Part 1

FUNDAMENTALS

COPYRIGHTED MATERIAL

1

INTRODUCTION AND BACKGROUND

Everything is connected to everything.
—Anonymous

Software and information technology professionals, managers, executives, and business analysts have to cope with an increasingly dynamic world. Gone are the days when one's software technology, hardware platforms, organizational environment, and competitive marketplace would stay relatively stable for a few years while developing a system. Thus, the ability to understand and reason about dynamic and complex software development and evolution processes becomes increasingly valuable for decision making.

Particularly valuable are automated aids built upon knowledge of the interacting factors throughout the software life cycle that impact the cost, schedule, and quality. Unfortunately, these effects are rarely accounted for on software projects. Knowledge gleaned from a global perspective that considers these interactions is used in executable simulation models that serve as a common understanding of an organization's processes. Systems thinking, as a way to find and bring to light the structure of the organizational system that influences its dynamic behavior, together with system dynamics as a simulation methodology, provide critical skills to manage complex software development.

System dynamics provides a rich and integrative framework for capturing myriad process phenomena and their relationships. It was developed over 40 years ago by Jay

Software Process Dynamics. By Raymond J. Madachy

Copyright © 2008 the Institute of Electrical and Electronics Engineers, Inc.

Forrester at MIT to improve organizational structures and processes [Forrester 1961]. It was not applied in software engineering until Tarek Abdel-Hamid developed his dissertation model, which is featured in the book *Software Project Dynamics* [Abdel-Hamid, Madnick 1991].

Simulation usage is increasing in many disparate fields due to constantly improving computer capabilities, and because other methods do not work for complex systems. Simulations are computationally intensive, so they are much more cost-effective than in the past. Simulation is general-purpose and can be used when analytic solutions are extremely difficult if not impossible to apply to complex, nonlinear situations. Simulation is even more powerful with improved data collection for the models. Example areas where increased processing power combined with improved models and data include meteorology to better predict hurricane paths, environmental studies, physical cosmology, chemistry to experiment with new molecular structures, or archaeology to understand past and future migrations. These are practical applications but simulation can also be used for experimentation and theory building.

The simulation process in an organization involves designing a system model and carrying out experiments with it. The purpose of these "what if" experiments is to determine how the real or proposed system performs and to predict the effect of changes to the system as time progresses. The modeling results support decision making to improve the system under study, and normally there are unintended side effects of decisions to consider. The improvement cycle continues as organizational processes are continually refined.

Simulation is an efficient communication tool to show how a process works while stimulating creative thinking about how it can be improved. The modeling process itself is beneficial; it is generally acknowledged that much of the reward of modeling is gained in the early stages to gather data, pose questions, brainstorm, understand processes, and so on.

There are many practical benefits of performing simulation in organizations. Besides individual project planning, simulation can help evaluate long-run investment and technology strategies. Companies can use simulation for continuous process improvement, regardless of their current process maturity. It can support organizational learning by making models explicit in a group setting, where all participants can contribute and buy into the model. Such collaboration can go a long way to effect team-building.

Simulation can also be used in individual training, since participants can interact with executing models in real time to see the effects of their decisions. Simulations are used extensively for training in aerospace, military, and other fields. Student awareness is heightened when virtual "games" with simulations are used, particularly when they participate interactively. Visual dynamic graphs or virtual rendering provide faster and more easily remembered learning compared to the traditional lecture format. Exploration is encouraged through the ability to modify and replay the models.

Another significant motivation is that simulation can help reduce the risk of software development. Particularly when used and cross-checked with other complementary analyses that embody different assumptions, process modeling can minimize the

uncertainties of development. Previously unforeseen "gotchas" will be brought to the forefront and mitigated through careful planning.

System dynamics modeling can provide insights by investigating virtually any aspect of the software process at a macro or micro level. It can be used to evaluate and compare different life-cycle processes, defect detection techniques, business cases, interactions between interdisciplinary process activities (e.g. software and nonsoftware tasks), deciding "how much is enough" in terms of rigor or testing, and so on. Organizations can focus on specific aspects of development cost, schedule, product quality, or the myriad trade-offs, depending on their concerns.

