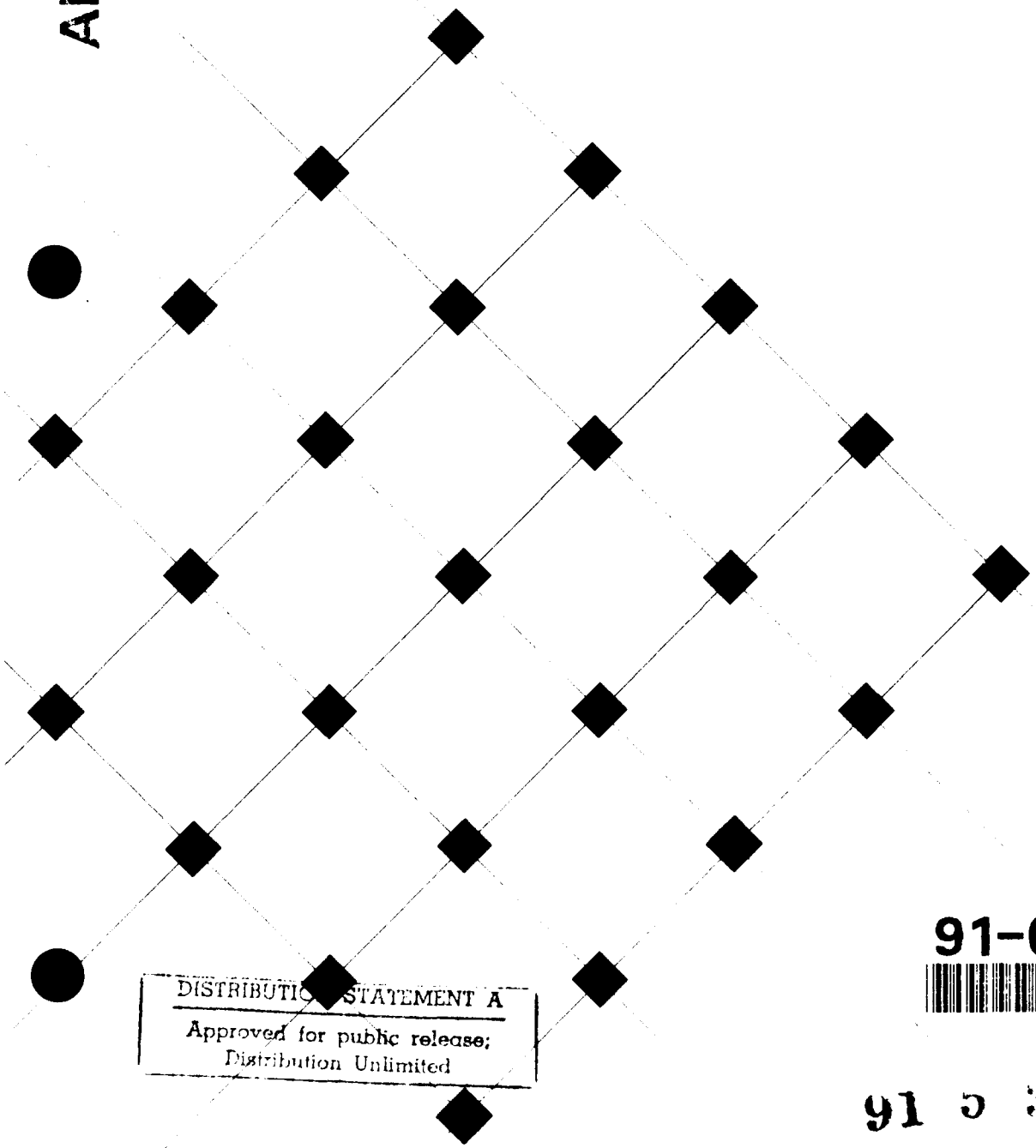
(2)

Carnegie-Mellon University
Software Engineering Institute

Models of Software Evolution: Life Cycle and Process

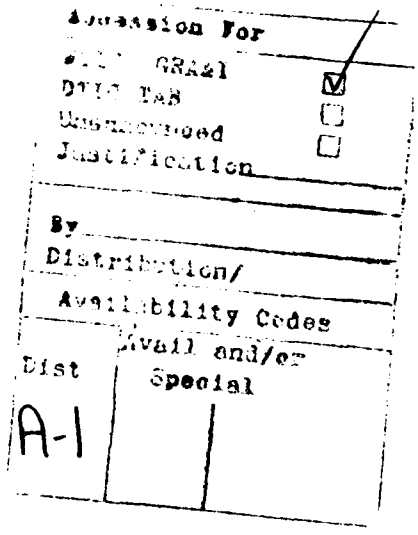
Curriculum Module SEI-CM-10-1.0



Models of Software Evolution: Life Cycle and Process

SEI Curriculum Module SEI-CM-10-1.0

October 1987



Walt Scacchi
University of Southern California

Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.

Draft For Public Review

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

JOHN S. HERMAN, Capt, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Models of Software Evolution: Life Cycle and Process

Acknowledgements

Priscilla Fowler provided helpful comments and suggestions during the development of this module. Robert Glass and Marc Kellner also provided helpful comments in review of this module.

Contents

Models of Software Evolution: Life Cycle and Process

Module Revision History

Version 1.0 (October 1987) Draft for public review

Models of Software Evolution: Life Cycle and Process

Capsule Description

This module presents an introduction to models of software system evolution and their role in structuring software development. It includes a review of traditional software life-cycle models as well as software process models that have been recently proposed. It identifies three kinds of alternative models of software evolution that focus attention to either the products, production processes, or production settings as the major source of influence. It examines how different software engineering tools and techniques can support life-cycle or process approaches. It also identifies techniques for evaluating the practical utility of a given model of software evolution for development projects in different kinds of organizational settings.

Philosophy

This module presents the concepts and approaches for organizing software engineering activities over the life of software systems. As such, it focuses attention to:

- what software life-cycle models are and how they are used
- what software process models are and how they can be used to model the software life-cycle
- traditional software life-cycle models
- alternative software evolution models centered around software product, production process, or production setting characteristics
- how software engineering tools and techniques fit into the models
- techniques for evaluating software evolution models and methodologies
- techniques for customizing software life-cycle process models to best suit individual own needs.

Objectives

The material covered by this module seeks to convey to students the following objectives:

- a basic recognition that software systems can be produced and consumed according to different systematic models of software evolution
- there are alternative ways to organize software development efforts, and that the alternatives can focus attention to software product, production process, or production setting characteristics
- more attention is being focussed to codifying models of software evolution into computational forms amenable to simulation, analysis, and articulation of schemes for integrating various software engineering tools and techniques
- software evolution is itself a process that can be prototyped, systematically developed, (re-)configured, measured, refined, maintained, and managed

Prerequisite Knowledge

The prerequisites for this module depend on the level of coverage intended for students. For a short introduction to life-cycle models of three hours or less, an introduction to computer science and pro-

gramming is sufficient. For a more in-depth treatment of traditional and alternative software life-cycle models of 15-20 hours, then prior experience as a participant in a software development project is strongly recommended, as is knowledge of computational process models (e.g., state machines, augmented transition networks, petri networks). For an advanced, full course-length examination of software life-cycle and process models, then prior coursework in software engineering and experience on a large software project is strongly recommended, as is some prior training or experience with experimental research design methods.

Module Content

Outline

I. Introduction
  1. Historical origins for system life-cycle models
  2. Software life-cycle activities
  3. What is a software life-cycle model?
  4. How can software life-cycle models be used?
  5. What is a software process model?
  6. Evolutionistic vs. Evolutionary Models
  7. The neglected activities of software evolution

II. Traditional Software Life-Cycle Models
  1. Classic Software Life-Cycle
  2. Stepwise Refinement and Iterative Enhancement
  3. Incremental Release
  4. Industrial Practices and Military Standards

III. Alternative Life-Cycle Models
  1. Software Product Development Models
    a. Prototyping
    b. Assembling Reusable Components
    c. Application Generation
    d. Program Evolution Models
  2. Software Production Process Models
    a. Non-Operational Process Models
    b. Operational Process Models
  3. Software Production Setting Models
    a. Software project management process models
    b. Organizational software development models
    c. Customer resource life-cycle models
    d. Software technology transfer and transition models
    e. Other models for the organization of system production and manufacturing

IV. Where do tools and techniques fit into the models?
  1. Life-Cycle support mechanisms
  2. Process support mechanisms

V. Evaluating Life-Cycle Models and Methodologies
  1. Comparative evaluation of life-cycle and process methodologies
  2. Research problems and opportunities

VI. Customizable Life-Cycle Process Models
  1. Selecting an Existing Model
  2. Customizing your own Model
  3. Using Process Metrics and Empirical Measurements
  4. Staffing the Life-Cycle Process Modeling Activity

Annotated Outline

I. Introduction

Software evolution represents the cycle of activities involved in the development, use, and maintenance of software systems. Software systems come and go through a series of passages that account for their inception, initial development, productive operation, upkeep, and retirement from one generation to another. Material in this section identifies the historical origins of the software life-cycle concept, the general activities included, the similarities and differences between software life-cycle and software process models, and related issues. This section is therefore appropriate for all students of software engineering.

1. Historical origins for system life-cycle models

Originally, system life-cycle models emerged in the fields of evolutionary biology and cybernetics. In turn, models of software evolution date back to the earliest projects developing large software systems [Benington56, Hosier61, Royce70]. Overall, the apparent purpose of these software life-cycle models was to provide an abstract scheme accounting for the "natural" or engineered development of software systems. Such a scheme could therefore serve as a basis for planning, organizing, staffing, coordinating, budgeting, and directing software development activities.

2. Software life-cycle activities

For more than a decade, many descriptions of the classic software life-cycle (often referred to as "the waterfall chart") have appeared (e.g., [Royce70, Boehm76, Distaso80, Scacchi84, Fairley85]) and have usually included some version of the following activities:

- *System Initiation/Adoption:* identifies

where systems come from. In most situations, new systems replace or supplement existing processing mechanisms whether they were previously automated, manual, or informal.

