucky, we may know the date of the first publication or of a specific event. Thus, we could date the rise of Object-oriented programming from the release of Simula in 1967, the release of Smalltalk-80 in 1980, or the OOPSLA-1 conference in 1986. But, even the most popular events may only foreshadow a surge of interest that takes years to catch on. Perhaps it is more accurate to date a technology from some period of time after an event, say three years, to allow it time to catch on. The trick is to distinguish between when an event occurs and when it matters.

In this paper, I strive to consistently choose dates that bracket the Growth and Maturity stages. I strive to date each technical development from its upswing to the upswing of the next technical development. Thus, I date the Function-oriented programming wave from 1958, one or two years after the first Fortran compilers, to 1973, when modules took over. And, I date the Object-oriented programming wave from 1988, a couple years after the OOPSLA-1 conference and more than twenty years after Simula. These dates are my current best estimate.

Table 1: Fifty Years of Progress in Software Engineering (Approximate)

	Naive Tide 1945—1955	Function Tide 1956—1966	Structured Programming Tide 1967—1977	Module Tide 1978—1988	Object Tide 1989—1999?	Patterned Programming Tide 2000?—2010?
Hardware Economics	Research Mainframes	Commercial Mainframes	Commercial Mini-Computers	Personal Computers		Internet
Organization	Statements	Functions		Modules	Objects	
Optimizers	▨	Statement	Loop, Basic Block		Function	Inter-Function
Programming Environments	▨	Editors and Compilers		General-Purpose Tools	Domain-Specific Tools	
Concepts	▨	▨	Algorithms	Abstract Data Types		Patterns
Program Ideals	▨	Useful	Documented	Correct	Usable	
Models	▨	▨	Waterfall Model		Spiral Model	Chaos Model
Life Cycles	▨	▨	Waterfall Life Cycle		Sashimi Life Cycle	Chaos Life Cycle
Process Structures	▨	▨	Unified Process		Macro- and Micro-Process	Complexity Gap
Strategies	▨	▨	Stepwise Refinement Strategy	Module Decomposition	Object-Oriented Design Strategy	Chaos Strategy
User Participation	None		Once		Periodic	Ongoing

ELEVEN STREAMS

In this section, I describe the eleven streams that most influenced the Chaos model, life cycle, and strategy, even though they do not exactly define Software Engineering. I exclude management, estimation, and testing streams, as they did not influence the Chaos model and Chaos life cycle directly. (I invite others to write about these subjects.) I include hardware economics, organization, and optimization streams, because they strongly influence software engineering, even though depending on your point of view they more properly belong within another branch of computer science. I exclude hardware concerns, such as MIPS and memory, since many people have written about them.

Table 1 details the progress of these eleven streams. In the table, cells represent waves, rows represent streams, columns represent tides, and the whole table represents the flow of progress in Software Engineering. The first five streams denote technologies and the last six streams denote concepts. The figures throughout this text refer to entries in this table.

Hardware Economics

Hardware economics drives software economics. Each generation of hardware makes computers affordable to a new class of users and changes the economics for programmers. I define hardware waves in terms of user perception and economics, rather than electronic technologies which users may not understand or appreciate.

Research Mainframes — 1945 to 1955: In the 1940s and early 1950s computer hardware was developed within research projects. These projects built one-of-a-kind hardware and emphasized research. The programs written from 1945 to 1955 reflect the issues that stemmed from research projects: experimental mathematics, control, and business systems. Early programs solved puzzles or research problems in an ad hoc way. Developers were learning what hardware and software could do. Researchers placed little emphasis on productivity, as the software projects were sponsored as an outgrowth of very expensive hardware development projects.

Commercial Mainframes — 1956 to 1966: During the 1950s, with the commercialization of mainframes, businesses began to emphasize productivity. Mainframes cost millions of dollars, but we recognized that developers were now business employees. To improve productivity, developers created macro-languages, speed coding, and Fortran. But, given the cost of a mainframe and the large support staff needed to run the mainframe, businesses could afford to write their own custom applications for a small portion of their computer budget.

Commercial Mini-Computers — 1967 to 1977: In the 1960s, mini-computers became popular, costing upwards of 50 thousand dollars. For the first time, small businesses could afford their own computers, and the need for software development productivity increased because businesses could no longer hide software development staff in the hardware budget. The quest for more productive software development coincided with the Structured Programming movement of the late 1960s and early 1970s. Compiled languages, such as Cobol and PL/1 became increasingly popular. This wave launched large-scale commercial software, because the owners of mini-computers wanted to purchase software rather than write software. Commercial software companies spread development costs over tens or hundreds of copies of a program.

Personal Computers — 1978 to 2000?: Beginning in the early 1980s, a single software developer's annual salary cost much more than one computer. Personal computers cost thousands of dollars. Thus, the personal computer exaggerated the economics of software development even further. The decreased operating margins forced a completely different way to finance, build, and sell software which led to the current mass markets for consumer and business software. Commercial software companies spread development costs over thousands or millions of copies of a program.

Internet — 2000? to ?: In the late 1990s, the Internet, internationalization, ubiquitous computing, and other economic forces are coming to bear. The Internet provides inexpensive access to tremendous computing resources. Developers will increasingly create, sell, and distribute software directly to consumers, without the overhead of packaging and corporate distribution channels. The Internet will encourage even greater productivity from developers as margins drop even further.

Organization Technologies

Organization technologies are motivated by our need to organize larger and larger programs. In *Object-Oriented Analysis and Design*, Booch describes the evolution of organization technologies in terms of the "program topology" or the relationship between data and code. To some extent, organization defines the relationship between function and data, so it is really a name space problem. But organization affects the way that we think about programs. Figure 2 shows the evolution of the Organization stream from the Statement wave to the Function wave to the Module wave to the Object wave to the Framework wave.

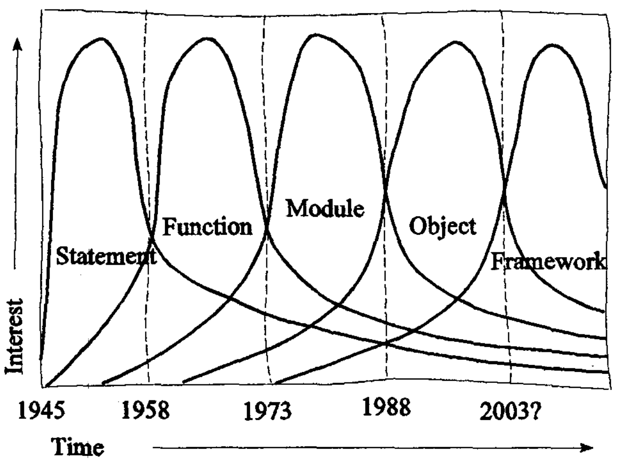

Figure 2: The Organization Stream.

Statement-Oriented Programming — 1945 to 1957: Statement-oriented programming began when programs imitated the step-by-step operation of hardware. Statement languages include machine code and assembly languages. Assembly languages are a big improvement over machine code because Assembly languages eliminate much, if not all, detail about the machine's bit patterns, much of which is arbitrary. According to Booch, in statement-oriented programming all data is global. All statements can access all data. ● We could say that assemblers are expression compilers. In fact, assemblers only compile parts of an expression, but they point to compiling larger expressions, such as the statements found in Fortran 1. Using Assemblers today is more a matter of habit than intent, because assemblers do not define any new ideas and industry shows very little interest in using them. Statement-oriented languages have continued evolving into general-purpose languages like APL and J and command languages like Tcl. The end of this wave became apparent when developers began using macros and Fortran extensively.

Function-Oriented Programming — 1958 to 1972: Function-oriented programming began in the 1950s with Fortran. Fortran had an assembler-like syntax for statements, but extended organization to the function level by adding a special syntax for functions. A function wraps a group of statements together to express one idea more completely. The syntax and implementation of functions was later refined in Algol and Pascal. According to Booch, function-oriented languages distinguish between global and local data and enable programmers to limit access to local data. ● I date the Function wave from 1958 when Algol duplicated Fortran's functions. Subroutines were used in the earliest days of computing. In *The Preparation of Programs for a Digital Computer* from 1951, Wilkes, Wheeler, and Gill describe dividing programs into subroutines. Functions remain vital parts of nearly all contemporary languages. Developers still write lots of code in Function languages or in Object languages using a function-oriented programming style.