The issues of software processes are very wide-ranging, so the scope and boundaries of this book will be defined. The focus is not on technical fundamentals of software programming or specific methodologies, but on the *dynamics* of software processes. The second definition from Webster's dictionary describes the prime focus of this book, particularly the relations between forces:

> *Dynamics*—1. The branch of mechanics dealing with the motions of material bodies under the action of given forces **2. a) the various forces, physical, moral, economic, etc. operating in any field b) the way such forces shift or change in relation to one another c) the study of such forces.**

Essentially, this book is about understanding the dynamics of software processes with the help of simulation modeling. *Software process dynamics* is a more general term than *software project dynamics,* which is limiting in the sense that dynamics occur outside of project boundaries such as continuous product line development, organizational reuse processes contributing to many projects, or other strategic processes. A project is also considered an execution of a process, roughly analogous to how a programming object is an instance or execution of a class.

When simulation is used for personnel training, the term *process flight simulation* is sometimes used to invoke the analogy of pilots honing their skills in simulators to reduce risk, with the implicit lesson that software managers and other personnel should do the same. Use of the system dynamics method may on occasion be referred to as *dynamic process simulation, dynamic simulation,* or *continuous systems simulation.*

Alternative titles for this book could be *The Learning Software Organization* or *Software Process Systems Thinking,* depending on the camp de jour. System dynamics and, particularly, organizational learning gained wider public exposure due to Peter Senge's bestselling book *The Fifth Discipline* [Senge 1990]. Organizational learning in the context of a software process involves translating the common "mental model" of the process into a working simulation model that serves as a springboard for increased learning and improvement. This learning can be brought about by applying system dynamics to software process and project phenomena.

There are other excellent references on system dynamics modeling that one could use to learn from, but why should a busy software engineer studying the software process spend so much time with examples outside of his/her field? This book uses examples solely from the software process domain to minimize modeling skill transfer time. Organizational learning and systems thinking are also well documented else-

where (see the popular books by Peter Senge and collaborators [Senge 1990], [Senge et al. 1994]).

1.1 SYSTEMS, PROCESSES, MODELS, AND SIMULATION

Important terminology for the field is defined in this section. A systems orientation is crucial to understanding the concepts herein, so system will first be defined generally as a subset of reality that is a focus of analysis. Technically, systems contain multiple components that interact with each other and perform some function together that cannot be done by individual components. In simulation literature, a system is typically defined as "a collection of entities, e.g., people or machines, that act and interact together toward the accomplishment of some logical end" [Law, Kelton 1991]. Forrester's system definition is very close: "a grouping of parts that operate together for a common purpose" [Forrester 1968].

Systems exist on many levels; one person's system is another person's subsystem. Since systems are influenced by other systems, no system is isolated from external factors. How to define system boundaries for meaningful analysis is discussed later in this book.

Systems are classified as "open" if the outputs have no influence on the inputs; open systems are not aware of their past performance. A "closed" system is also called a feedback system; it is influenced by its own behavior through a loop that uses past actions to control future action. The distinction between open and closed systems is particularly important in the context of system dynamics.

A system can be characterized by (1) parameters that are independent measures that configure system inputs and structure, and (2) variables that depend on parameters and other variables. Parameters in human systems are directly controllable. The collection of variables necessary to describe a system at any point in time is called the *state* of the system. Examples of state variables for a software process are the number of personnel executing the process; the amount of software designed, coded, and tested; the current number of defects; and so on.

Real-world systems can be classified as static or dynamic depending on whether the state variables change over time. The state of a static system does not change over time, whereas the state of a dynamic system does. Dynamic systems can be further classified as continuous, discrete, or combined, based on how their variables change over time.

Variables change continuously (without breaks or irregularities) over time in a continuous system, whereas they change instantaneously at separated time points in a discrete system. A lake is an example of a continuous system since its depth changes continuously as a function of inflows and outflows, whereas a computer store queue would be considered discrete since the number of customers changes in discrete quantities. A software process arguably has continuous quantities (personnel experience, motivation, etc.) and discrete ones (lines of code, defects, etc.)