- *Requirement Analysis and Specification*: identifies the problems a new software system is supposed to solve.
- *Functional Specification or Prototyping*: identifies and potentially formalizes the objects of computation, their attributes and relationships, the operations that transform these objects, the constraints that restrict system behavior, and so forth.
- *Partition and Selection* (Build vs. Buy vs. Reuse): given requirements and functional specifications, divides the system into managable pieces that denote logical subsystems, then determines whether new, existing, or reusable software systems correspond to the needed pieces.
- *Architectural Configuration Specification*: defines the interconnection and resource interfaces between system modules in ways suitable for their detailed design and overall configuration management.
- *Detailed Component Design Specification*: defines the procedural methods through which each module's data resources are transformed from required inputs to provided outputs.
- *Component Implementation and Debugging*: codifies the preceding specifications into operational source code implementations and validates their basic operation.
- *Software Integration and Testing*: affirms and sustains the overall integrity of the software system architectural configuration through verifying the consistency and completeness of implemented modules, verifying the resource interfaces and interconnections against their specifications, and validating the performance of the system and subsystems against their requirements.
- *Documentation Revision and System Delivery*: packaging and rationalizing recorded system development description into systematic documents and user guides, all in a form suitable for dissemination and system support.
- *Training and Use*: providing system users with instructional aids and guidance for understanding the system's capabilities and limits in order to effectively use the system.
- *Software Maintenance*: sustaining the useful operation of a system in its host/target environment by providing requested functional enhancements, repairs, performance improvements, and conversions.

3. What is a software life-cycle model?

A software life-cycle model is either a descriptive or prescriptive characterization of software evolution. Typically, it is easier to articulate a prescriptive life-cycle model for how software systems should be developed. This is possible since most such models are intuitive. This means that many software development details can be ignored, glossed over, or generalized. This, of course, should raise concern for the relative validity and robustness of such life-cycle models when developing different kinds of application systems in different kinds of development settings. Descriptive life-cycle models, on the other hand, characterize how software systems are actually developed. As such, they are less common and more difficult to articulate for an obvious reason: one must observe or collect data throughout the development of a software system, a period of elapsed time usually measured in years. Also, descriptive models are specific to the systems observed, and only generalizable through systematic analysis. Therefore, this suggests the prescriptive software life-cycle models will dominate attention until a sufficient base of observational data is available to articulate empirically grounded descriptive life-cycle models.

4. How can software life-cycle models be used?

Some of the ways these models can be used include:

- as a means to organize, plan, staff, budget, schedule and manage software project work over organizational time, space, and computing environments.
- as prescriptive outlines for what documents to produce for delivery to client.
- as a basis for determining what software engineering tools and methodologies will be most appropriate to support different life-cycle activities.
- as frameworks for analyzing or estimating patterns of resource allocation and consumption during the software life-cycle [Boehm81a].
- as comparative descriptive or prescriptive accounts for how software systems come to be the way they are.
- as a basis for conducting empirical studies to determine what affects software productivity, cost, and overall quality.

5. What is a software process model?

A software process model often represents a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution [Potts84, Wileden86, Dowson86]. Such models can be used to develop more precise and formalized descriptions of software life-cycle activities. Their power emerges from their use of a sufficiently rich notation, syntax, or semantics, often suitable for computational processing.

Software process networks can be viewed as representing methodical *task chains*. Task chains structure the transformation of computational entities through a passage of sequence of actions that denote each process activity. Task chains are *idealized plans* of what actions should be accomplished, and in what order. For example, a task chain for the activity of object-oriented software design might include the following task actions:

- Develop an informal narrative specification of the system.
- Identify the objects and their attributes.
- Identify the operations on the objects.
- Identify the interfaces between objects, attributes, or operations.
- Implement the operations.

Task chains join or split into other task chains resulting in an overall *production lattice*. The production lattice represents the "organizational system" that transforms raw computational, cognitive, and other organizational resources into assembled, integrated software systems. The production lattice therefore represents the structure of how a software system is developed, used, and maintained. However, tasks chains and actions are never sufficiently described to anticipate all possible contingencies or problems that can emerge in the real-world of software development. Thus any software production lattice will in some way realize only an approximate or incomplete description of software development. As such, *articulation work* will be performed when a task chain is inadequate or breaks down. The articulation work then represents a non-deterministic sequence of actions taken to restore progress on the disarticulated task chain, or else to shift the flow of productive work onto some other task chain [Bendifallah87].

6. Evolutionistic vs. Evolutionary Models

Every model of software evolution makes certain assumptions about the meaning of evolution. In one such analysis of these assumptions, two distinct views are apparent: *evolutionistic* models focus on the direction of change in terms of progress through a series of stages eventually leading to some final stage; *evolutionary* models, on the other hand, focus attention to the mechanisms and processes that change systems [King84]. Evolutionistic models are often intuitive and useful as organizing frameworks for managing and tooling software development efforts. But they are poor predictors of why certain changes are made to a system, and why systems evolve in similar or different ways [Bendifallah87]. Evolutionary models are concerned less with the stage of development, and more with the technological mechanisms and organizational processes that guide the emergence of a system over space and time. As such, it should become apparent that the traditional models are evolutionistic, while most of the alternative models are evolutionary.

7. The neglected activities of software evolution

Three activities critical to the overall evolution of software systems are maintenance, technology transfer, and evaluation. However, these activities are often inadequately addressed in most models of software evolution. Thus, any model of software evolution should be examined to see to what extent it addresses these activities.

Software maintenance often seems to be described as just another activity in the evolution of software. However, many studies indicate that software systems spend most of their useful life in this activity [Boehm76, Boehm81a]. A reasonable examination of the activity indicates that maintenance represents ongoing incremental iterations through the life-cycle activities that precede it [Basili75]. These iterations are an effective way to incorporate new functional enhancements, remove errors, restructure code, improve system performance, or convert a system to run in another environment. Subsequently, software maintenance activities represent micro-level passages through the life-cycle. However, it is also clear that many other technical and organizational circumstances profoundly shape the evolution of a software system and its host environment [Lehman86a, Bendifallah87]. Thus, every software life-cycle or process model should be closely examined to see to what extent its accounts for what happens to a software system during most of its sustained operation.

Concerns for system installation and support need to be addressed during the earliest stages of software evolution. These concerns eventually become the basis for determining the success or failure of software system use and maintenance activities. Early and sustained involvement of users in system development is one of the most direct ways to affect a successful software technology transfer. Failure to involve users is one of the most common reasons why system use and maintenance is troublesome. Thus, any model of software evolution can be evaluated according to the extent that it accommodates activities or mechanisms that encourage system developers and users to cooperate.

Evaluating the evolution of software systems helps determine which development activities or actions could be made more effective. Many models of software evolution do not address how system developers (or users) should evaluate their practices to determine which of their activities could be improved or restructured. Technical reviews and software inspections often focus attention to how to improve the quality of the software products being developed, while the organizational and technological processes leading to these products receive less attention. Evaluating development activities also implies that both the analytical skills and tools are available to a development group. Thus, models of software evolution can also be scrutinized to determine to what extent they incorporate or structure development activities in ways that provide developers with the means to evaluate the effectiveness of the engineering practices.

Finally, one important purpose of evaluating local practices for software evolution is to identify opportunities where new technologies can be inserted. In many situations, new software engineering tools, techniques, or management strategies are introduced during the middle of a system development effort. How do such introductions impact existing practices? What consequences do such introductions have on the maintainability of systems currently in use or in development? Software maintenance, technology transfer, and process evaluation are each critical to the effective evolution of software systems, as is their effect on each other. Thus, they should be treated collectively, and in turn, models of software evolution can be reviewed in terms of how well they address this collective.

II. Traditional Software Life-Cycle Models

These models of software evolution have been with us in some cases since the earliest days of software engineering. The classic software life-cycle (or "waterfall" model) and stepwise refinement are widely instantiated in just about all books on modern programming practice and software engineering. The incremental release model is closely related to industrial practices where it most often occurs. Military standards have also reified certain forms of the classic life-cycle model into required practice for government contractors. Since of these life-cycle models have been in use for some time, we refer to them as the traditional models, and identify each below.

1. Classic Software Life-Cycle

The classic software life-cycle is often represented as a simple waterfall software phase model, where software evolution proceeds through an orderly sequence of transitions from one phase to the next in linear order. Such models resemble finite state machine descriptions of software evolution. However, such models have been perhaps most useful in helping to structure and manage large software development projects in organizational settings.

2. Stepwise Refinement and Iterative Enhancement

This model advocates developing software systems through ongoing refinement and enhancement of high-level system specifications into source code components [Wirth71, Basili75]. These models have been most effective in helping to teach individual programmers how to organize their software development work. Many interpretations of the classic software life cycle subsume this approach within their design and implementations.

3. Incremental Release

This model advocates developing systems by first providing essential operating functions, then providing system users with improved and more capable versions of a system at regular intervals [Tully84]. This model combines the classic software life-cycle with iterative enhancement at the level of system development organization. It also provides a way to periodically distribute software maintenance updates and services to dispersed user communities. This in turn accommodates the provision of standard software maintenance contracts. It is therefore a popular model of software evolution used by commercial firms.

4. Industrial Practices and Military Standards

Industrial firms often adopt some variation of the classic model as the basis of the software development practice [Royce70, Boehm76, Distaso80, Scacchi84, Scacchi86a]. Many government contractors organize their activities according to military standard life-cycle models such as that embodied in MIL-STD-2167 [MIL-STD-2167]. Such standards outline not only some variation of the classic life-cycle activities, but they also the content of documents required by clients who procure either software systems or complex mechanisms with embedded software systems. These standards are also intended to be compatible with provision of software quality assurance, configuration management, and independent verification and validation services in a multi-contractor development project. More recent progress in industrial practice appears in [Humphrey85, Radice85, Yacobellis84].

III. Alternative Life-Cycle Models

There are at least three alternative sets of models of software evolution. These models are alternatives to the traditional software life-cycle models. These three sets focus attention on either the *products*, *production processes*, or *production settings* associated with software evolution. Since these models are not in widespread practice, discussion of them is appropriate at an intermediate level of coursework, while in-depth re-

view is appropriate at an advanced level. However, all students of software engineering should have an overview of models of program evolution and software technology transfer.