Module-Oriented Programming — 1973 to 1987: People noticed Module-oriented programming in the early 1970s, when developers began asking, "how should groups of functions work together?" The goal of modules is to help functions to cooperate with each other. The Module wave began with languages like C, which used the C preprocessor and some innovative header file conventions to create modules. C combines an Algol-like syntax for functions with a macro preprocessor for gluing modules together, enabling developers to build programs out of modules. Other module languages, including Alphard, CLU, Modula, and Ada, improved the syntax and error checking between modules, but did not significantly improve the functionality beyond that provided by C. Unfortunately, the ANSI C committee took fifteen years to bring it up to the standards set by Ada, CLU, and Modula and fix the problems with C's type system. According to Booch, Module-oriented languages distinguish between global, module, and function data, and enable programmers to limit the sharing of information between functions. ● The roots of modules go back to the earliest "assemblers" and linkers from around 1950, Fortran from the middle 1950s, and PL/1 from the middle 1960s. Unfortunately, these early tools provided little help for module design. Early linkers were oblivious to the meanings and types of names. The Fortran common block does a poor job of sharing data, because developers must retype all of the declarations in each module, risking mistakes. During the 1950s and 1960s, developers used modules to optimize compilation rather than to organize programs.

Object-Oriented Programming — 1988 to 2002?: Object-oriented programming helps functions and data to work together very closely. Objects help developers to control the allocation, deallocation, initialization, and especially the management of multiple instances of state variables and data structures, much more effectively than modules. According to Booch, Object-oriented languages distinguish between global, module, object, and function data. Object-oriented programming languages refine the sharing of information between functions, by shifting control from modules to objects. ● Object-oriented technologies were developed shortly after functions, specifically, Simula in the 1960s and Smalltalk in the 1970s. Simula and Smalltalk influenced many of the programming languages of the 1970s and 1980s, even though objects did not truly succeed until after 1988. Objects have now reached the Maturity stage, because we are beginning to criticize their lack of emphasis on semantics. We are making type-safe libraries possible by using templates to plug the holes in the type system. I expect that objects will remain important concepts for many decades to come.

Framework-Oriented Programming — 2003? to ?: Templates and frameworks help objects work together. They can be thought of as type-safe macros or as code generators for objects. In many ways, templates and frameworks apply the data abstraction mechanisms from CLU, Alphard, and Ada to objects. To simplify the distinction between them, templates are language constructs and frameworks are tool or programming environment constructs. Balance defines frameworks indirectly with the statement, "frameworks are to objects as parents are to unruly children." In A Framework-Based Environment for Object-Oriented Scientific Codes, Balance, Giancola, Luger, and Ross describe frameworks in more detail.

Optimizers

Optimizers matter because they free developers from the need to deal with specific small details to concentrate on larger issues, so that developers can do more important things. Optimizers are more important than software engineers may realize. I believe that, for good or bad, developers feel responsible for the efficiency for all of the code that they produce. Developers take responsibility for the optimizer's shortcomings and all code that the compiler does not generate well. This means that before the statement optimizer, developers felt obliged to take responsibility for the assembly code. They felt it necessary to specify the gotos, until optimizers could handle loops and basic blocks. The point is not to forbid access to small details, but to free developers from dealing with unimportant small details. Each transition in optimization technology moves more functionality into the optimizers.

Statement Optimizers: Early Fortran compilers began with statement optimizers. The first compilers optimized each statement independently, one at a time. Statement optimizers allowed developers to ignore the implementation of individual statements, but developers still had to worry about the flow between statements.

Loop Optimizers: In the early 1960s, the Fortran community developed loop and basic block optimization techniques, to assure Fortran's continued reputation as the most efficient language and to stay ahead of the compilers for Algol and PL/1. Compiler writers learned how to optimize loops using "reduction of strength" transformations, which required a thorough understanding of "while" and "if" statements and basic blocks. Developers no longer had to worry about gotos.

Function Optimizers: Full function optimization was part of the on-going improvement in compiler technology. But it wasn't until the middle 1980s that "global optimizing" compilers were well understood, and it wasn't until the 1990s that global optimizing compilers became widely available from commercial sources. Developers no longer had to worry about the code within a function.

Inter-Function Optimizers: Inter-function optimizers that can analyze many functions at once have been under development for many years. Specific techniques range from improving data flow analysis to sharing one activation record among several functions to inlining functions selectively to specializing function calls. However, inter-function optimizers have not yet become part of off-the-shelf commercial compilers.

Programming Environments

Programming environments are the tool sets that we use to improve the productivity of Software Engineers. We have always had programming environments, but some are more useful than others. Now, software engineers assume that a programming environment is a rich tool set. But, note that even today, most tools still do not work together as well as they should and even the current crop of tools hardly encompasses the full scope of what developers do.

Compilers and Editors: The pioneering tool sets of the 1950s included assemblers, compilers, linkers to combine programs with libraries, and editors. Card punches are "card-oriented" editors. At the time, compilers and editors encompassed everything that we thought developers did.

General-Purpose Tools: The concept of "programming environment" really got going when the Unix operating system provided tools that went well beyond basic compilers and editors. Unix supported many more of the activities that all developers do: writing documentation and specifications, coding, testing, and communicating with others. Unix developers wrote the tools: vi, grep, make, cc, rcs, database tools, scripting tools, document processing tools, and mail. Typically each tool implemented one algorithm robustly and provided an interface to read one or a few representations of data from files. In other words, these tools raised algorithms to the level of the user, and pipes enabled users to combine algorithms together from the shell. "Make" is another sophisticated tool for combining programs together. Enthusiasm for the Unix operating system proves that developers both want and need a full set of tools to be productive. The Unix operating system supports the C programming language and other programming environments support the Lisp and Ada programming languages.

Domain-Specific Tools: In the last few years, we have developed tools oriented toward the special needs of specific groups of developers. As developers have specialized into groups that address user interfaces, testing, embedded systems, design, and other issues, they have acquired their own tools. User interface groups use interface builders and dialog editors. Testing groups use scripting tools, test case generators, and error tracking databases. Designers use dataflow diagrams and entity relationship editors. Another recent development is domain-specific languages, such as Perl for scripting and Tcl for embedding. Ad hoc versions of these tools appeared decades ago, but we are now getting them right.

Conceptual Structures

Conceptual structures describe the relationship between specific problems and specific solutions. Conceptual structures are the units of programming analysis and description that we use to catalog our concepts. Conceptual structures exist independently of both applications and technologies. Algorithms, abstract data types, and patterns all document specific solutions to specific problems.

Algorithms: Algorithms describe specific solutions to specific low-level problems. Because of their close ties to analysis, algorithms emphasize low-level details. Knuth was the first to organize and analyze them consistently. *The Art of Computer Programming* was a break-through because, for the first time, developers could use a book of algorithms like chefs use a book of recipes. ● Algorithms extend back to the middle ages, and became increasingly important in the late nineteenth and early twentieth centuries. Knuth didn't invent algorithms, but he was the first to show how important they are to computer science.

Abstract Data Types: In the 1970s, theorists realized that groups of algorithms work together to implement larger concepts. Few algorithms work in isolation. For example, a search tree abstract data type combines algorithms that insert one element, insert many elements, delete one element, find one element, combine two trees, and so forth. Abstract Data Types enable researchers to analyze the behavior of sequences of operations, notably Amortized Complexity. Tarjan and Sleator showed that any sequence of Union-Find and Splay Tree operations is efficient despite the possible inefficiencies of any one operation. Because of their close ties to analysis, Abstract Data Types emphasize low-level details. ● The algorithms that manipulate data structures were developed in groups from the earliest days.

Patterns: The most recent development in conceptual structures is patterns. Developers are now applying Alexander's ideas about patterns to software design. Patterns express conceptual structures from all levels of a project, including the problems defined by applications, the solutions defined by technologies, and everything in between. Some patterns fit mid-way between a problem and a solution, while other patterns focus more on the problem or more on the solution. Because analysis is not the primary use for patterns, patterns are not necessarily tied to low-level details, and so they help to transition away from the underlying technologies. Patterns are particularly important because they describe how middle- and upper-level concepts work together. ● In 1968, Knuth used English to describe his algorithms. Knuth's use of stylized English presaged the style of patterns by twenty five years. But until patterns became popular, describing algorithms in English, rather than pseudo-code, seemed old-fashioned. We can think of algorithms and abstract data types as low-level patterns.

Program Ideals

Program ideals are the goals or properties of programs that we strive to achieve in our code. In one sense, program ideals are the properties that make a program elegant, well-designed, well-implemented, and salable, and therefore make the program worth writing. Ideals are still changing because both developers and users are still learning what computers can and should do. Figure 3 shows the evolution from Useful to Documented to Correct to Usable.