Whether a system is seen as continuous, discrete, or combined depends on one's perspective. Furthermore, the choice of a continuous or discrete representation de-

pends on the modeling purpose, and some discrete systems can be assumed to be continuous for easy representation. For example, some would consider a software process to be a system with discrete entities since it can be described by the number of people working, number of units/lines/objects produced, defects originated, and so on, but much difficulty will be avoided if each entity does not need to be traced individually. Hence, the approach in this book and system dynamics in general is to treat the "flow" of the software process as continuous.

A software process is a set of activities, methods, practices, and transformations used by people to develop software. This is a general definition from the commonly accepted Software Engineering Institute's Capability Maturity Model (SEI CMM) [Paulk et al. 1994]. In the context of this book, the software process is the system under study.

A system must be represented in some form in order to analyze it and communicate about it. A *model* in the broadest sense is a representation of reality, ranging from physical mockups to graphical descriptions to abstract symbolic models. Software programs are themselves executable models of human knowledge. A model in the context of this book is a logical, quantitative description of how a process (system) behaves. The models are abstractions of real or conceptual systems used as surrogates for low cost experimentation and study. Models allow us to understand a process by dividing it into parts and looking at how they are related.

Dynamic process models can be discrete, continuous, or a combination of the two. The essential difference is how the simulation time is advanced. Continuous systems modeling methods such as system dynamics always advance time with a constant delta. Since variables may change within any time interval in a continuous system, the delta increment is very small and time-dependent variables are recomputed at the end of each time increment. The variables change continuously with respect to time. Discrete modeling normally is event based. State changes occur in discrete systems at aperiodic times depending on the event nature, at the beginning and end of event activities. The simulation time is advanced from one event to the next in a discrete manner.

All classes of systems may be represented by any of the model types. A discrete model is not always used to represent a discrete system and vice versa. The choice of model depends on the specific objectives of a study. Models of the software processes are either static,[1] in which time plays no role, or dynamic, in which a system evolves over time. The dynamic process models described this book are classified as symbolic, or mathematical ones.

Models may be deterministic, with no probabilistic components, or stochastic, with randomness in the components. Few, if any, software processes are wholly deterministic. Stochastic models produce output that is random and must be handled as such with independent runs. Each output constitutes an estimate of the system characteristics.

[1]A cost model such as COCOMO II [Boehm et al. 2000] is traditionally a static model since the cost factors are treated as constant for the project duration. However, there is a continuum between static and dynamic versions of COCOMO. There are variations that make it possible to introduce time into the calculations.

Simulation is the numerical evaluation of a mathematical model describing a system of interest. Many systems are too complex for closed-form analytical solutions, hence, simulation is used to exercise models with given inputs to see how the system performs. Simulation can be used to explain system behavior, improve existing systems, or to design new systems too complex to be analyzed by spreadsheets or flowcharts.

Finally, *system dynamics* is a simulation methodology for modeling continuous systems. Quantities are expressed as levels, rates, and information links representing feedback loops. Levels represent real-world accumulations and serve as the state variables describing a system at any point in time (e.g., the amount of software developed, number of defects, number of personnel on the team, etc.) Rates are the flows over time that affect the levels. See Table 1.3-1 for a preview description of model elements. System dynamics is described in much more detail in Chapter 2.

A complete and rigorous reference for terms related to modeling and simulation can be found at [DMSO 2006].

1.2 SYSTEMS THINKING

Systems thinking is a way to ferret out system structure and make inferences about the system, and is often described as an overall paradigm that uses system dynamics principles to realize system structure. Systems thinking is well suited to address software process improvement in the midst of complexity. Many organizations and their models gloss over process interactions and feedback effects, but these must be recognized to effect greater improvements.

Systems thinking involves several interrelated concepts:

- A mindset of thinking in circles and considering interdependencies. One realizes that cause and effect can run both ways. Straight-line thinking is replaced by closed-loop causality.
- Seeing the system as a cause rather than effect (internal vs. external orientation). Behavior originates within a system rather than being driven externally, so the system itself bears responsibility. It is the structure of a system that determines its dynamic behavior.
- Thinking dynamically in terms of ongoing relationships rather than statically.
- Having an operational vs. a correlational orientation; looking at how effects happen. Statistical correlation can often be misleading. A high correlation coefficient between two factors does not prove that one variable has an impact on the other.