1. Software Product Development Models

Software product development models represent an evolutionary extension to the traditional software life-cycle models. The extensions arose due to the availability of new software development technologies such as software prototyping languages and environments, reusable software, and application generators. Each of these technologies seeks to enable the creation of executable software implementations either earlier in the life-cycle, or more rapidly but with reduced functionality. Discussion of these models is most appropriate when such technologies are available for use or experimentation.

a. Prototyping

Prototyping is a technique for providing a reduced functionality version of a software system early in its development [Balzer82, Boehm84, Budde84, Hekmatpour87]. Prototyping technologies usually accept some form of software functional specifications as input, which in turn are either simulated, analyzed, or directly executed. As such, these technologies allow software design activities to be initially skipped or glossed over. In turn, these technologies can allow developers to rapidly construct early or primitive versions of software systems that users can evaluate. These user evaluations can then be incorporated as feedback to refine the emerging system specifications and designs. Further, depending on the prototyping technology, the complete working system can be developed through a continually process of revising/refining the input specifications. This has the advantage of always providing a working version of the developing system, while redefining software design and testing activities to input specification refinement and execution. Alternatively, other prototyping approaches are best suited for developing "throwaway" (demonstration only) systems, or for building prototypes by reusing part/all of some existing software systems. Two collections of papers on the subject can be found in [Sen82, Budde84].

b. Assembling Reusable Components

The basic approach of reusablity is to configure and specialize pre-existing software components into viable application systems [Biggerstaff84, Neighbors84, Goguen86]. However, the granularity of the components (i.e., size, complexity, functional capability) vary greatly across different approaches. Most approaches attempt to utilize components similar to common data structures with algorithms for their manipulation: *small-grain components*. However, the use/reuse of small-grain components in and of themselves does not constitute a distinct approach to software evolution. Other approaches attempt to utilize components resembling functionally complete systems or subsystems (e.g., user interface management system): *large-grain components*. The use/reuse of large-grain components does appear to be an alternative approach to developing software systems, and thus is an area of active research. There are probably many ways to utilize reusable software components in evolving software systems. However, cited studies suggest their initial use during architectural or component design specification as a way to speed implementation. They might also be used for prototyping purposes if a suitable software prototyping technology is available.

c. Application Generation

Application generation is an approach to software development similar to reuse of parameterized, large-grain software components. Such components are specialized to an application domain via a formalized specification language used as input to the application generator. Common examples provide standardized interfaces to database management system applications, and include generators for reports, graphics, user interfaces, and application-specific editors. Application generators give rise to a model of software evolution whereby software design activities are either almost eliminated, or reduced to a database design problem. Similarly, users of application generators are usually expected to provide input specifications and application maintenance services. These capabilities are possible since the generators can usually only produce software systems specific to a small number of similar application domains, and usually those that depend on a database management system [Horowitz85].

d. Program Evolution Models

In contrast to the preceding three models, Lehman and Belady sought to develop a descriptive model of software product evolution. They conducted a series of studies of the evolution of large software systems at IBM during the 1970's [Lehman85]. Based on their investigations, they identify five properties that characterize the evolution of large software systems. These are:

1. *Continuing change*: a large software system undergoes continuing change or becomes progressively less useful
2. *Increasing complexity*: as a software system evolves, its complexity increases unless work is done to maintain or reduce it

3. *Fundamental law of program evolution*: program evolution, programming process, and global measures of project and system attributes are statistically self-regulating with determinable trends and invariances
4. *Invariant work rate*: the rate of global activity in a large software project is statistically invariant
5. *Incremental growth limit*: during the active life of a large program, the volume of modifications made to successive releases is statistically invariant.

However, it is important to observe that these are global properties of large software systems, not causal mechanisms of software evolution.

2. Software Production Process Models

There are two kinds of software production process models: non-operational and operational. Both are software process models. The difference between the two primarily stems from the fact that the operational models can be viewed as programs: programs that implement a particular regimen of software engineering and evolution. Non-operational models, on the other hand, denote conceptual approaches that have not yet been sufficiently articulated in a form suitable for codification.

a. Non-Operational Process Models

(i) The Spiral Model

The spiral model of software development and evolution represents a *risk-driven* approach to software process analysis and structuring [Boehm86]. The approach incorporates elements of specification-driven and prototype-driven process methods. It does so by representing iterative development cycles in a spiral manner, with inner cycles denoting early analysis and prototyping, and outer cycles denoting the classic system life-cycle. The radial dimension denotes cumulative development costs, and the angular dimension denotes progress made in accomplishing each development spiral. Risk analysis, which seeks to identify situations which might cause a development effort to fail or go over budget/schedule, occurs during each spiral cycle. In each cycle, risk analysis represents roughly the same amount of angular displacement, while the displaced sweep volume denotes increasing levels of effort required for risk analysis. System development in this model therefore spirals out only so far as needed according to the risk that must be managed.

(ii) Continuous Transformation Models

These models propose a process whereby software systems are developed through an ongoing series of transformations of problem statements into abstract specifications into concrete implementations [Wirth71, Basili75, Bauer76, Balzer81]. Lehman, Stenning, and Turski, for example, propose a scheme whereby there is no traditional life-cycle nor separate stages, but instead an ongoing series of reifying transformations of abstract specifications into more concrete programs [Lehman84a, Lehman84b]. In this sense then, problem statements and software systems can emerge somewhat together, and thus can continue to co-evolve.

Continuous transformation models also accommodate the interests of software formalists who seek the precise statement of formal properties of software system specifications. Accordingly, the specified formalisms can be mathematically transformed into properties that a source implementation should satisfy. The potential for automating such models is apparent, but it still the subject of ongoing research (and addressed below).

(iii) Miscellaneous Process Models

Many variations of the non-operational life-cycle and process models have been proposed, and appear in the proceedings of the three software process workshops [Potts84, Wileden86, Dowson86]. These include fully interconnected life-cycle models which accommodate transitions between any two phases subject to satisfaction of their pre- and post-conditions, as well as compound variations on the traditional life cycle and continuous transformation models. However, the cited reports generally indicate that in general most software process models are analytical or theoretical, so little experience with these models has been reported.

b. Operational Process Models

(i) Operational specifications for rapid prototyping

The operational approach to software development assumes the existence of a formal specification language and processing environment [Bauer76, Balzer82, Balzer83a, Zave84]. Specifications in the language are "coded" and when possible constitute a functional prototype of the specified system. When such specifications can be developed and processed incrementally, then the resulting system prototypes can be refined and evolved into functionally more complete systems, and are always operational during their development. Variations within this approach represent either efforts where the prototype is the end sought,

or where specified prototypes are kept operational but refined into a complete system.

(ii) Software process automation and programming

Process automation and programming are concerned with developing "formal" specifications of how a (family of) software system(s) should be developed. Such specifications therefore should provide an account for an organization and description of the various software production task chains, how they interrelate, when then can iterate, etc. as well as what software tools to use to support different tasks, and how these tools should be used [Hoffnagel85, Huseth86, Osterweil87]. See [Lehman87] and [Curtis87] for provocative reviews of the potential and limitations of current proposals for software process automation and programming.

(iii) Knowledge-based software automation

This model attempts to take process automation to its limits by assuming that process specifications can be used directly to develop software systems, and to configure development environments to support the production tasks at hand. The common approach is to seek to automate the continuous transformation model. In turn, this implies an automated environment capable of recording the formalized development of operational specifications, successively transforming and refining these specifications into an implemented system, assimilating maintenance requests by inserting the new/enhanced specifications into the current development derivation, then replaying the revised development toward implementation [Bauer76, Balzer83b, Balzer85]. However, current progress has been limited to demonstrating such mechanisms and specifications to narrowly-defined software coding, maintenance, project communication and management tasks [Balzer83b, Balzer85, Cheatham86, Polak86, Kedzierski84, Sathi85, Sathi86].

3. Software Production Setting Models

In contrast to product or production process models of software evolution, production setting models draw attention to organizational and management strategies for developing and evolving software systems. With rare exception, such models are non-operational. As such, the focus is less technological and more strategic. But it should become clear that such strategies do affect what software products get developed and how software production processes will be organized.

Also, note that the last entry in this section on other models of system production and manufacturing is marked optional, and thus is perhaps most appropriate at an advanced level.

a. Software project management process models

In parallel to (or on top of) a software development effort, there is normally a management superstructure to configure the effort. This structure also represents a cycle of activities for which project managers assume the responsibility. The activities include project planning, budgeting and controlling resources, staffing, dividing and coordinating staff, scheduling deliverables, directing and evaluating (measuring) progress, and intervening to resolve conflicts, breakdowns, or resource distribution anomalies [Thayer81, Scacchi84, Kedzierski84, Radice85, Humphrey85].

b. Organizational software development models

Software development projects are plagued with many recurring organizational dilemmas which can slow progress. Experienced managers recognize these dilemmas and develop strategies for mitigating or resolving their adverse effects. Such strategies therefore form an informal model for how to manage software development throughout its life-cycle. See [Kling80, Kidder81, Kling82, Scacchi84, Gasser86, Curtis87] as well as [Liker86].

c. Customer resource life-cycle models

With the help of information (i.e., software) systems, a company can become more competitive in all phases of its customer relationships [Ives84, Wiseman85]. The customer resource life-cycle (CRLC) model is claimed to make it possible for such companies to determine when opportunities exist for *strategic applications*. Such applications change a firm's product line or the way a firm competes in its industry. The CRLC model also indicates what specific application systems should be developed.

The CRLC model is based on the following premises: the products that an organization provides to its customers are, from the customer's viewpoint, supporting resources. A customer then goes through a cycle of resource definition, adoption, implementation and use. This can require a substantial investment in time, effort, and management attention. But if the supplier organization can assist the customer in managing this resource life-cycle, the supplier may then be able to differentiate itself from its competitors via enhanced customer service or direct cost savings. Thus, the supplier organization should seek to develop and apply software systems that support the customer's resource life-cycle. [Ives84] and

[Wiseman85] describe two approaches for articulating CRLC models and identifying strategic software system applications to support them.

The purpose of examining such models is to observe that forces and opportunities in a marketplace such as customer relationships, corporate strategy, and competitive advantage can help determine the evolution of certain kinds of software systems.

d. Software technology transfer and transition models

The software innovation life-cycle circumscribes the technological and organizational passage of software system technologies. This life-cycle therefore includes the activities that represent the transfer and transition of a software system from its producers to its consumers. This life-cycle includes the following activities [Redwine85, Scacchi86b]:

- *Invention and prototyping*: software research and exploratory prototyping
- *Product development*: the software development life-cycle
- *Diffusion*: packaging and marketing systems in a form suitable for widespread dissemination and use
- *Adoption and Acquisition*: deciding to commit organizational resources to get new systems installed
- *Implementation*: actions performed to assimilate newly acquired systems into existing work and computing arrangements
- *Routinization*: using implemented systems in ways that seem inevitable and part of standard procedures
- *Evolution*: sustaining the equilibrium of routine use for systems embedded in community of organizational settings through enhancements, restructuring, debugging, conversions, and replacements with newer systems.