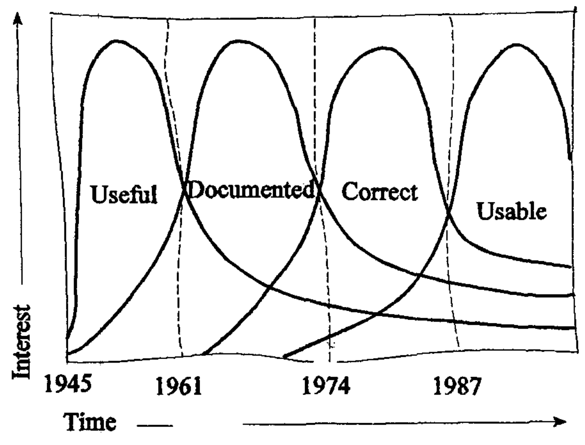

Figure 3: The Program Ideal Stream.

Useful — 1945 to 1960: In the 1940s and 1950s, developers used software to make computers useful. Research often focused on what hardware could accomplish. In today's terms, projects solved very small problems. The programs were often considered less important than the hardware. Programming made the machines useful to mathematicians, scientists, and accountants.

Documented — 1961 to 1973: With the first real commercialization of software, programs needed to be documented to be salable. Many applications were big, ad hoc, and ugly, but if they were documented they could be sold. This was the era of OS/360 and huge manuals. In fact the term "documented" was often interpreted as "well-planned" and "deliberate." ● In the 1990s, some companies still ship enormous manuals on CD-ROM to impress and befuddle their users. Though increasingly, companies strive to minimize the documentation of a program, which often contains errors and overly constrains changes to the functionality of the program.

Correct — 1974 to 1986: In the 1970s, people realized that documentation must not only exist, it must also be meaningful and correct. This means both that documentation must accurately describe the program and that the program must live up to the documentation. The documentation of the time was occasionally wrong, but more often it was ambiguous or incomplete. Consider the old jokes about IBM's "Great Oral Tradition." Many researchers advocated formally proving all programs correct. ● In Programming Languages—The First 25 Years, Wegner points out that the Correctness movement began in the middle 1960s with papers by McCarthy, Naur, Dijkstra, Floyd, and others. Correctness grew to dominate the ideals of the late 1970s and early 1980s. The Correctness wave began to wane in the middle 1980s when DeMillo, Lipton, and Perlis's argument that proofs reflect social processes rather than absolute truths caught on. The Correctness movement never accepted that a program (such as a word processor) with many minor correctness flaws can provide much more benefit to users than no program at all and it never dealt with the rapid life cycles and iterative nature of the consumer software industry. The movement then self-destructed after a controversial editorial to the *Communications of the ACM* in March 1989 about the dangers of criticizing correctness. The ensuing backlash showed that political appeals cannot stay the decline of a wave. The correctness issue continues on today within a small sub-culture, addressing life-critical applications such as pace makers and bomb triggers.

Usable — 1987 to ?: In the 1980s, developers realized that users do not care about the relationship between a specification and its implementation or "correctness." Users want meaningful functionality. Users just want to get their work done. Developers admitted that not all computer users are experts or ever will be. This led developers to admit that naive and casual users won't do difficult tasks and even experts may not make full use of a program with a poor user interface. ● In the 1970s, usability started off at Xerox. In the middle 1980s, the Personal Computer industry (led by Apple) picked up the concept of usability, and graphical user interfaces made it out of the lab. Software vendors learned that sales depend on keeping users happy.

Models

Models describe the structures within software development projects. Models matter because we use them as the banners of the various camps campaigning to improve Software Engineering. Figure 4 shows the progression of structures from the Waterfall model to the Spiral model to the Chaos model.

Waterfall Model: The Waterfall Model, described in Managing the Development of Large Software Systems by Royce, is the mother of all models and it describes simple projects well. The Waterfall model describes software development as a fixed sequence of discrete, irrevocable steps. Programmers should first design everything, then implement everything, and so on. The Waterfall model emphasizes one-shot planning. ● We interpret the Waterfall model to suggest that problems encountered early in a project will only get worse and that to improve the process, we should improve the front end parts of the process most. One criticism is that the Waterfall model fails to account for change and other evolutionary aspects of projects, such as debugging and maintenance. It also fails to guide large, complex, or exploratory projects. ● Royce was the first to define a specific structure for software development.

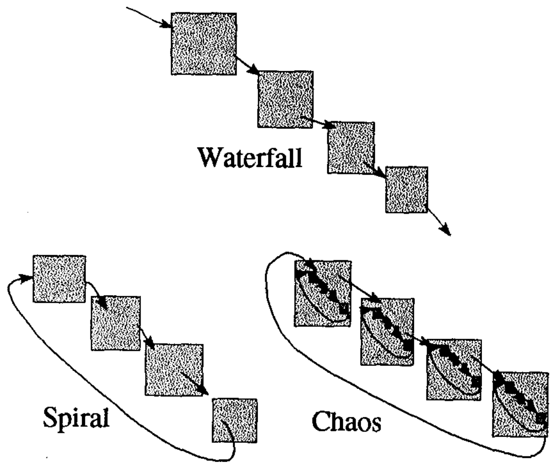

Figure 4: Three Models.

Spiral Model: Boehm described the Spiral model in A Spiral Model of Software Development and Enhancement. The Spiral model says, "build a prototype using the Waterfall model, then revise the objective as necessary, and build a new prototype." Essentially a project is a sequence of prototypes, each of which refines the previous prototype. Since each prototype develops according to the Waterfall model, software development projects resemble a loop of Waterfalls. The Spiral model emphasizes iterative planning. This iterative structure can accommodate more complex, ambiguous, and misunderstood problems. ● In his paper on the Waterfall model, Royce remarked that projects are iterative, but he did not develop the reasons why iteration matters. Boehm was the first to explain how and why the different iterations work together.

Chaos Model: Raccoon described the Chaos model in The Chaos Model and the Chaos Life Cycle. The Chaos model combines a simple problem-solving loop with fractals to describe the many levels of a complex project. All levels matter equally. So, software development resembles a chaotic cascade of Waterfalls. The chaotic complexity allows it to reflect the behavior of the most complex and misunderstood problems. The Chaos model emphasizes planning throughout the process. Raccoon interprets the Chaos model to suggest that software development can be very unpredictable. ● Prior to 1995, researchers proposed many recursive models. In the early 1980s, I recall hearing several speakers at technical conferences comment that the Waterfall model could be applied recursively to parts of a project. In The Impact of DoD-Std-2167A on Iterative Design Methodologies: Help or Hinder? from 1990, Overmyer describes many recursive models. In 1993, Olson argued that because of feedback, software development is chaotic, but he did not describe a specific model. And, in 1996, Kokol, Brest and Zumer discuss chaotic software complexity. Before the Chaos model, nobody carried the recursion down to the "one line of code" level or interpreted the recursion level-by-level.

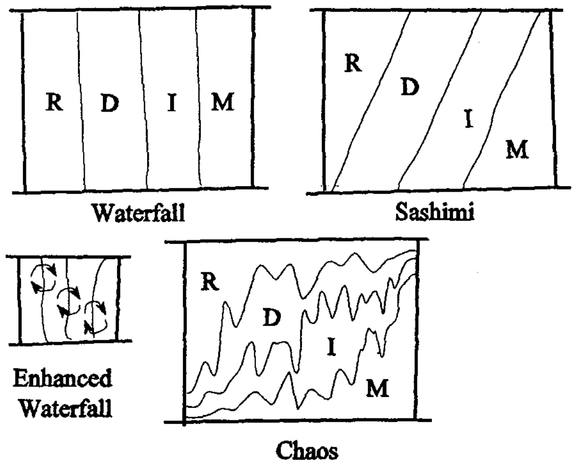

Figure 5: Three Life Cycles.
These diagrams show the percent of effort devoted to Requirements analysis, Design, Implementation, and Maintenance phases as a function of time.

Life Cycles

Life cycles describe the sequence of events within a project. Figure 5 shows the evolution from the Waterfall life cycle to the Sashimi life cycle to the Chaos life cycle.

Waterfall Life Cycle: In Managing the Development of Large Software Systems, Royce makes no distinction between the Waterfall life cycle and the Waterfall model. The Waterfall life cycle rigidly separates the phases of development. We interpret the Waterfall life cycle to suggest that developers should plan to meet specific deadlines and other goals. The Waterfall life cycle contradicts concepts like "design for test," which mix up the phases.

Sashimi Life Cycle: In *Wicked Problems, Righteous Solutions*, DeGrace and Stahl report on the Sashimi life cycle defined by Takeuchi and Nonaka. The Sashimi life cycle allows phases to overlap and the process evolves more flexibly. ● Early extensions of the Waterfall life cycle allowed a project to go back and forth between two adjacent phases, as shown by the Enhanced Waterfall diagram in Figure 5. Unfortunately, this diagram with back arrows reminds me of a Markov chain and conveys a different shift in emphasis than the Sashimi life cycle.