Systems thinking is, therefore, a conceptual framework with a body of knowledge and tools to identify wide-perspective interactions, feedback, and recurring structures. Instead of focusing on open-loop, event-level explanations and assuming cause and effect are closely related in space and time, it recognizes the world really consists of multiple closed-loop feedbacks, delays, and nonlinear effects.

1.2.1 The Fifth Discipline and Common Models

Senge discusses five disciplines essential for organizational learning in [Senge 1990]: personal mastery, mental models, shared vision, team learning, and systems thinking. Systems thinking is the "fifth" discipline that integrates all the other disciplines and makes organizational learning work. Improvement through organizational learning takes place via shared mental models.

Mental models are used in everyday life for translating personal or organizational goals into issues, questions, and measures. They provide context for interpreting and acting on data, but seldom are stated explicitly. Mental models become more concrete and evolve as they are made progressively explicit. The power of models increases dramatically as they become more explicit and commonly understood by people; hence, process modeling is ideally suited for organizational improvement.

For organizational processes, mental models must be made explicit to frame concerns and share knowledge among other people on a team. Everyone then has the same picture of the process and its issues. Senge and Roberts provide examples of team techniques to elicit and formulate explicit representations of mental models in [Senge et al. 1994]. Collective knowledge is put into the models as the team learns. Elaborated representations in the form of simulation models become the bases for process improvement.

1.2.2 Systems Thinking Compared to System Dynamics

Systems thinking has been an overloaded term in the last 15 years with many definitions. Virtually any comparison with system dynamics is bound to be controversial due to semantic and philosophical issues. Barry Richmond addressed the differences between systems thinking and system dynamics mindsets in detail in [Richmond 1994a]. His major critique about "the historical emphasis of system dynamics" is that the focus has been on product rather than transferring the process (of model building). Only a privileged few developed models and presented them to the world as "the way" as opposed to educating others to model and letting them go at it.[2] His prescription is a systems thinking philosophy of providing skills rather than models per se. Relevant aphorisms include "Give a fish, eat for a day; teach to fish, eat for a lifetime," or "power to the people."

His definition of systems thinking is "the art and science of making reliable inferences about behavior by developing an increasingly deep understanding of underlying structure." It is both a paradigm and a learning method. The paradigm is a vantage point supplemented with thinking skills and the learning method is a process, language, and technology. The paradigm and learning method form a synergistic whole. System dynamics inherently fits in as the way to understand system structure. Thus, system dynamics is a methodology to implement systems thinking and leverage learning efforts.

We prefer not to make any hard distinctions between camps because it is a semantic issue. However, this book is architected in the spirit of systems thinking from the perspective of transferring the process. The goal is to teach people how to model and give

[2]It should be noted encouragingly that the system dynamics pioneer Jay Forrester and others at MIT are involved in teaching how to model with system dynamics in K–12 grades.

them tools to use for themselves, rather than say "here is the model for you to use." This is a major difference between *Software Project Dynamics* and this book. Abdel-Hamid and Madnick present a specific model with no guidance on how to develop a system dynamics model, though very few organizations are content to use the model as-is. Their work is still a seminal contribution and it helped make this book possible.

1.2.3 Weinberg's Systems Thinking

Gerry Weinberg writes about systems thinking applied to software engineering in *Quality Software Management, Volume 1: Systems Thinking* [Weinberg 1992]. It is an insightful book dealing with feedback control and has a close kinship with this book, even though it is almost exclusively qualitative and heuristic. Some academic courses may choose his book as a companion to this one. It provides valuable management insights and important feedback situations to be modeled in more detail.

Weinberg's main ideas focus around management thinking correctly about developing complex software systems—having the right "system model" for the project and its personnel. In a restatement of Brooks's dictum that lack of schedule time has doomed more projects than anything else, Weinberg writes in [Weinberg 1992], "Most software projects have gone awry from management's taking action based on *incorrect system models* than for all other causes combined."