Available research indicates that progress through the software innovation life-cycle can take 7-20 years for major software technologies (e.g., Unix, expert systems, programming environments, Ada) [Redwine85]. Thus, moving a software development organization to a new technology can take a long time and great effort. Research also indicates that most software innovations (small or large) fail to get properly implemented, and thus result in wasted effort and resources [Scacchi86b]. The failure here is generally not technical, but instead primarily organizational. Thus, organizational circumstances and the people who animate them have far greater affect in determining the successful use and evolution of a software innovation, than the innovation's technical merit. However, software technology transfer is an area requiring much more research.

e. Other models for the organization of system production and manufacturing

(This section is optional.) What other kinds of models of software production might be possible? If we look to see how other technological systems are developed, we find the following sort of models for system production:

- *Ad-hoc problem solving, tinkering, and articulation work*: the weakest model of production is when people approach a development effort with little or no preparation or task chain plan at hand, and thus rely solely upon their skill, ad hoc tools, or the loosely coordinated efforts of others get them through. It is situation specific, and driven by accommodations to local circumstances. It is therefore perhaps the most widely practiced form of production and system evolution.
- *Group project*: software life-cycle and process efforts are usually realized one at a time, with every system being treated somewhat uniquely. Thus such efforts are often organized as group projects.
- *Custom job shop*: job shops take on only particular kinds of group project work, due to more substantial investment in tooling and production skill/technique refinement.
- *Batched production*: provides the customization of job shops but for a larger production volume. Subsystems in development are configured on jigs that can either be brought to workers and production tools, or that tools and workers can be brought to the workpieces or subsystems.
- *Pipeline*: when system development requires the customization of job shops or the specialization of volume of batched production, while at the same time allowing for concurrent development sequences of subsystems.
- *Flexible manufacturing systems*: seek to provide the customization capabilities of job shops, while relying upon advanced automation to allow economies of scale, task standardization, and delivery of workpieces of transfers lines realized through rapidly reconfigurable

workstation tooling and process programming. Recent proposals for "software factories" have adopted a variation of this model [Scacchi87].

- *Transfer (assembly) lines*: when raw input resources or semi-finished subassemblies can be moved through a network of single action workcells, then transfer lines are appropriate.
- *Continuous process control*: when the rate or volume of uniform raw input resources and finished output products can be made continuous and automatically variable, then a continuous process control form of production is appropriate. Oil refining is an example of such a process, with crude oil from wells as input, and petroleum products (gasoline, kerosene, multi-grade motor oil) as outputs. Whether software can be produced in such a manner is unlikely at this time.

IV. Where do tools and techniques fit into the models?

Given the diversity of software life-cycle and process models, where do software engineering tools and techniques fit into the picture? This section briefly identifies some of the places where different software engineering technologies can be matched to certain models. Another way to look at this section might be to look instead at what software engineering technologies might be available in an individual setting, then seek a model of software evolution that is compatible.

1. Life-Cycle support mechanisms

Most of the traditional life-cycle models are decomposed as stages. These stages then provide boundaries whereby software engineering technologies are targeted. Thus, we find engineering techniques or methods (e.g., Yourdon structured design, TRW's software requirements engineering methodology (SREM)) being targeted to support different life-cycle stages, and tools (e.g., TRW's requirements engineering and verification system (REVS)) targeted to support the associated activities. However, there are very few, if any, package of tools and techniques that purport to provide integrated support for engineering software systems throughout their life-cycle [Scacchi87]. Perhaps this is a shortcoming of the traditional models, perhaps indicative that the integration required is too substantial to justify its expected costs or benefits, or prehaps the necessary technology is still in its infancy. Thus, at present, we are more likely to find ad-hoc or loose collections of software engineering tools and techniques that provide partial support for software life-cycle engineering.

2. Process support mechanisms

There are at least three kinds of software process support mechanisms:

- *Process articulation technologies* denote the prototyping, reusable software, and application generator languages and environments for rapidly developing new software systems.
- *Process measurement and analysis technologies* denote the questionnaire, survey, or performance monitoring instruments used to collect quantifiable data on the evolving characteristics of software products and processes. Collected data can in turn be analyzed with statistical tools to determine descriptive and inferential relationships within the data. These relationships can then be interpreted as indicators for where to make changes in current practices through a restructuring of work/resources, or through the introduction of new software engineering technologies. Such measurement and analysis technologies can therefore accommodate process refinements that improve its overall performance and product quality.
- *Computational process models* denote formalized descriptions of software development activities in a form suitable for automated processing. Such models are envisioned to eventually be strongly coupled to available software engineering tools and techniques in ways that allow their configuration and use to be programmed. However, at present, such models serve to help articulate more precise descriptions for how to conduct different software engineering activities.

V. Evaluating Life-Cycle Models and Methodologies

Given the diversity of software life-cycle and process models, how do we decide which if any is best, or should be the one to follow? Answering this question requires further research. Therefore, material in this section is perhaps most appropriate at an advanced level.

1. Comparative evaluation of life-cycle and process methodologies

As noted in Section I, descriptive life-cycle models require the empirical study of software evolution products and processes. Therefore, how should such a study be designed to realize useful, generalizable results?

Basically, empirical studies of actual software life-cycles or processes should ultimately lead to models

of evolution with testable predictions [Curtis80, Basili86]. Such models in turn must therefore be applicable across different sets of comparable data. This means that such studies must use measurements that are reliable, valid, and stable. Reliability refers to the extent that the measures are accurate and repeatable. Validity indicates whether the measured values of process variables are in fact correct. Stability denotes that the instrument measures one or more process variables in a consistent manner across different data sets [Curtis80].

However, most statistical instruments are geared for snapshot studies where certain variables can be controlled, while others are independent. Lehman and Belady use such instruments in their evaluation of large software system attributes [Lehman85]. Their study uses data collected over periodic intervals for a sample of large software systems over a number of years. However, their results only make strong predictions about *global* program evolution dynamics. That is, they cannot predict what will happen at different life-cycle stages, in different circumstances, or for different kinds of software systems. To make such predictions requires a different kind of study.

[vandenBosch82] and [Curtis87] propose two alternative approaches to studying software evolution. Both rely upon long-term field studies of a sample of software efforts in different organizational settings. Their approach is targeted to constructing a framework for discovering the mechanisms and organizational processes that shape software evolution with a comparative study sample. The generality of the results they derive can thus be assessed in terms of their sample space.

[Kelly87] provides an informing comparative analysis of four methods for the design of real-time software systems. Although his investigation does not compare models of software evolution, his framework is suggestive of what might be accomplished through comparative analysis of such models.

Other approaches that report on the comparative analysis of software evolution activities and outcomes can be found elsewhere [Kling80, Basili81, Boehm81b].

2. Research problems and opportunities

As should be apparent, most of the alternative models of software evolution are relatively new, and in need of improvement and empirical grounding. It should however also be clear that such matters require research investigations. Prescriptive models can be easy to come by, whereas descriptive models require systematic research regimens which can be costly. Nonetheless, there are many opportunities to further develop, combine, or refute any of the alternative models of software evolution. Comparative research design methods, data sampling, collection, and analysis are all critical topics that require careful articulation and scrutiny [Basili86]. And each of the alternative models, whether focusing attention to either software products, production processes, production settings, or their combination can ideally draw upon descriptive studies as the basis of their prescriptions. Thus, we are at a point where empirical studies of software life-cycle or process models (or their components) are needed, and likely to be very influential if performed systematically and rigorously.

Therefore, for advanced level students, it is appropriate to devote some attention to the problem of designing a set of experiments intended to substantiate or refute a model of software evolution, where critical attention should then be devoted to evaluating the quality and practicality (i.e., time, effort, and resources required) of the proposed research.

VI. Customizable Life-Cycle Process Models

Given the emerging plethora of models of software evolution, how does one choose which model to put into practice? This will be a recurring question in the absence of empirical support for the value of one model over others. We can choose whether to select an existing model, or else to develop a custom model. Either way, the purpose of having a model is to use it to organize software development efforts in a more effective, more productive way. But this is not a one-shot undertaking. Instead, a model of software evolution is likely to be most informing when not only used to prescribe software development organization, but also when used to continually measure, tune, and refine the organization to be more productive, risk-reducing, and quality driven [Humphrey85, Radice85, Basili87].

1. Selecting an Existing Model

Choosing the one that's right for an individual software project and organization is the basic concern. At this time, we can make no specific recommendation for which model is best in different circumstances. The choice is therefore open-ended. However, we might expect to see the following kinds of choices being made with respect to existing models: Generally, most software development organizations are likely to adopt one of the traditional life-cycle models. Then they will act to customize it to be compatible with other organizational policies, procedures, and market conditions. Software research organizations will more likely adopt an alternative model, since they are likely to be interested in evaluating the potential of emerging software technologies. When development organizations adopt software technologies more closely aligned to the alternative models (e.g., reusable components, rapid prototyping), they may try to use them either experimentally, or to shoehorn them into a traditional life-

cycle model, with many evolutionary activities kept informal and undocumented. Alternatively, another strategy to follow is to do what some similar organization has done, and to use the model they employ. Studies published by researchers at IBM and AT&T Bell Laboratories are often influential in this regard [Humphrey85, Radice85, Yacobellis84].

2. Customizing your own Model

[Basili87] can be recognized as one of the foremost advocates for developing a custom life-cycle process model for each project and organization. Empirical studies of software development seem to indicate that life-cycle process modeling will be most effective and have the greatest benefit if practiced as a regular activity. Process metrics and measurements need to be regularly applied to capture data on the effectiveness of current process activities. As suggested above, it seems likely that at this time, the conservative strategy will be to adopt a traditional life-cycle model and then seek to modify or extend it to accommodate new software product or production process technologies. However, it seems just as likely that software development efforts that adopt software product, production process and production setting concerns into a comprehensive model may have the greatest potential for realizing substantial improvement in software productivity, quality, and cost reduction [Scacchi86c].

3. Using Process Metrics and Empirical Measurements

One important purpose of building or buying a process model is to be able to apply it to current software development projects in order to improve their productivity, quality, and cost-effectiveness [Humphrey85, Radice85]. The models therefore provide a basis for instrumenting the software process in ways that potentially reveal where development activities are less effective, where resource bottlenecks occur, and where management interventions or new technologies could have a beneficial impact [Basili87, Yacobellis84]. [Scacchi86c] go so far as to advocate a radical approach involving the application of knowledge-based technologies for modeling and simulating software product, production process, and production setting interactions based upon empirical data (i.e., knowledge) acquired through questionnaire surveys, staff interviews, observations, and online monitoring systems. Such an approach is clearly within the realm of basic research, but perhaps indicative of the interest in developing high-potential, customizable models of software evolution.