Chaos Life Cycle: Raccoon describes the Chaos life cycle in The Chaos Model and the Chaos Life Cycle. The Chaos life cycle allows phases to come and go, though the whole process gradually shifts from Requirement Analysis to Maintenance. Since all development activities occur throughout the process, the phases show a change in emphasis, rather than substance. Because unexpected difficulties and opportunities can arise, software development projects may evolve unpredictably and developers must respond to these circumstances.

Process Structures

Process structure is about the number of levels that we perceive within a process. Figure 6 shows the evolution from a Unified process to the Macro- and Micro-processes to the Complexity Gap.

Unified Process: In this wave, we perceive software development as one cohesive process. The same kinds of activities occur throughout the whole process. For example, the specification activities are essentially the same as the implementation activities. A unified process suggests that control is possible. One implication is that we can schedule the writing of individual lines of code.

Macro-Process and Micro-Process: In *Object-Oriented Analysis and Design*, from 1994, Booch distinguished between different activities in the Macro-process and the Micro-process. The Macro-process concerns management issues of project scheduling, while the Micro-process concerns hacker-level activities of writing lines of code. In this wave, we recognize that using tools is hard and managing the project is hard, but that these two parts of the process require distinct skills. Booch implies that by elaborating on the parts of process that we already understand, parts that are addressed by contemporary management and developer tools and structures, we will improve software development. Acknowledging that both levels embody a separate process implies that we cannot schedule the writing of lines of code.

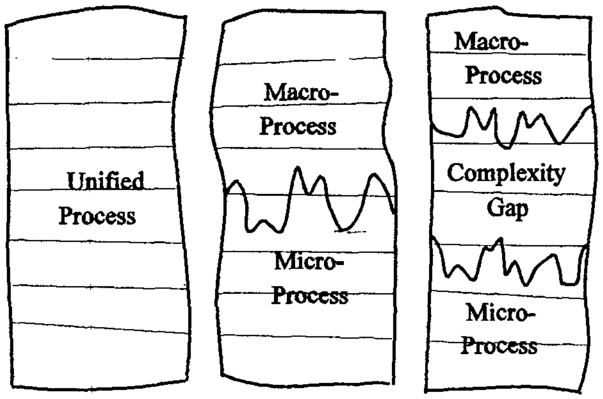

Figure 6: Three Process Structures.
The Top Represents the "Whole Program" Level.
The Bottom Represents the "One Line of Code" Level.

Complexity Gap: In The Complexity Gap, Raccoon argues that the Macro-process and Micro-process do not touch because they concern radically different issues. The Complexity Gap corresponds to the middle-level structures that developers use to match Macro-process goals with Micro-process solutions. The size of the Complexity Gap corresponds to the nature of the problem, as well as the structures and support tools available to help solve the problem. Raccoon argues that matching goals and solutions is very difficult and, in fact, developers spend most of their time working within the Complexity Gap. Raccoon emphasizes the importance of the parts of the process that we don't understand, the parts that are not defined by conventional management and developer tools and structures. Acknowledging that there is a gap between the Macro-process and the Micro-process implies that schedules are related to lines of code indirectly through middle-level structures.

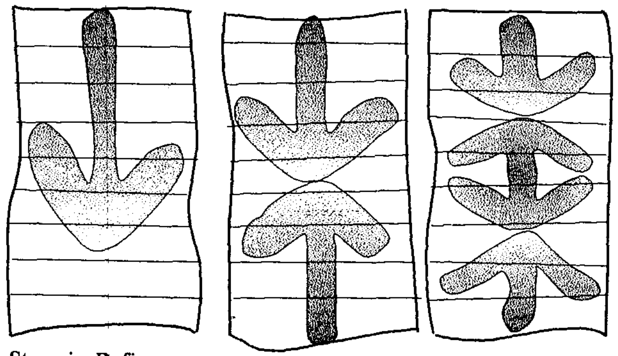

Figure 7: Each Strategy Adds a New Emphasis.
Stepwise Refinement and Module Decomposition strategies emphasizes Top-Down.
Object-Oriented Design strategy emphasizes Bottom-Up.
Chaos strategy emphasizes Middle-Out.

Strategies

All strategies raise the issue of efficient production. Strategies help developers to produce effective programs quickly by expressing priorities. A strategy is a body of knowledge that guides developers through a sequence of actions or state changes, consistently pointing out good moves and avoiding bad moves. Strategies focus attention on the tasks left to finish by identifying the steps that must be completed. An appropriate strategy encourages developers to keep working until the to do list is empty and the program is complete.

Every process follows a strategy. Beginners, randomly choosing any legal move, follow a very naive strategy, though most strategies are more sophisticated. The advantages of Stepwise Refinement over random programming are obvious. Figure 7 shows the evolution of strategies, from the Stepwise Refinement and Module Decomposition strategies to the Object-Oriented Design strategy to the

Chaos strategy.

Stepwise Refinement Strategy: The Stepwise Refinement strategy emphasizes top-down influences. We also call this strategy the "top-down refinement" or "function decomposition" strategy. The developer begins with a conception of the entire program and then breaks that conception into a sequence of smaller steps, expanding each in turn until a program is produced. The Stepwise Refinement strategy defines simple but explicit sequences that work well for simple programs. ● The Stepwise Refinement strategy has four main limitations. First, it works best for programs with lots of code and little data as it ignores the concept of state. Second, it presumes the independence of sub-problems which may be untrue when everything appears connected to everything else. The Stepwise Refinement strategy blindly assumes that a problem really is decomposable. Third, it doesn't allow for corrections and revisions. Some researchers suggested that when this strategy is used properly, developers cannot make mistakes and will not need to make revisions. Thus, it fails to cope with mistakes and requirement changes and other difficulties encountered during typical projects. Minor changes to the problem definition can force major rewrites. Fourth, the Stepwise Refinement strategy provides no means to reconsider past decisions or make corrections. Thus, it can easily lead to local minima that are very frustrating to work with. The Stepwise Refinement strategy can lead to code bloat when developers produce many similar versions of a function because the strategy provides no means to revise the code.

Module Decomposition Strategy: The Module Decomposition strategy emphasizes top-down influences. Parnas applied the Stepwise Refinement strategy to modules and argued that developers should break the application into separate modules, identifying one module after the next, and that good modules should hide information. Parnas discusses the properties of good modules, but unfortunately, not how to create good modules. ● Parnas began describing module decomposition in 1971, and he points out that Gauthier and Pont addressed modules in *Designing Systems Programs* from 1970.

Object-Oriented Design Strategy: The Object-Oriented Design strategy combines top-down and bottom-up goals, but it emphasizes bottom-up goals. Objects combine functions with state variables, so developers need a bottom-up concept to reflect the design of state. ● The analog of "top-down algorithmic decomposition" is "object-oriented decomposition." When the Object-Oriented Design strategy simply means, "design the classes first," it resembles the Stepwise Refinement and Module Decomposition strategies. Booch, in *Object-Oriented Analysis and Design*, and Coad and Yourdon, in *Object-Oriented Design*, each define their own version of the Object-Oriented Design strategy. The Object-Oriented Design strategy is actually independent of programming technology, because it simply emphasizes both top-down and bottom-up issues, and it can be applied to Function and Module languages. ● Booch first described parts of the Object-Oriented Design strategy in Describing Software Development in Ada from 1981. Booch defined the whole strategy in Object-Oriented Development in 1986 and the strategy caught on around 1990.

Chaos Strategy: In The Chaos Strategy, Raccoon defines the Chaos strategy as, "resolve the most important issue first," where issues may come from any level of the project. The Chaos strategy combines top-down, bottom-up, and middle-out goals, but it emphasizes middle-out goals. It points out the difficulties of creating middle-level structures to match up the top and bottom levels. The middle-out emphasis is important for coping with the middle-level issues, such as the components and modules identified by the Complexity Gap. Because the activities in the middle levels are independent of applications and technologies, they differ from activities on other levels. The Chaos strategy is independent of technologies and applies to both Function and Object languages.

User Participation

The relationship between developers and users has evolved slowly over the years. The User Participation stream shows a gradual shift from our emphasis on hardware to our emphasis on individual users, as well as the shift in our perception of the user from being a corporation to being an individual. I believe that users should be present throughout software development, at least conceptually. In some cases, users know what they want and should have authority over the project. In other cases, such as mission-critical and life-critical applications, developers may have special expertise and should have total authority over the project. Developers and users should share the authority over the project appropriately. The following questions describe some of the issues. "How do users fit into software development?" "To what extent do we encourage users to participate?" "Who is responsible for understanding the users's needs?" "How far into the project do we cut off user input?" Figure 8 shows the evolution of the User Participation stream from None to Once to Periodic to Ongoing.

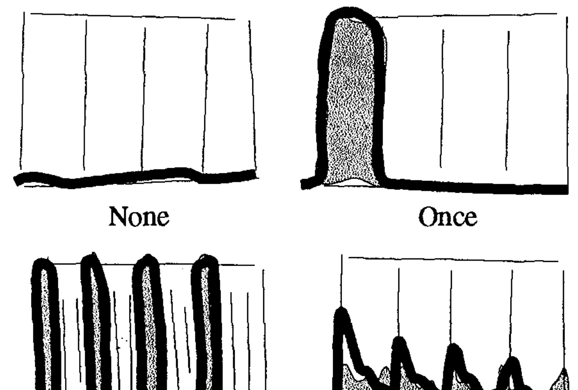

Figure 8: Periods of User Participation in the Process. This shows the percent of process concerned with users.

No Interaction — 1945 to 1969: In the earliest days, we didn't acknowledge that users existed. Developers wrote programs for companies or government agencies rather than "users." And, we defined software development almost entirely in terms of hardware.

One Interaction — 1970 to 1984: During this wave, we tolerated one period of interaction with users: the Requirements Analysis phase. During the requirements analysis phase, developers queried the sponsoring organization. Developers thought that applications were easy and programming was hard. So, once the user pointed us in the right direction, we would valiantly struggle with the hardware on behalf of users. Satisfying the needs of users meant getting a program done. We thought that the real problem was the computer.

Periodic Interaction — 1985 to 1996?: During this wave, we accepted that many applications are hard to define. Developers must repeatedly refer to the user, to ensure that they solve the right problem. In this wave we opened the door to users a little wider, but we still kept users at arm's length. We referred to users periodically, though within any one prototype developers allowed only one step for interaction. As user interfaces became more important in the 1980s, user interface developers included users throughout their process, but software engineers still excluded users from most of their process.

Ongoing Interaction — 1997? to ?: During this wave, we recognize that users participate throughout the process. All software is developed on behalf of users, and working with users is a basic part of all processes. We accept that users are as important as hardware. Satisfying the needs of users means more than simply getting a program done. Users provide an ongoing dialog by requesting enhancements and reporting bugs. This comes up in business and consumer application for personal computers. Developers working on products that compete in the PC mass market value their ongoing relationship with users.

RELATIONSHIPS BETWEEN STREAMS

In this section, I describe the connections between streams, particularly between technologies, strategies, and models. All streams progress together and each stream reacts to changes in the other streams. Advances in one stream often suggest advances in other streams. In fact, an advance in one stream will often be ignored until it is supported by similar advances in other streams. If only one stream in Software Engineering changed, we might not even notice it.

Linking Organization Technologies with Optimization Technologies

Organization and optimization technologies evolve very closely together. One way to think about the importance of optimization is that most developers refuse to use a new notation unless it does them a specific favor. A new organization without a new level of optimization adds the burden of notation for larger concepts, without reducing the developer's responsibilities. On the other hand, improving only the optimizer without improving the organization leaves out half of the benefit to developers. Thus, to make the most improvement, we must improve both organization and optimization technologies at the same time.

In a similar vein, interpreters do not free developers from the need for speed. Interpreters that implement high-level languages do not shrink the range of a developer's responsibility. Interpreters are tools for abstraction and organization, but not optimization. Interpreters have often been used to develop, but have rarely been used to popularize, new organizations. In Compiling Matlab, Johnson and Moler describe the intricate relationship between interpreters and compilers in much more detail.

Function Languages and Statement Optimizers: Function languages organize functions and they optimize statements. Moving from Assembly languages to Function languages with statement optimizers removed several layers of problem solving. Programmers gained more than just smaller, more focused programs. They avoided several levels of problem solving: interpreting the higher-level concepts, generating the assembler code, and optimizing the results. Developers can ignore immense amounts of assembler trivia and they can put that energy to use implementing other aspects of the application. ● Macros with parameters evolved very closely with functions. Macros resemble functions, but provide little support for optimization because macros are based on simple text-based transformations that are unable to express or use the underlying semantic structure. Note: the difference between functions and macros is that functions can be optimized, which explains why modules and objects are built on top of functions rather than on top of macros.

Module Languages and Loop Optimizers: Soon after compiler writers learned to optimize loops, language designers created Module languages. Loop optimizers were developed several years before Module languages, but both technologies caught on together in the 1970s. Loop optimizers allowed developers to use loops and conditional statements for clarity, avoid gotos, and still write efficient code. Loop optimizers became part of the effort to implement Fortran and Algol well, but were used most effectively in module languages. Module organizations could not succeed without improved loop optimizers that free developers from even more low-level details than do statement optimizers. ● By the time global optimizing compilers became available, (which optimized whole functions), we were already on the threshold of the Object wave.

Object Languages and Function Optimizers: Compilers for Object languages can parse the syntax that combines functions into objects and they can optimize whole functions, one at a time. The delay in creating "global optimizing" compilers that could optimize whole functions explains why Object-oriented programming took so long to mature. Until compilers could optimize whole functions, developers felt responsible for implementing all concepts below the function level and were unwilling to focus only on object-level concepts. When compilers could optimize functions, Cox invented Objective-C, Stroustrup invented C++, and Meyer invented Eiffel. Stroustrup was the first to work through the implications of efficiently placing objects on the stack and in static memory. Compilers for early Object-Oriented languages, including Simula and Smalltalk, lacked function optimizers and could not work with other languages and compilers.

Framework Languages and Inter-Function Optimizers: I believe that when we can optimize groups of functions at one time, we will invent another organization technology. Various forms of inter-function optimization have been proposed, including inter-function data flow analysis and function specialization, but we still lack organization principles that empower these optimizers. In *The Design and Evolution of C++*, Stroustrup argues that templates must be as efficient as type-safe macros, so he prohibits templates from providing interesting inter-function optimizations, which means that templates provide less functionality than they could. One important optimization enabled by frameworks is the selection of functions. Balance and Giancola used frameworks to choose the most efficient implementation of each function call in a program. When language designers overcome their current perspectives and meaningfully combine templates with inter-function optimizers, they will invent languages that outstrip C++ and Ada++.

Linking Strategy with Organization and Optimization Technologies

I want to point out the strong connection between technologies and strategies. It seems natural that developers should use language resources as well as their circumstances permit. Developers not only need good technologies, they need to know when and how to use the technologies. Good strategies tell us both. Strategies are tied to optimization and organization in that most strategies say: "use current language resources well" or more specifically "trust both your organization and your optimizer." ● We can think of strategies in both technology-dependent and technology-independent ways. One criticism of the technology-dependent versions of strategies is that they are much too literal. A strategy that is defined in terms of one technology will not keep up with evolving technologies and will succumb to the end of interest in the defining technology. So, the Object-Oriented Design strategy will succumb to the end of current object-oriented programming languages. Yet, the technology-independent concepts within these strategies will last for many decades to come.

Stepwise Refinement Strategy: The Stepwise Refinement strategy guides the development of Structured code. This strategy is best known for emphasizing top-down goals but it actually combines both top-down and bottom-up goals. Both Mill's top-down ideas about functions and Dijkstra's bottom-up ideas about trusting loop optimizers combine to form the whole Stepwise Refinement strategy. **Top-Down:** Mills is renowned for urging developers to use functions well. A simplification of his proposal suggests: when writing a program in C, developers should begin making decisions with the `main()` function and proceed down the "call graph," defining one function after the next. **Bottom-Up:** Dijkstra is renowned for urging developers to "avoid gotos." For developers to follow his advice, they had to trust that their optimizer would implement loops, conditionals, and basic blocks well. Dijkstra wrote about gotos, but he effectively promoted trusting loop optimizers. I believe that without effective optimization technologies, Dijkstra's advice would have been ignored. ● In Structured Programming with Goto Statements, Knuth points out that Schorre was avoiding gotos in 1963 and Landin was avoiding gotos in 1966, but the issue caught fire with Dijkstra's paper in 1968.

Module Decomposition Strategy: The Module Decomposition strategy guides the development of module-oriented code. This strategy is an adaptation of Mill's Function Decomposition strategy to modules that uses the principles of information hiding.