4. Staffing the Life-Cycle Process Modeling Activity

Ideally, the staff candidate best equipped to organize or analyze an organizational's model of software evolution is one who has mastered the range of material outlined in this curriculum module. That is, a staff member who has only had an introductory or even intermediate level exposure to this material is not likely to perform software life-cycle or process modeling competently. Large software development organizations with dozens, hundreds, or even thousands of software developers are likely to rely upon one or more staff members with a reasonably strong background in local software development practices and experimental research skills. This suggests that such staff are therefore likely to possess the equivalent of a masters or doctoral degree software engineering or experimental computer science. In particular, a strong familiarity with experimental research methods, sampling strategies, questionnaire design, survey analysis, statistical data analysis packages, and emerging software technologies are the appropriate prerequisites. Simply put, this is not a job for any software engineer, but instead a job for software engineer (or industrial engineer) with advanced training and experience in experimental research tools and techniques.

Glossary

articulation work

a non-deterministic series of actions taken by people in response to foul-ups, breakdowns, mistakes, resource bottlenecks, or other unexpected circumstances that cause planned task chains to disarticulate. Hacking together software kludges in response to system glitches is a frequently observed form of articulation work that occurs during software evolution.

evolutionary models

represent software evolution in terms that focus attention to the *mechanisms* that give rise to changes made in a system. Such models seek to account for how and why software systems emerge the way they do. Systems evolve not so much according to prescriptive stages, but rather in response to the actions people take to make the system fit their circumstantial needs. Thus, when circumstances change, people will seek opportunities to change the system.

evolutionist models

represent software evolution in terms that focus attention to the *direction* of changes made to systems. Such models seek to explain the logic of development typically in the form of stages the follow one another, where each stage is the

precursor for the next one, and ultimately toward a final state (e.g., classic waterfall life cycle model).

production lattice

the intersecting network of task chains that collectively denote the structure of software development activities.

software evolution

the collection of software life cycle or process activities that cause systems to be produced and consumed.

software life cycle

a typical sequence of phased activities that represent the various stages of engineering through which software system pass.

software process

the network of object states and transitional events that represent the production of a software system in a form suitable for computational encoding and processing.

task chain

a planned, possibly iterative, sequence of actions taken by people in order to transform raw production resources into consumable product resources.

Teaching Considerations

This module collects and organizes a body of knowledge about software evolution for the first time. The material has not been taught in this form, and therefore suggestions for effective teaching have not been developed. However, prior experience in teaching part of this material suggests the use of case studies of large system development projects as an excellent source material for study and review. For an advanced level course, a book such as *The Soul of a New Machine* by Tracy Kidder is an excellent choice. For an intermediate level of coverage, individual case studies provide suitable source material that can introduce students to the interrelationship of software products, production processes, and production settings as sources of influence in system evolution. A subsequent release of this module will include suggestions from instructors who have taught the material.

Bibliography

Balzer81
Balzer, R. "Transformational Implementation: An Example." *IEEE Trans. Software Eng. SE-7*, 1 (1981), 3-14.

Abstract: A system for mechanically transforming formal program specifications into efficient implementations under interactive user control is described and illustrated through a detailed example. The potential benefits and problems of this approach to software implementation are discussed.

Balzer82
Balzer, R., N. Goldman, and D. Wile. "Operational Specifications as the Basis for Rapid Prototyping." *ACM Software Engineering Notes 7*, 5 (1982), 3-16.

Among the first papers to assert the desirability of rapidly developing software systems through the use of operational process and database-oriented specifications and supporting environment. Also asserts the importance of being able to specify hence prototype descriptions of the user and computational environments in which the emerging system is to operate as an equally important component.

Balzer83a
Balzer, R., D. Cohen, M. Feather, N. Goldman W. Swartout, and D. Wile. "Operational Specifications as the Basis for Specification Validation." In *Theory and Practice of Software Technology*, Ferrari, Bolognani, and Goguen, eds. Amsterdam: North-Holland, 1983.

Abstract: This paper describes a set of freedoms which both simplify the task of specifing systems and make the resulting specification more comprehensible. These freedoms eliminate the need, in specific areas, to consider: the mechanisms for accomplishing certain capabilities, the careful coordination and integration of separate operations, the cost of those operations, and other detailed concerns which characterize implementation.

These freedoms are partitioned into the areas of efficiency, method, and data, and providing them has resulted in a novel formal specification language, Gist. The main features of this language are described in terms of the freedoms it affords. An overview of the language is then presented together with an example of its use to specify the behavior of a real system.

Balzer83b
Balzer, R., T. Cheatham, and C. Green. "Software Technology in the 1990's: Using a New Paradigm." *Computer 16*, 11 (Nov. 1983), 39-46.

Proposes a radical alternative to traditional approaches to software development and evolution through the use of knowledge-based operational specification languages and tools. The approach seeks to introduce and rely upon a degree of automation in software development far beyond what is available at present. However, it is also clear that the approach is inherently long-term in its orientation; thus, it may take a decade or more before it is fully implemented in a form suitable for large-scale experimentaion.

Balzer85
Balzer, R. "A 15 Year Perspective on Automatic Programming." *IEEE Trans. Software Eng. SE-11*, 11 (Nov. 1985), 1257-1267.

Abstract: Automatic programming consists not only of an automatic compiler, but also some means of acquiring the high-level specification to be compiled, some means of determining that it is the intended specification, and some (interactive) means of translating this high-level specification into a lower-level one which can be automatically compiled.

We have been working on this extended automatic programming problem for nearly 15 years, and this paper presents our perspective and approach to this problem and justifies it in terms of our successes and failures. Much of our recent work centers on an operational testbed incorporating usable aspects of this technology. This testbed is being used as a prototyping vehicle for our own research and will soon be released to the research community as a framework for development and evolution of Common Lisp systems.

Basili75
Basili, V. R., and A. J. Turner. "Iterative Enhancement: A Practical Technique for Software Development." *IEEE Trans. Software Eng. SE-1*, 4 (Dec. 1975), 390-396.

Abstract: This paper recommends the "iterative enhancement" technique as a practical means of using a top-down, stepwise refinement approach to software development. This technique begins with a simple initial implementation of a properly chosen (skeletal) subproject which is followed by the

gradual enhancement of successive implementations in order to build the full implementation. The development and quantitative analysis of a production compiler for the language SIMPL-T is used to demonstrate that the application of iterative enhancement to software development is practical and efficient, encourages the generation of an easily modifiable product, and facilitates reliability.

Basili81

Basili, V. R., and R. W. Reiter. "A Controlled Experiment Quantitatively Comparing Software Development Approaches." *IEEE Trans. Software Eng. SE-7*, 3 (May 1981), 299-320.

One of the earliest experimental studies to compare the utility and effectiveness of software development techniques available at that time.

Basili86

Basili, V. R., R. Selby, and D. Hutchens. "Experimentation in Software Engineering." *IEEE Trans. Software Eng. SE-12*, 7 (July 1986), 733-743.

Presents a survey of the issues, techniques, and published studies that involve experimental studies of software development practices. An excellent companion paper to [Curtis80] for those who seek to develop a deeper understanding of the challenges and rigors of experimental research in software engineering.

Basili87

Basili, V. R., and H. D. Rombach. "Tailoring the Software Process to Project Goals and Environments." *Proc. 9th. Intern. Conf. Software Engineering.* IEEE Computer Society, 1987, 345-357.

Abstract: This paper presents a methodology for improving the software process by tailoring it to the specific project goals and environment. This improvement process is aimed at the global software process model as well as methods and tools supporting that model. The basic idea is to use defect profiles to help characterize the environment and evaluate the project goals and the effectiveness of methods and tools in a quantitative way. The improvement process is implemented iteratively by setting project improvement goals, characterizing those goals and the environment, in part, via defect profiles in a quantitative way, choosing methods and tools fitting those characteristics, evaluating the actual behavior of the chosen set of methods and tools, and refining the project goals based on the evaluation results. All these activities require analysis of large amounts of data and, therefore, support by an automated tool. Such a tool — TAME (Tailoring A Measurement Environment) — is currently being developed.

Bauer76

Bauer, F. L. "Programming as an Evolutionary Process." *Proc. 2nd. Intern. Conf. Software Engineering.* IEEE Computer Society, Jan. 1976, 223-234.

Describes one of the first approaches to the development of a wide-spectrum language for both specifying and implementing evolving software systems.

Bendifallah87

Bendifallah, S., and W. Scacchi. "Understanding Software Maintenance Work." *IEEE Trans. Software Eng. SE-13*, 3 (March 1987), 311-323.

Abstract: Software maintenance can be successfully accomplished if the computing arrangements of the people doing the maintenance are compatible with their established patterns of work in the setting. To foster and achieve such compatibility requires an understanding of the reasons and the circumstances in which participants carry out maintenance activities. In particular, it requires an understanding of how software users and maintainers act toward the changing circumstances and unexpected events in their work situation that give rise to software system alterations. To contribute to such an understanding, we describe a comparative analysis of the work involved in maintaining and evolving text-processing systems in two academic computer science organizations. This analysis shows that how and why software systems are maintained depends on occupational and workplace contingencies, and vice versa.

Benington56

Benington, H. D. "Production of Large Computer Programs." *Annals of the History of Computing 5*, 4 (1983), 350-361. (Original version appeared in 1956. Also appears in *Proc. 9th. Intern. Conf. Software Engineering*, 299-310).

Abstract: This paper is adapted from a presentation at a symposium on advanced programming methods for digital computers sponsored by the Navy Mathematical Computing Advisory Panel and the Office of Naval Research in June 1956. The author describes the techniques used to produce the programs for the Semi-Automatic Ground Environment (SAGE) system.

Biggerstaff84

Special Issues on Software Reusability. T. Biggerstaff and A. Perlis, eds. *IEEE Trans. Software Eng. SE-10*, 5 (Sept. 1984).

This is a special issue of *IEEE Trans. Software*

Engineering that collects 15 or so papers on different approaches to software reuse that originally were presented at a workshop on the topic sponsored by ITT.

Boehm76

Boehm, B. "Software Engineering." *IEEE Trans. Computers C-25*, 12 (Dec. 1976), 1226-1241.

One of the classic papers in the field of software engineering that focuses attention to the primacy of engineering software systems throughout their development life cycle, rather than just to improved programming practice.

Boehm81a

Boehm, B. W. *Software Engineering Economics.* Englewood Cliffs, N. J.: Prentice-Hall, 1981.

Presents an extensive motivation and treatment of software development and evolution in terms of costs, quality, and productivity issues. Among the results, Boehm indicates that personnel/team capability and other attributes of a software production setting usually have far greater affect on the quality and cost of software products than do new software engineering tools and techniques. It also presents an in-depth discussion of the development and details of the software cost estimation model, COCOMO, that draws upon the extensive studies and analyses that Boehm and associates at TRW have conducted over the years.