Object-Oriented Design Strategy: The Object-Oriented Design strategy guides the development of object-oriented code. It emphasizes using objects for organization and exploits the new global optimizing compilers. **Top-Down:** From the top, object decomposition is an adaptation of the Function Decomposition and Module Decomposition strategies. The Object-Oriented Design strategy encourages developers to break the application into a set of objects, identifying each object, one after the next, until a program is produced. **Bottom-Up:** From the bottom, two versions of the Object-Oriented Design strategy encourage developers to use bottom-level technologies well. Both versions urge developers to trust that their compiler optimizes functions well. The Booch version says: "use Object-oriented concepts well" by writing or changing one line of code or restructuring one whole chunk of code, until the program is done. The Coad and Yourdon version extends this argument to "use all standard technologies well." The Coad and Yourdon version is not truly "Object-Oriented," as they recommend using all sorts of technologies including relational databases, but it is a contemporary strategy as they recommend using all technologies available in 1991 to their fullest.

Chaos Strategy: Templates and patterns are clearly middle-level constructs and the Chaos strategy says use resources from all levels well. Since patterns apply to all levels, the Chaos strategy means, "use patterns well." In *The Timeless Way of Building*, Alexander describes a version of the Stepwise Refinement strategy called "One Pattern at a Time" that is much too simple and out of keeping with the flexibility of patterns. He really needs the Chaos strategy. No other book on patterns discusses strategies for applying patterns. The Chaos strategy can handle the up-coming inter-function optimizers, frameworks, and templates, which address issues closer to the middle levels of contemporary applications.

Linking Models with Strategies

Models and strategies exist independently of each other, yet they need each other. Models define the structures within processes. Strategies give day-by-day and line-by-line advice about software development. Since models don't give us line-by-line advice, we need another concept like strategy. Since strategies don't define the larger structures within a project, we need models to support the use and application of each strategy.

Waterfall Model and Stepwise Refinement Strategy: The Waterfall model describes large-scale structures in a project rather than day-to-day programming activities. Stepwise refinement gives advice for completing independent and decomposable problems, rather than large-scale program structures. The Waterfall model and Stepwise Refinement strategy seem well suited to each other because both define a very self-assured, straight-forward, one-shot approach to software development and neither accommodates revisions or mistakes.

Spiral Model and Object-Oriented Design Strategy: The Spiral model describes large-scale iterations within projects and ignores the day-to-day programming activities. The Object-Oriented Design strategy gives advice about adding one line of code, changing one line of code, and reorganizing sections of code rather than large-scale structures. However, the Object-Oriented Design strategy needs an iterative concept of process structure like the Spiral model. In the context of the rigid Waterfall model, the flexible Object-Oriented Design strategy seems wishy-washy and out of place.

Chaos Model and Chaos Strategy: One criticism of the Chaos model is that it is so flexible that anything is possible and it doesn't really tell anybody what to do, so developers need a strategy to guide their efforts. However, without a multi-level model such as the Chaos model, the flexible Chaos strategy seems wishy-washy. The Chaos model and Chaos strategy support each other.

Boehm used the Spiral model to argue that managers should do risk analysis at the beginning of each prototype, when they define the goals of the prototype. Raccoon used the Chaos model to argue that developers should always resolve the most important issue first, which can be interpreted as extending Boehm's advice to all levels of a chaotic process.

Linking User Participation with Models

Models discuss the relationship between developers and users. These relationships can be explicit, as described in the Chaos model, which devotes a whole section to the issue, or implicit, as described in the Waterfall model, which defines the Requirements Analysis phase. I believe that the role that users play should depend on the particular application, so models should support user participation to the extent that it is necessary. ● Before the Waterfall model, we didn't document the roles of users explicitly.

Waterfall Model and One Interaction: The Waterfall model focuses on contract-based relationships and specifications, and simplifies the goal of pleasing the user to meeting the specification. The Waterfall model defines exactly one period of interaction with users: the Requirements Analysis phase. We use the Waterfall model to emphasize one-shot projects. This perspective is more prevalent in projects sponsored by the DOD and DOE because it fits with the top-down, authoritarian style of these organizations.

Spiral Model and Periodic Interaction: The Spiral model focuses on contract-based relationships and specifications. It still simplifies the goal of pleasing the user to meeting the specification, but it enables more user input throughout the process. We use the Spiral model to emphasize long-term, sequential relationships with users. The Spiral model encourages developers to repeatedly refer to the user, though each prototype permits one specific step for this interaction.

Chaos Model and Ongoing Interaction: The Chaos model states that users participate throughout the process. The Chaos model documents the ongoing interaction between users and developers as an important part of software development. Raccoon uses the Chaos model to show that users, developers, and technologies work together throughout the process to create useful technical solutions. Users strongly influence the top half of the Chaos model. Raccoon did not invent ongoing interaction, rather the Chaos model documents the relationship with users that many companies have fostered for years.

In A Rational Design Process: How and Why to Fake It from 1986, Parnas and Clements argue that even though processes are iterative or perhaps even random, they should be treated as if they resembled the Waterfall model. They call this a "rational" approach to process. ● Now that we recognize our Ongoing interaction with users, we structure it into a Periodic interaction. In the consumer software industry, developers cannot produce a special version of software after each enhancement request and bug report. So, we group the users' requests into minor and major revisions, and typically produce a minor release every three months and a major release every year or two. For very "rational" reasons, we structure our Ongoing interaction into a Periodic interaction.

Linking Other Streams With Models

Many people think that models are theoretical curiosities. Yet many aspects of our contemporary reality can be interpreted in terms of contemporary models. Table 2 links models with three other streams. I believe that each of these streams evolved of its own volition. Models reflect, but do not impose, these perspectives.

Table 2: Linking Models with Three Other Streams

Models	Reuse	Estimation	Maintenance
Waterfall	Each project was independent. We didn't think about reuse, except in the context of portability.	We estimated projects independently, one at a time, using ad hoc methods.	Maintenance is a total loss. So emphasize proper specification, design, and implementation to avoid maintenance.
Spiral	Reuse became popular with the Spiral model. Of the 178 references in *Confessions of a Used Program Salesman* by Tracz, only 38 came before 1985, and only 13 before 1980. During the late 1980s and early 1990s, reuse was a very "Spiral" concept. We reused a unit of code on one level. We treated it like a product and reused the whole chunk or didn't reuse anything.	Boehm estimates the effect of each project based on past projects. The Spiral model led to the Capability Maturity Model using a sequence of projects as the basis for the next bid.	Maintenance feeds into the next project. As long as code is reused, maintenance can be paid for by the next project, so maintenance is good. We emphasized maintaining code, but not any other artifacts.
Chaos	In the late 1990s, we will reuse code on many levels. Different people will reuse parts from all scales of a project, including all sizes of code, designs, and specifications.	Learning curves emphasize the economics of ongoing processes. (Mandelbrot shows that log-log space is chaotic.)	The Chaos life cycle shows that maintenance equals development throughout the process. We maintain code in all phases (including specifications, design, testing) on all levels.

SIX TIDES

When many aspects of our software development reality change, the combined waves form a tide. We cannot ignore these broad-based transitions, even if many of the changes are only loosely related. This section builds on the linking between streams in the previous section.

I believe that from 1945 to 1999, Software Engineering will have evolved through five major tides, each lasting about eleven years. I believe that we are on the verge of the next tide that will take Software Engineering into the next millennium.

Naive — 1945 to 1955

The zeitgeist of the Naive tide is "computers are neat." We just began to understand computer hardware and software. Since many people believed that software controlled hardware, they believed that software and hardware shared common problems and solutions. Large programs were hundreds of lines long.

Software Engineering began with a decade of naive programming: the Naive tide. This tide could also be called the "Natural," "All-At-Once," or "Mystical" tide. The name "naive" does not necessarily mean that developers did a bad job. At the beginning of this tide, we didn't know what we were doing, and uninformed developers often did bad work. We had no tools, no methodologies, no experience, and no body of knowledge about how to create quality software productively. I suspect that many developers were very good and intuitively did many things well. But in general, developers were unaware of the common concerns and solutions in Software Engineering, and they were unable to consistently do a good job.

As the Naive tide evolved, we developed the inkling that software differs from hardware. Researchers invented linkers and assemblers and other tools that exaggerated the differences between software and hardware. Software development resembled proving a theorem or writing a novel, because a motivated developer worked hard and almost magically produced a program. The process was undefined and we made few distinctions about what went on within the process. Anything goes. Whatever works. Programs happen.

Functions — 1956 to 1966

The zeitgeist of the Function tide is, "functions are neat." We understood that software development differs from hardware development. Programming was accepted as an activity in its own right, as it had its own concepts and goals and its own tools, problems, and methodologies. Fortran embodied the first concept of a high-level language for users. The Function tide combines compilers for function and macro languages, editors, and other tools, and the new objective of making computers useful. Functions flourished with the commercialization of mainframes and the subsequent push to improve developer productivity.

Functions are Neat: The second tide concerned the use of functions. Language designers used functions in many ways, inventing macros, Fortran, Lisp, Algol, and PL/1. In one way or another, functions have influenced computer science ever since. Functions raised the following questions. The question, "Is there a better syntax?" led to the Algol, Pascal, and C families of languages. The question, "Is there a better software architecture for Algol?" led to the refinement of stack allocation and recursion. The question, "Is there better hardware functionality?" led to stack architectures.

Structured Programming — 1967 to 1977

The zeitgeist of the Structured Programming tide combines "use functions and loop optimizers well" with "comparing techniques." Structured programming was the first movement to be big and popular, in part because it had a very clear identity. The Structured Programming tide combines the following waves: the Stepwise Refinement strategy, the Waterfall model, using functions and loop optimizers well, the rise of algorithms, and the debates over Goto statements. Structured programming evolved with the commercialization of mini-computers and the subsequent demand for cheaper software. Also, in the 1960s, applications grew much larger than ever before.

Use Functions and Loop Optimizers Well: The Structured Programming tide emphasized using the optimizing compilers for Fortran, Algol, and PL/1 well. By the late 1960s, compilers for function languages could optimize loops, and developers began wondering how to make the best use of these technologies. The question of how to use functions well, led to the Stepwise Refinement strategy and the Goto wars. Ever since, we have accepted that we should minimize gotos, though to what degree is a matter of individual taste and "religion."

Comparing Techniques: Intuitively, some software development concepts work better than others. In the middle 1960s, people began seriously studying the differences between software development technologies. Though, comparisons were applied to many areas of software development, three streams had a special influence. In programming languages with the infamous Goto debates, in the development of algorithms with analysis showing that some algorithms were better than others, and in the strategy stream with the comparison of Stepwise Refinement with Naive strategies, developers made large progress. ● The first comparisons were often either trivial or ridiculous. The Stepwise Refinement strategy is obviously better than the Naive strategy, and Algol is obviously a better tool than assembly language for writing general purpose code. On the other hand, the comparisons between Algol and Fortran, which consumed so much effort, now seem spurious. Both languages enabled developers to write much cleaner and smaller programs than assemblers, and the differences between them have faded in the light of newer languages and technologies.

Algorithms: Algorithms arose at the beginning of the Structured Programming tide. The study of algorithms evolved closely with the study of functions. We learned that each function should implement one algorithm well. Analysis of algorithms is also a metaphor for making comparisons in other streams.

It seems natural that the first "Software Engineering" conference in 1968 would coincide with the rise of the Structured Programming tide. I believe that all of the waves in this tide strove to reach beyond the naive and ad hoc practices of the first two decades of software development. The waves in the Structured Programming tide share the same motivation: the quest to find the right way to develop and manage software.

Modules — 1978 to 1988

The zeitgeist of the Module tide combines "groups are neat" with "correctness." Much of the Module tide reacted to improvements made during the Structured Programming tide. In some ways this tide represents the consolidation of Structured Programming goals. In other ways this tide is a stepping stone toward objects. Modules represent the shift from individual things to groups of things. The Module tide combines modules, general-purpose tools, the beginnings of programming environments, the Module Decomposition strategy, correctness, Abstract Data Types, and the debates over Ada. The Module tide was a step on the way from functions to objects, because data was still not the equal of functions. This wave began with the start of the PC era and the increasing need for productivity. In the late 1970s and 1980s, software engineering began to address huge problems, such as the Strategic Defense Initiative.

Groups are Neat: Groups sprouted in every possible way. Abstract data types are groups of algorithms, modules are groups of functions, and programs are groups of modules. Developers need more than one compiler and one editor, they need a whole suite of tools. Parnas even argued that applications are groups of programs.

Correctness: In the 1970s, we moved from the notion of correct functions to correct programs, partly as a matter of scaling up, and partly as a reaction to Structured Programming and Software Engineering. As the meaning of algorithms sank in fully, many people concluded that programs were just large algorithms, and if algorithms could be correct, why not programs? Researchers interpreted algorithms as a metaphor and suggested that if developers "do it right," they can and will produce perfect software.

Objects — 1989 to 1999?

The zeitgeist of the Object tide combines "iteration" with "objects are neat." The Object tide combines object-oriented programming languages, global optimizing compilers, the Spiral model, the Sashimi life cycle, the Object-Oriented Design strategy, domain specific tools, and usability. This tide also took off with graphical user interfaces. It has taken decades for developers to appreciate the need for objects, in part because functions and modules effectively organize programs up to 50,000 or 100,000 lines of code long and in part because it needed collaborating changes in other streams.

Iteration: In the middle 1980s, iteration became one of the main themes in Software Engineering. The Spiral model and the Object-Oriented Design strategy both embody iteration or a sequence of prototypes. Applications for businesses and consumers, such as word processors and spreadsheets, were rewritten every couple of years. Reusing software over and over became an important goal.

Objects are Neat: The Object tide moved into full swing with the popularization of C++, Objective-C, Eiffel, and the OOPSLA conferences. People began using objects everywhere. They created object-oriented versions of every language, system, and concept around. They defined the Object-Oriented Design strategy and object versions of tools, notations, and methodologies. They even rewrote many algorithm texts in terms of objects.

Graphical User Interfaces: Many people link graphical user interfaces with objects, because we use objects to implement many graphical user interfaces. In fact, objects do not really do graphical user interfaces very well, but they do organize larger programs well, which is particularly a problem for programs with graphical user interfaces. Tools such as dialog editors do graphical user interfaces well.

Patterned Programming — 2000? to 2010?

Extrapolating out from the previous tides reveals glimpses of the next tide to rise within Software Engineering. It is hard to make concrete predictions about how the Patterned Programming tide will evolve and we will probably only understand it in retrospect. But I can make some estimations. I believe that the zeitgeist of the Patterned Programming tide will combine "using objects well" with "understanding the middle levels." The Patterned Programming tide will combine the technologies: Frameworks and templates, inter-function optimizers, and patterns; and the concepts: the Chaos model, the Chaos strategy, the Complexity Gap, and the Ongoing nature of software development. All of these waves show that we are striving to go beyond the object-oriented concepts from the late 1980s and early 1990s. I also believe that the "Patterned Programming" tide will rise in the late 1990s and fall around 2010 when new ideas, technologies, and perspectives will address the holes found during this tide.

Using Objects Well: Over the last ten years, we have been adopting object-oriented technologies. We now realize that objects are not enough, objects have limitations. We must do more than simply use object-oriented languages and tools, we must use objects well. For example, Liskov and Wing point out some of the limitations of "structural" uses of inheritance and advocate "semantic" or "behavioral" inheritance. Templates have been added to C++ and Sather to plug holes in the type systems.

Middle Levels: The Chaos model, Chaos strategy, Complexity Gap, Patterns, and templates all focus on the middle levels of a project. The Chaos model and the Complexity Gap define a multi-level structure and explicitly identify the middle levels as interesting. I believe that patterns work on all levels, specifically the middle levels, and that frameworks and templates will address issues near the middle levels of large software development projects.

I call this tide the "Patterned Programming" tide, because patterns seem to be the most widely used (and misused) concept in the middle 1990s. I suspect that eventually somebody will argue that programs are just large patterns. ● A whole tide requires collaborating changes in many streams. By themselves, patterns cannot define a whole tide. Patterns are not a programming language nor an optimizer, and they don't define program structures. Ironically, Alexander's simple variation on the Stepwise Refinement strategy for applying patterns is very simple for such a sophisticated concept. He really needs the Chaos strategy. ● I also want to suggest a parallel to the "Structured Programming" tide. I believe that during the Patterned Programming tide, we will strive to use objects well, like we strove to use functions well during the Structured Programming tide. We are now adding type parameters to objects, like they added type parameters to functions during the Structured Programming tide.

CONCLUSION

Software engineering may seem to evolve very rapidly, with one wave after the next crashing against the shore of contemporary practice in a seemingly endless storm of change. Yet, when viewed in terms of streams and tides, we can see that the core ideas evolve at a remarkably steady rate.

The history described in this paper shows that developers, managers, and researchers have made steady progress toward the goal of creating reliable software productively during the last fifty years. I believe that Software Engineering can only progress so fast. To improve, we must find the middle of the current perspective and get good at using the current technology, before we can understand its limitations and how it will affect other streams. To progress to the next tide, all of the streams must progress together. I believe that Software Engineering will steadily progress and grow for decades to come. For comparison, we can look to the much older fields of physics and the fine arts that are still progressing after millennia of improvement.

Some streams of software development may stop evolving. For example, I do not know of any interesting complexities beyond Chaotic, so models may stop evolving in a complexity sense. If models continue to evolve, they will change in a different sense than complexity. Other streams will continue evolving. I expect that organization and optimization technologies will continue improving for many decades to come and therefore strategies will have to keep up.