Boehm81b

Boehm, B. "An Experiment in Small-Scale Software Engineering." *IEEE Trans. Software Eng. SE-7*, 5 (Sept. 1981), 482-493.

Abstract: This paper reports the results of an experiment in applying large-scale software engineering procedures to small software projects. Two USC student teams developed a small, interactive application software product to the same specification, one using Fortran and one using Pascal. Several hypotheses were tested, and extensive experimental data collected. The major conclusions were as follows.

- *Large-project software engineering procedures can be cost-effectively tailored to small projects.*
- *The choice of programming language is not the dominant factor in small application software product development.*
- *Programming is not the dominant activity in small software product development.*
- *The "deadline effect" holds on small software projects and can be used to help manage software development.*
- *Most of the code in a small application software product is devoted to "housekeeping."*

The paper presents the experimental data supporting these conclusions, and discusses their context and implications.

Boehm84

Boehm, B. W., T. Gray, and T. Seewaldt. "Prototyping vs. Specifying: A Multi-project Experiment." *Proc. 7th. Intern. Conf. Soft. Engr.* , 1984, 473-484.

Abstract: In this experiment, seven software teams developed versions of the same small-size (2000-4000 source instruction) application software product. Four teams used the Specifying approach. Three teams used the Prototyping approach.

The main results of the experiment were:

- *Prototyping yielded products with roughly equivalent performance but with about 40% less code and 45% less effort.*
- *The prototyped products rated somewhat lower on functionality and robustness, but higher on case of use and ease of learning.*
- *Specifying produced more coherent designs and software that were easier to integrate.*

The paper presents the experimental data supporting these and a number of additional conclusions.

Boehm86

Boehm, B. W. "A Spiral Model of Software Development and Enhancement." *ACM Software Engineering Notes 11*, 4 (1986), 22-42.

Presents a new model for modeling the software process that explicitly attempts to address how to manage the risks associated with the development of different kinds of software systems. The presentation of the model is somewhat obscure; however, its focus on addressing risk as a central component in determining how to structure the software development process is unique and worth careful examination.

Budde84

Budde, R., K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven. *Approaches to Prototyping.* New York: Springer-Verlag, 1984.

Presents a collection of papers on software prototyping originally presented at a conference on tht topic in Europe in 1984. After [SEN82], the most extensive survey of approaches to software development and evolution through the use of prototyping tools and techniques.

Cheatham86
Cheatham, T. "Supporting the Software Process." *Proc. 19th. Hawaii Intern. Conf. Systems Sciences.* , 1986, 814-821.

Describes a segment of the radical approach to automating software development introduced in Balzer83b. This segment addresses how to support development and debugging of software components through use of task-level protocols and associated tools.

Curtis87
Curtis, B., H. Krasner, V. Shen, and N. Iscoe. "On Building Software Process Models Under the Lamppost." *Proc. 9th. Intern. Conf. Software Engineering.* IEEE Computer Society, April 1987, 96-103.

Abstract: Most software process models are based on the management tracking and control of a project. The popular alternatives to these models such as rapid prototyping and program transformation are built around specific technologies, many of which are still in their adolescence. Neither of these approaches describe the actual processes that occur during the development of a software system. That is, these models focus on the series of artifacts that exist at the end of phases of the process, rather than on the actual processes that are conducted to create the artifacts. We conducted a field study of large system development projects to gather empirical information about the communication and technical decision-making process that underlie the design of such systems. The findings of this study are reviewed for their implications on modeling the process of designing large software systems. The thesis of the paper is that while there are many foci for process models, the most valuable are those which capture the processes that control the most variance in software productivity and quality.

Curtis80
Curtis, B. "Measurement and Experimentation in Software Engineering." *Proceedings IEEE 68*, 9 (1980), 1144-1157.

Provides a survey of basic concerns that should be addressed in any systematic or experimental study of software development practices.

Distaso80
Distaso, J. "Software Management — A Survey of Practice in 1980." *Proceedings IEEE 68*, 9 (1980), 1103-1119.

Provides a survey of the general issues of software project management based upon experiences in large projects during the 1970's.

Dowson86
Proc. 3rd. Intern. Software Process Workshop, M. Dowson, ed. Los Alamitos, Calif.: IEEE Computer Society, 1986.

Proceedings of the most recent workshop on software process models. Presents short papers on a variety of different approaches to process modeling including object-oriented process programming.

Fairley85
Fairley, R. *Software Engineering Concepts.* New York: McGraw-Hill, 1985.

One of the best textbooks on software engineering currently available.

Gasser86
Gasser, L. "The Integration of Computing and Routine Work." *ACM Trans. Office Info. Sys. 4*, 3 (July 1986), 205-225.

Describes the results of an empirical study of software evolution practices in a large manufacturing organization. Gasser reports that software systems regularly fail to be compatible with the instrumental work activities they are suppose to support, and that a variety of forms of "work-arounds" and other accomodations are performed by users and maintainers to deal with such systems. These accomodations and negotiations therefore play a central role in shaping the evolution of such systems.

Goguen86
Goguen, J. "Reusing and Interconnecting Software Components." *Computer 19*, 2 (Feb. 1986), 16-28.

Abstract: Realizing the considerable economic potential of software reuse requires new programming environment ideas. This article presents a library interconnection language featuring modest use of semantics.

Hekmatpour87
Hekmatpour, S. "Experience with Evolutionary Prototyping in a Large Software Project." *ACM Software Engineering Notes 12*, 1 (1987), 38-41.

Describes three alternative approaches to evolving the development of software systems through prototyping techniques and tools.

Hoffnagel85
Hoffnagel, G. F., and W. Beregi. "Automating the Software Development Process." *IBM Systems J. 24*, 2 (1985), 102-120.

Describes a complementary approach to [Radice85] that introduces automated mechanisms and tech-

niques for supporting large-scale software production processes.

Horowitz85
Horowitz, E., A. Kemper, and B. Narasimhan. "A Survey of Application Generators." *IEEE Software* 2, 1 (Jan. 1985), 40-54.

As the title suggests, this article provides a survey of the basic software mechanisms and components used in many application generators. The presentation is clear and succinct, and represents one of the few published descriptions of the increasingly important software development technology.

Hosier61
Hosier, W. A. "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming." *IRE Trans. Engineering Management EM-8* (June 1961). (Also appears in *Proc. 9th. Intern. Conf. Software Engineering*, 311-327).

Abstract: Real-time digital systems are largely a technical innovation of the past decade, but they appear destined to become more wide spread in the future. They monitor or control a real physical environment, such as an air-traffic situation, as distinguished from simulating that environment on an arbitrary time scale. The complexity and rapid variation of such an environment necessitates use of a fast and versatile central-control device, a role well suited to digital computers. The usual system will include some combination of sensors, communication, control, display, and effectors. Although many parts of such a system pose no novel management problems, their distinguishing feature, the central digital device, frequently presents unusually strict requirements for speed, capacity, reliability and compatibility, together with the need for a carefully designed stored program. These features, particularly the last, have implications that are not always foreseen by management. An attempt is made to point out specific hazards common to most real-time digital systems and to show a few ways of minimizing the risks associated with them.

Humphrey85
Humphrey, W. S. "The IBM Large-Systems Software Development Process: Objectives and Direction." *IBM Systems J. 24*, 2 (1985), 76-78.

The companion paper to [Radice85] and [Hoffnagel85] that introduces and motivates the approaches to modeling and measuring software production at IBM with explicit attention to process organization and management.

Huseth86
Huseth, S., and D. Vines. "Describing the Software Process." *Proc. 3rd. Intern. Software Process Workshop*. IEEE Computer Society, 1986, 33-35.

Briefly describes an approach to the use of object-oriented and frame-oriented knowledge specification languages in developing operational models of software products and production processes.

Ives84
Ives, B., and G. P. Learmonth. "The Information System as a Competitive Weapon." *Comm. ACM 27*, 12 (Dec. 1984), 1193-1201.

Abstract: With the help of information system technology, a company can become competitive in all phases of its customer relationships. The customer resource life cycle model makes it possible for such companies to determine not only when opportunities exist for strategic applications, but also what specific applications should be developed.

Kedzierski84
Kedzierski, B. I. "Knowledge-Based Project Management and Communication Support in a System Development Environment." *Proc. 4th. Jerusalem Conf. Info. Techology.* , 1984, 444-451.

Describes the development of a knowledge-based approach to representing software development task chains and communications between coordinated development agents. A prototype processing support environment is described, as is its suggested use.

Kelly87
Kelly, J. C. "A Comparison of Four Design Methods for Real-Time Systems." *Proc. 9th. Intern. Conf. Software Engineering*. IEEE Computer Society, 1987, 238-252.

Presents an elaborate but practical scheme for examining and comparing different tools/techniques for designing real-time software systems. Such a comparative framework and analysis of various models of software evolution might be derived from this approach. Alternatively, van den Bosch82 presents a different approach to evaluating software development methodologies (or models) through the use of a comparative framework.

Kidder81
Kidder, T. *The Soul of a New Machine*. New York: Atlantic Monthly Press, 1981.

This Pulitzer Prize-winning story describes the development life cycle of a new computing system (hardware and software) by a major computer ven-

dor, together with the dilemmas, opportunites, and social dynamics that shaped its development. Strongly recommended as one of the few descriptions of the real organizational complexities surrounding the development of computing systems.

King84

King, J. L., and K. K. Kraemer. "Evolution and Organizational Information Systems: An Assessment of Nolan's Stage Model." *Comm. ACM 27*, 5 (May 1984), 466-475.

Abstract: Richard Nolan's stage model is the best known and most widely cited model of computing evolution in organizations. The model's development over a decade demonstrates its own evolution from a simple theory, based on the factoring of change states indicated by changes in computing budgets, to an elaborate account of the characteristics of six stages of computing growth. An analysis of the model's logical and empirical structure reveals a number of problems in its formulation that help to account for the fact that its principal tenets have not been independently validated. The model is shown to be an "evolutionistic" theory within the theories of evolution in the social sciences, focusing on assumed directions of growth and an implied end state toward which growth proceeds, and suffering from problems inherent in such theories. Further research based on an "evolutionary" view of computing growth is suggested as a means of improving theories of computing in organizations.

Kling80

Kling, R., and W. Scacchi. "Computing as Social Action: The Social Dynamics of Computing in Complex Organizations." *Advances in Computers 19* (1980), 249-327. Academic Press, New York.

Provides a survey of the organizational dilemmas that can occur during the development and use of system embedded in complex organizational settings. Uses a case study of the life cycle of one system to help articulate six different analytical perspectives for understanding these dilemmas and their interaction.