Even though the term "Software Engineering" was first officially used in 1968, we have been addressing the issues of productivity and quality ever since the first digital computer was built in 1945. Of the streams discussed in this paper, half go back to the first decade, and the rest go back to the second or third decades of software development. In two very real senses, all streams go back to 1945. First, I could postulate that each stream began with a wave in 1945. I could postulate a Naive model and a Naive strategy that precede the Waterfall model and the Stepwise Refinement strategy. I could postulate that the first programming environment consisted of pen and paper and that the first optimization technology was the Null optimizer. Thus, all streams extend back to the earliest software development projects. Second, all of the improvements made during the last fifty years are grounded in our collective experience that began in 1945. According to the history presented in this paper, we just passed the 50th Anniversary of Software Engineering.

ACKNOWLEDGEMENTS

I want to thank everyone with whom I have discussed the history of Software Engineering or compared two technologies. I especially want to thank Bear, Charles Crowley, K. C. Cress, Anthony J. Giancola, Arthur B. Maccabe, Puppydog, Michael Thompson, Edwin E. Wing, Judith C. E. Wing, Margaret J. Wing, and Stephen P. Wing. I most especially want to thank to Mina Yamashita.

BIBLIOGRAPHY

Christopher Alexander (1979) *The Timeless Way of Building*, Oxford University Press.

Mark Ardis, Victor Basili, Susan Gerhart, Donald Good, David Gries, Richard Kemmerer, Nancy Leveson, David Musser, Peter Neumann, and Friedrich von Henke (1989) Editorial Process Verification, in *Communications of the ACM*, Volume 32, Number 3, Pages 287 and 288, March 1989, ACM Press.

Robert A. Ballance, Anthony J. Giancola, George F. Luger, and Timothy J. Ross (1994) A Framework-Based Environment for Object-Oriented Scientific Codes, in *Scientific Programming*, Volume 2, Pages 111 to 121, John Wiley and Sons.

Henry Bauer (1992) *Scientific Literacy and the Myth of the Scientific Method*, University of Illinois Press.

W. I. B. Beveridge (1950) *The Art of Scientific Investigation*, Vintage.

Barry Boehm (1986) A Spiral Model of Software Development and Enhancement, in *Software Engineering Notes*, Volume 11, Number 4, Pages 14 to 24, August 1986, ACM Press.

Grady Booch (1981) Describing Software Design in Ada, in *SIGPLAN Notices*, Volume 16, Number 9, Pages 42 to 27, September 1981, ACM Press.

Grady Booch (1982) Object-Oriented Design, in *Ada Letters*, Volume 1, Number 3, Pages 64 to 76, March/April 1982, ACM Press.

Grady Booch (1986) Object-Oriented Development, in *Transactions in Software Engineering*, Volume 12, Number 2, Pages 211 to 221, February 1986, IEEE Press.

Grady Booch (1994) *Object-Oriented Analysis and Design*, Benjamin Cummings.

Alan Bowness (1989) *The Conditions of Success*, Thames and Hudson.

Frederick P. Brooks, Jr. (1995) The Mythical Man-Month after 20 Years, in *The Mythical Man-Month*, 20th Anniversary Edition, Addison Wesley.

Peter Coad and Edward Yourdon (1991) *Object-Oriented Design*, Yourdon Press.

H. Floris Cohen (1994) *The Scientific Revolution: A Historiographical Inquiry*, University of Chicago Press.

O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (1972) *Structured Programming*, Academic Press.

Jeffrey Dean, Craig Chambers, and David Grove (1995) Selective Specialization for Object-Oriented Languages, in *SIGPLAN Notices*, Volume 30, Number 6, Pages 93 to 102, June 1995, ACM Press.

Peter DeGrace and Leslie Hulet Stahl (1990) *Wicked Problems, Righteous Solutions*, Yourdon Press.

R. DeMillo, R. Lipton, and A. Perlis (1979) Social Processes and Proofs of Theorems and Programs, in *Communications of the ACM*, Volume 22, Number 5, Pages 271 to 180, May 1979, ACM Press.

Edsger W. Dijkstra, Goto Statement Considered Harmful, in *Communications of the ACM*, Volume 11, Number 3, Pages 147 to 148, March 1968, ACM Press.

Eugene S. Ferguson (1992) *Engineering and the Mind's Eye*, MIT Press.

James H. Fetzer (1988) Program Verification: The Very Idea, in *Communications of the ACM*, Volume 31, Number 9, Pages 1048 to 1063, September 1988, ACM Press.

Richard Gauthier and Stephen Pont (1970) *Designing Systems Programs*, Prentice Hall.

Stephen C. Johnson and Cleve Moler (1994) Compiling Matlab, in *USENIX Very High Level Languages Symposium Proceedings*, Pages 119 to 127, October 1994, USENIX Association.

Donald E. Knuth (1968) *The Art of Computer Programming*, 3 Volumes, Addison Wesley.

Donald E. Knuth (1974) Structured Programming with Goto Statements, in *Computing Surveys*, Volume 6, Pages 261 to 301, December 1974, ACM Press.

Peter Kokol, Janez Brest, and Viljem Zumer (1996) Software Complexity - An Alternative View, in *SIGPLAN Notices*, Volume 31, Number 2, Pages 35 to 41, February 1996.

Thomas S. Kuhn (1970) *The Structure of the Scientific Revolutions*, University of Chicago Press.

Barbara H. Liskov and Jeanette M. Wing (1995) A Behavioral Notion of Subtyping, in *Transactions on Programming Languages and Systems*, Volume 16, Number 6, Pages 1811 to 1841, November 1994, ACM Press.

Benoit B. Mandelbrot (1983) *The Fractal Geometry of Nature*, Freeman.

Harlan D. Mills (1971) Top-Down Programming in Large Systems, in *Debugging Techniques in Large Systems*, Randall Rustin editor, Prentice Hall.

Dave Olson (1993) *Exploiting Chaos: Cashing in on the Realities of Software Development*, Van Nostrand Reinhold.

Scott P. Overmyer (1990) The Impact of DoD-Std-2167A on Iterative Design Methodologies: Help or Hinder?, in *Software Engineering Notes*, Volume 15, Number 5, Pages 50 to 57, October 1990, ACM Press.

David Lorge Parnas (1972) On the Criteria to Be Used in Decomposing Systems into Modules, in *Communications of the ACM*, Volume 15, Number 12, Pages 1053 to 1058, December 1972, ACM Press.

David Lorge Parnas (1976) On the Design and Development of Software Families, in *Transactions on Software Engineering*, Volume 2, Number 1, Pages 1 to 9, January 1976, IEEE Press.

David Lorge Parnas, Paul C. Clements, and David M. Weiss (1985), The Modular Structure of Complex Systems, in *Transactions on Software Engineering*, Volume 11, Number 3, Pages 259 to 266, March 1985, IEEE Press.

David Lorge Parnas and Paul C. Clements (1986) A Rational Design Process: How and Why to Fake It, in *Transactions on Software Engineering*, Volume 12, Number 2, Pages 251 to 257, February 1986, IEEE Press.

L. B. S. Raccoon (1995) The Chaos Model and the Chaos Life Cycle, in *Software Engineering Notes*, Volume 20, Number 1, Pages 55 to 66, January 1995, ACM Press.

L. B. S. Raccoon (1995) The Complexity Gap, in *Software Engineering Notes*, Volume 20, Number 3, Pages 37 to 44, July 1995, ACM Press.

L. B. S. Raccoon (1995) The Chaos Strategy, in *Software Engineering Notes*, Volume 20, Number 5, Pages 40 to 47, ACM Press.

L. B. S. Raccoon (1996) A Learning Curve Primer for Software Engineers, in *Software Engineering Notes*, Volume 21, Number 1, Pages 77 to 86, January 1996, ACM Press.

W. Royce (1970) Managing the Development of Large Software Systems: Concepts, WESCON Proceeding (Aug.).

Hirotaka Takeuchi and Ikujiro Nonaka (1986) The New New Product Development Game, *Harvard Business Review*, Volume 64, Number 1, Pages 137 to 146, January-February 1986.

Robert Endre Tarjan (1983) *Data Structures and Network Algorithms*, SIAM.

Will Tracz (1995) *Confessions of a Used Program Salesman*, Addison Wesley.

Doris B. Wallace and Howard E. Gruber (1989) *Creative People at Work*, Oxford University Press.

Peter Wegner (1976) Programming Languages—The First 25 Years, in *Transactions on Computers*, Volume 25, Number 12, Pages 1207 to 1225, December 1976, IEEE Press.

Maurice V. Wilkes, David J. Wheeler, and Stanley Gill (1951) *The Preparation of Programs for an Electronic Digital Computer*, Addison Wesley.