Kling82

Kling, R., and W. Scacchi. "The Web of Computing: Computer Technology as Social Organization." *Advances in Computers 21* (1982), 1-90. Academic Press, New York.

Asserts the thesis that computing systems and the ways how they are developed and used are inseparably bound to the settings where they are produced and consumed. This work employs case studies to assert the primacy of understanding the interrelationship between software systems, how they are produced, and the settings where they are produced and consumed in order to best understand how they will evolve.

Lehman84a

Lehman, M. M., V. Stenning, and W. Turski. "Another Look at Software Development Methodology." *ACM Software Engineering Notes 9*, 2 (April 1984), 21-37.

Abstract: Software design — from 'topmost' specification down to final implementation — is viewed as a chain of uniform steps, each step being a transformation between two linguistic levels. A canonical form of the step is discussed and it is argued that all rational design activities are expressible as a combination of canonical steps. The role of backtracking in software design is explained and a mechanism for introducing changes, both indigeneous and exogeneous, is proposed, again entirely by a combination of canonical steps. The main tenet of the 'canonical step approach' is that a design step contains a degree of unconstrained, creative invention and a calculable part which is the actual transformation effected.

Lehman84b

Lehman, M. M. "A Further Model of Coherent Programming Processes." *Proc. Software Process Workshop*. IEEE Computer Society, 1984, 27-33.

Abstract: Computer applications and the software that implements them evolve both during initial development and under subsequent usage. Current industrial processes to achieve such evolution are ad hoc. *The individual activities from which they are constituted do not have a common* theoretical base, *are now unified by a single* conceptual framework *and so cannot be combined into a* coherent process. *Yet the latter is essential for the design of* integrated programming support environments *and it is widely recognized that such support is necessary for the creation and evolution (maintenance) of* correct, reliable, cost-effective *programs in a manner that is* responsive *to societal needs.*

Coherent processes, that facilitate evolution of a program over its lifetime, cannot be expected to evolve by juxtaposition of established practices, except over many generations of process instances. The rate at which computerization is penetrating all aspects of societal activity and the reliance this implies on correct definition and operation of software systems, suggest that mankind cannot wait for the 'natural' evolution of responsive and reliable processes. Their design and implementation is a matter of some urgency.

This paper outlines the first steps in the design *of coherent programming processes by* decomposition

and successive refinement of a model of program development and evolution based on a view of programming as a transformation *process.*

Lehman85

Lehman, M. M., and L. Belady. *Program Evolution: Processes of Software Change.* New York: Academic Press, 1985.

Presents a collection of previously published papers that identify and reiterate the "laws" of large program evolution as discovered through empirical investigations at IBM and elsewhere over the preceding 10 year period. Unfortunately, many of the papers state the same data and results, and therefore limit the impact of its contribution.

Lehman86a

Lehman, M. M. "Modes of Evolution." *Proc. 3rd. Intern. Software Process Workshop.* IEEE Computer Society, 1986, 29-32.

Abstract: Computer applications inevitably evolve. The very activity of designing and creating a mechanistic system to automate some human activity leads to a change of perspective and an increase of insight into the problems and approaches to its solution. Installation and operation of the completed system only increases and broadens this effect. The pressures that arise from the changed perceptions, newly recognized needs and opportunities can be controlled but not suppressed. They lead inevitably to demand and, hence, authorization and implementation of system change. And the key to system functional and quality change is primarily through modification of its software. Hence the unending maintenance burden, the continuing process of change and evolution of programs.

Lehman86b

Lehman, M. M. "Approach to a Disciplined Development Process: The ISTAR Integrated Project Support Environment." *ACM Software Engineering Notes 11*, 4 (1986), 49-60.

As part of the papers presented at the second workshop on software process, Lehman describes the development of an approach and an environment that support the production of large software systems by teams of "sub-contractors" working on the project.

Lehman87

Lehman, M. M. "Process Models, Process Programming, Programming Support." *Proc. 9th. Intern. Conf. Software Engineering.* IEEE Computer Society, April 1987, 14-16.

An invited paper that responds to and debates the proposal by Osterweil87 for programming the software process. His critique cites the inherent openess of software development practices and the limits of being able to characterize such practices with algorithmic languages.

Liker86

Liker, J. K., and W. M. Hancock. "Organizational Systems Barriers to Engineering Effectiveness." *IEEE Trans. Engineering Management EM-33*, 2 (1986), 82-91.

Identifies a number of organizational conditions that inhibit or reduce the productivity and effectiveness of engineers working in large organizational settings. Although not specific to software engineering, its analysis and findings are easily applied to this domain.

MIL-STD-2167

Dept. of Defense. DRAFT Military Standard: Defense System Software Development. DOD-STD-2167A.

The current draft of the standard guidelines for developing and documenting software systems by contractors working for the U.S. Department of Defense.

Narayanaswamy87

Narayanaswamy, K., and W. Scacchi. "A Database Foundation to Support Software System Evolution." *J. Sys. and Software 7*, 1 (March 1987), 37-49.

Abstract: Most software engineering researchers focus on supporting the maintenance of large-scale software systems to tackle problems such as managing source code alterations or automating the reconstruction and release of incrementally altered systems from descriptions of their configurations. In this paper, we take the view that information pertaining to the configurations of a system constitute a basic source of knowledge about the system's design and how its component modules fit together. This knowledge is articulated by the use of a special language called NuMIL, which captures the interdependencies between the interfaces of components within a system. We then use a relational database system to store the descriptions. This enables management of the description of large software configurations in an elegant manner, and it facilitates the interactive use of the descriptions in analyzing incremental system alterations and in enhancing the maintainer's understanding of a system.

Neighbors84

Neighbors, J. "The Draco Approach to Constructing Software from Reusable Components." *IEEE Trans. Software Eng. SE-10*, 5 (Sept. 1984), 564-574.

Abstract: This paper discusses an approach called Draco to the construction of software systems from reusable software parts. In particular we are concerned with the reuse of analysis and design information in addition to programming language code. The goal of the work on Draco has been to increase increase the productivity of software specialists in the construction of similar systems. The particular approach we have taken is to organize reusable software components by problem area or domain. Statements of programs in these specialized domains are then optimized by source-to-source program transformations and refined into other domains. The problems of maintaining the representational consistency of the developing program and producing efficient practical programs are discussed. Some examples from a prototype system are also given.

Nolan73

Nolan, R. "Managing the Computer Resource: A Stage Hypothesis." *Comm. ACM 16*, 7 (July 1973), 39-405.

Abstract: Based on the study of expenditures for data processing, a descriptive stage hypothesis is presented. It is suggested that the planning, organizing, and controlling activities associated with managing the computer resource will change in character over a period of time, and will evolve in patterns roughly correlated to four stages of the computer budget: Stage I (computer acquisition), Stage II (intense system development), Stage III (proliferation of controls), and Stave IV (user/service orientation). Each stage is described and related to individual tasks for managing the computer resource.

Osterweil87

Osterweil, L. "Software Processes are Software Too." *Proc. 9th. Intern. Conf. Software Engineering*. IEEE COmputer Society, April 1987, 2-13.

Describes an innovative approach to developing operational programs that characterize how software development activities should occur and how tools can be used to support these activities.

Polak86

Polak, W. "Framework for a Knowledge-Based Programming Environment." *Workshop on Advanced Programming Environments*. Springer-Verlag, 1986.

Describes another segment of the knowledge-based approach to automating software production originally presented in Balzer83b. This segment focuses attention to a specification language and environment supporting the (semi-)automated transformation of software specifications into an implementation language. The techniques and mechanisms employed have since migrated into a commercial product called REFINE.

Potts84

Proc. Software Process Workshop, C. Potts, ed. Los Alamitos, CA: IEEE Computer Society, 1984.

Proceedings of the first workshop on software process modeling which brought attention to the inadequacies of traditional life cycle models as well as suggesting some alternative ways for describing software evolution.

Radice85

Radice, R. A., N. K. Roth, A. L. O'Hara, Jr., and W. A. Ciarfella. "A Programming Process Architecture." *IBM Systems J. 24*, 2 (1985), 79-90.

Describes experiences with the development and practice of an approach to engineering large software systems at IBM. The PPA is a framework for describing the required activities for an operational process for developing software systems. The architecture includes process management tasks, mechanisms for analysis and development of the process, and product quality reviews. It also requires explicit entry criteria, validation, and exit criteria for each task in the software production process.

Redwine85

Redwine, S., and W. Riddle. "Software Technology Maturation." *Proc. 8th. Intern. Conf. Software Engineering*. IEEE Computer Society, 1985, 189-200.

Abstract: We have reviewed the growth and propagation of a variety of software technologies in an attempt to discover natural characteristics of the process as well as principles and techniques useful in transitioning modern software technology into widespread use. What we have looked at is the technology maturation process, the process by which a piece of technology is first conceived, then shaped into something usable, and finally "marketed" to the point that it is found in the repertoire of a majority of professionals.

A major interest is the time required for technology maturation — and our conclusion is that technology maturation generally takes much longer than popularly thought, especially for major technology areas. But our prime interest is in determining what actions, if any can accelerate the maturation of technology, in particular that part of maturation that has to do with transitioning the technology into widespread use. Our observations concerning mat-

uration facilitators and inhibitors are the major subject of this paper.

Royce70

Royce, W. W. "Managing the Development of Large Software Systems." *Proc. 9th. Intern. Conf. Software Engineering.* IEEE Computer Society, 1987, 328-338. Originally published in Proc. WESCON, 1970.

Often cited as the first article to explicate the software life cycle through use of the classic waterfall chart. However, it wasn't until Boehm76 that the central focus of software engineering was explicitly linked to the tools and techniques required to adequately support software life cycle engineering.

Sathi85

Sathi, A., M. S. Fox, and M. Greenberg. "Representation of Activity Knowledge for Project Management." *IEEE Trans. Patt. Anal. and Mach. Intell. PAMI-7*, 5 (1985), 531-552.

Describes a schematic language for representing knowledge about complex production processes. Use of such a knowledge representation language and its associates intelligent system (shell) environment provides an advanced basis for developing knowledge-based models of software products, production processes and their interactions.

Sathi86

Sathi, A., T. Morton, and S. Roth. "Callisto: An Intelligent Project Management System." *AI Magazine 7*, 5 (1986), 34-52.

The follow-on report to Sathi85 which describes the continuing development of a knowledge-based approach to representing and processing complex development projects, with emphasis on emerging issues in knowledge representation.

Scacchi84

Scacchi, W. "Managing Software Engineering Projects: A Social Analysis." *IEEE Trans. Software Eng. SE-10*, 1 (Jan. 1984), 49-59.

Abstract: Managing software engineering projects requires an ability to comprehend and balance the technological, economic, and social bases through which large software systems are developed. It requires people who can formulate strategies for developing systems in the presence of ill-defined requirements, new computing technologies, and recurring dilemmas with existing computing arrangements. This necessarily assumes skill in acquiring adequate computing resources, controlling projects, coordinating development schedules, and employing and directing competent staff. It also requires people who can organize the process for developing and evolving software products with locally available resources. Managing software engineering projects is as much a job of social interaction as it is one of technical direction. This paper examines the social arrangements that a software manager must deal with in developing and using new computing systems, evaluating the appropriateness of software engineering tools or techniques, directing the evolution of a system through its life cycle, organizing and staffing software engineering projects, and assessing the distributed costs and benefits of local software engineering practices. The purpose is to underscore the role of social analysis of software engineering practices as a cornerstone in understanding what it takes to productively manage software projects.

Scacchi86a

Scacchi, W. "Shaping Software Behemoths." *UNIX Review 4*, 10 (Oct. 1986), 46-55.

Describes in an accessible manner how to support the life cycle engineering of large software systems through the use of tools available in the Unix operating system environment.

Scacchi86b

Scacchi, W. and J. Babcock. Understanding Software Technology Transfer. Internal report, Software Technology Program, Microelectronics and Computer Technology Corp., Austin, Texas. (Submitted for publication).

This report surveys empirical studies of software technology transfer and transitions experiences and proposes a framework for understanding how different software technologies should be developed and packages to facilitate their transfer to other settings.

Scacchi86c

Scacchi, W., and C. M. K. Kintala. Understanding Software Productivity. Internal report, Advanced Software Concepts Dept., AT&T Bell Laboratories, Murray Hill, N. J. (Submitted for publication).

This report surveys empirical studies of software productivity measurement. It reports that there are still no adequate quantitative measures or devices that can reliably and accurately measure software productivity. As an alternative, a radical approach to understanding what affects software productivity is proposed that utilizes a knowledge-based approach to modeling and simulating software products, production processes, and production settings as well as their interactions.

Scacchi87
Scacchi, W. "The System Factory Approach to Software Engineering Education." In *Educational Issues in Software Engineering*, R. Fairley and P. Freeman, eds. New York: Springer-Verlag, 1987. (To appear).

This chapter describes an approach to engineering large software systems in a graduate-level software engineering project course. The report describes some of the software engineering tools, techniques, and project management strategies that have been developed over the history of the SF project, as well as some experiences in transferring these technologies to other organizational settings.

SEN82
Special Issue on Rapid Prototyping. ACM Software Engineering Notes 7, 5 (Dec. 1982).

Presents the first collection of full papers on the subject of rapid prototyping of software systems originally appearing at a small workshop on the same topic. Most of the techniques for rapid prototyping that have appeared in subsequent literature and research investigations further explore work appearing in this collection.

Thayer81
Thayer, R., A. Pyster, and R. Wood. "Major Issues in Software Engineering Project Management." *IEEE Trans. Software Eng. SE-7*, 4 (July 1981).

Abstract: Software engineering project management (SEPM) has been the focus of much recent attention because of the enormous penalties incurred during software development and maintenance resulting from poor management. To date there has been no comprehensive study performed to determine the most significant problems of SEPM, their relative importance, or the research directions necessary to solve them. We conducted a major survey of individuals from all areas of the computer field to determine the general consensus on SEPM problems. Twenty hypothesized problems were submitted to several hundred individuals for their opinions. The 294 respondents validated most of these propositions. None of the propositions was rejected by the respondents as unimportant. A number of research directions were indicated by the respondents which, if followed, the respondents believed would lead to solutions for these problems.

Tully84
Tully, C. "Software Development Models." *Proc. Software Process Workshop*. IEEE Computer Society, 1984, 37-44.

This paper discusses information systems, and the system development process, and presents a number of models both of systems and of system development. It also presents one of the few descriptions of the incremental release model of software development practiced by many large system development organizations.

vandenBosch82
van den Bosch, F., J. Ellis, P. Freeman, L. Johnson, C, McClure D. Robinson, W. Scacchi, B. Scheft, A. van Staa, and L. Tripp. "Evaluating the Implementation of Software Development Life Cycle Methodologies." *ACM Software Engineering Notes 7*, 1 (Jan. 1982), 45-61.

Abstract: The cost of developing, maintaining and enhancing software is a major cost factor in many projects. The inability to understand, on a quantitative basis, what factors affect this process severely limits the ability of an organization to make changes that will have a predictable affect on improving quality and productivity of software products.

In the past decade most software organizations have developed a life cycle approach for their organization. The approaches which describe the actions and decisions of the life cycle phases have been formalized as a methodology. Little has been done, however, to define a basis for comparison of these methodologies or even portions of these methodologies. Therefore, there is little data to guide management to direct its organization on what methodologies should be used in the life cycle phases in order to enhance performance in terms of cost, schedule, and technical quality.

This is a proposal for a project to develop a basis for a standard quantitative and qualitative analysis of a software life cycle methodology. The goals of this project are to define a process by which an organization can monitor its life cycle and develop this process to produce better quality software product at a cheaper and more competitive price. In addition, this project will provide a means by which methodologies can be compared across organizations or phases of the software development life cycle. This would be invaluable to large corporations that have many different software development organizations and large agencies who have their own internal software development agencies as well as funding other organizations for large software development projects. This project would provide data that would enable these corporations to specify methodologies to the suborganizations in order to have a positive control on the quality and price of the software product produced.

This project consists of two phases. Both phases will be discussed by this proposal but the actual funding request will only cover the pilot phase. The

pilot phase is a one-year $100,000 project to validate the case study approach to this problem and to redefine the type of questions and methods by which to conduct the interviews and the case study analysis. This pilot project will be followed by a three year project that will begin by studying approximately seven projects and will be the start of establishing the data base to compare methodologies across organizations and phases of a software life cycle.

Wileden86
Intern. Workshop on Software Process and Software Environments. J. Wileden and M. Dowson, eds. *ACM Software Engineering Notes 11*, 4 (1986).

Proceedings of the second workshop on software process modeling. Includes short papers that continue debates over the appropriateness of alternative models of software evolution started in the first software process workshop.

Wirth71
Wirth, N. "Program Development by Stepwise Refinement." *Comm. ACM 14*, 4 (April 1971), 221-227.

Abstract: The creative activity of programming — to be distinguished from coding — is usually taught by examples serving to exhibit certain techniques. It is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures. The process of successive refinement of specifications is illustrated by a short but nontrivial example, from which a number of conclusions are drawn regarding the art and the instruction of programming.

Wiseman85
Wiseman, C. *Strategy and Computers: Information Systems as Competitive Weapons.* New York: Dow Jones Irwin, 1985.

An elaboration of some of the ideas presented in Ives84 that focus attention to viewing the development and evolution of software systems as corporate resources whose capabilities create or inhibit competitive opportunities in the marketplace.

Yacobellis84
Yacobellis, R. H. "Software and Development Process Quality Metrics." *Proc. COMPSAC 84.* IEEE Computer Society, 1984. 262-269.

Describes some early experiments at AT&T Bell Laboratories to monitor and measure software production processes and products. Together with the studies at IBM (cf. Humphrey85), this suggests the growing importance of software process measurement as the basis for studying and improving large-scale industrial software development practices.

Zave84
Zave, P. "The Operational Versus the Conventional Approach to Software Development." *Comm. ACM 27* (Feb. 1984), 104-118.

Abstract: The conventional approach to software development is being challenged by new ideas, many of which can be organized into an alternative decision structure called the "operational" approach. The operational approach is explained and compared to the conventional one.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) SEI-CM-10-1.0	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/AVS HANSCOM AIR FORCE BASE, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) ESD/ AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	10 SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: 63752F PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) Models of Software Evolution: Life Cycle and Process	
12. PERSONAL AUTHOR(S) Walt Scacchi, University of Southern California	
13a. TYPE OF REPORT FINAL 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) October 1987 15. PAGE COUNT 26
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB GR.	software process process model software life cycle life cycle model

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This module presents an introduction to models of software system evolution and their role in structuring software development. It includes a review of traditional software life-cycle models as well as software process models that have been recently proposed. It identifies three kinds of alternative models of software evolution that focus attention to either the products, production processes, or production settings as the major source of influence. It examines how different software engineering tools and techniques can support life-cycle or process approaches. It also identifies techniques for evaluating the practical utility of a given model of software evolution for development projects in different kinds of organizational settings.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION
22a. NAME OF RESPONSIBLE INDIVIDUAL JOHN S. HERMAN, Capt, USAF	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630 22c. OFFICE SYMBOL ESD/AVS (SEI JPO)

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Software Engineering Curriculum Project is developing a wide range of materials to support software engineering education. A *curriculum module* (CM) identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in designing a course. A *support materials* package (SM) contains materials related to a module that may be helpful in teaching a course. An *educational materials* package (EM) contains other materials not necessarily related to a curriculum module. Other publications include software engineering curriculum recommendations and course designs.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

Comments on SEI educational materials and requests for additional information should be addressed to the Software Engineering Curriculum Project, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213. Electronic mail can be sent to education@sei.cmu.edu on the Internet.

Curriculum Modules (* Support Materials available)

CM-1 [superseded by CM-19]
CM-2 Introduction to Software Design
CM-3 The Software Technical Review Process*
CM-4 Software Configuration Management*
CM-5 Information Protection
CM-6 Software Safety
CM-7 Assurance of Software Quality
CM-8 Formal Specification of Software*
CM-9 Unit Testing and Analysis
CM-10 Models of Software Evolution: Life Cycle and Process
CM-11 Software Specifications: A Framework
CM-12 Software Metrics
CM-13 Introduction to Software Verification and Validation
CM-14 Intellectual Property Protection for Software
CM-15 Software Development and Licensing Contracts
CM-16 Software Development Using VDM
CM-17 User Interface Development*
CM-18 [superseded by CM-23]
CM-19 Software Requirements
CM-20 Formal Verification of Programs
CM-21 Software Project Management
CM-22 Software Design Methods for Real-Time Systems*
CM-23 Technical Writing for Software Engineers
CM-24 Concepts of Concurrent Programming
CM-25 Language and System Support for Concurrent Programming*
CM-26 Understanding Program Dependencies

Educational Materials

EM-1 Software Maintenance Exercises for a Software Engineering Project Course
EM-2 APSE Interactive Monitor: An Artifact for Software Engineering Education
EM-3 Reading Computer Programs: Instructor's Guide and Exercises