One reason management action contributes to a runaway condition is the tendency to respond too late to deviations, which then forces management to take big actions, which themselves have nonlinear consequences. In order to stay in control of the software process, Weinberg advises to "act early, act small." Managers need to continually plan, observe the results, and then act to bring the actuals closer to planned. This is the prototypical feedback loop for management.

Weinberg was working on his book at the same time that Abdel-Hamid and Madnick were working on theirs, unknown to each other. The day after Weinberg submitted his work to the publisher, he met Abdel-Hamid and realized they were working on parallel and complementary paths for years. Weinberg describes the relationship between the two perspectives as follows. He starts from the low end, so projects get stable enough so that the more precise, high-end modeling exemplified by system dynamics can be even more useful.

Much of Weinberg's book discusses quality, on-the-job pressures, culture, feedback effects, dynamics of size and fault resolution, and more. He proceeds to describe the low-level interactions of software engineering, which are the underlying mechanics for many of the dynamic effects addressed by various process models described in this book. His work is referenced later and provides fodder for some exercises.

1.3 BASIC FEEDBACK SYSTEMS CONCEPTS APPLIED TO THE SOFTWARE PROCESS

Continuous systems modeling has a strong cybernetic thread. The word cybernetic derives from "to control or steer," and cybernetics is the field of science concerned with

processes of communication and control (especially the comparison of these processes in biological and artificial systems) [Weiner 1961]. Cybernetic principles are relevant to many types of systems including moving vehicles (ground, air, water, or space), biological systems, individuals, groups of individuals, and species.

We are all familiar with internal real-time control processes, such as when driving down a road. We constantly monitor our car's position with respect to the lane and make small adjustments as the road curves or obstacles arise. The process of monitoring actual position against desired position and making steering adjustments is similar to tracking and controlling a software project. The same mathematics apply, so system dynamics can be used to model the control aspects of either human driving or project management.

Control systems theory provides a rigorous framework for analyzing complex feedback systems. This section will introduce some basic system notations and concepts, and apply to them to our system of study—the software process. The purpose is to realize a high-level analogy of control principles to our domain, and we will forego mathematical formulae and more sophisticated feedback notations.[3] System dynamics is our chosen method for modeling feedback systems in a continuous-time fashion, as used in the rest of this book.

Figure 1.1 shows the most basic representation of an open system, whereby a black-box system transforms input to output per its internal processing functions. Input and output signals are treated as fluxes over time. It is open because the outputs have no system influence (frequently, it is also called an *open-loop* system despite the absence of any explicit loops). Figure 1.2 shows the closed-loop version with a controller implementing feedback. A decomposition of the controller shows two major elements: a sensor and a control device, shown in Figure 1.3.

The borrowing of these standard depictions from control systems theory can lead to misinterpretation about the "system" of interest for software processes. In both Figures 1.2 and 1.3, the controller is also of major concern; it should not be thought of as being "outside" the system. One reason for problems in software process improvement is that management is often considered outside the system to be improved. Therefore, the boundary for a software process system including management should encompass all the elements shown, including the controller.

Applying these elements to the software process, inputs traditionally represent requirement specifications (or capabilities or change requests), the system is the software development (and evolution) process with the management controller function, and the outputs are the software product artifacts (including defects). The sensor could be any means of measuring the output (e.g., analyzing software metrics), and the control device is the management action used to align actual process results with intended. This notation can represent either a one-time project or a continual software evolution process.

If we consider all types of inputs to the software process, the vector includes resources and process standards as well as requirements. Resources include people and

[3]We are not covering signal polarities, integrators, summers, transfer functions, Laplace transforms, cascaded systems, state space representation, and so on.

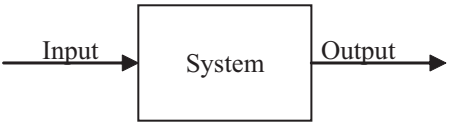

Figure 1.1. Open system.

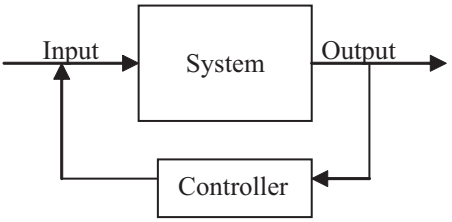

Figure 1.2. Closed system with controller.

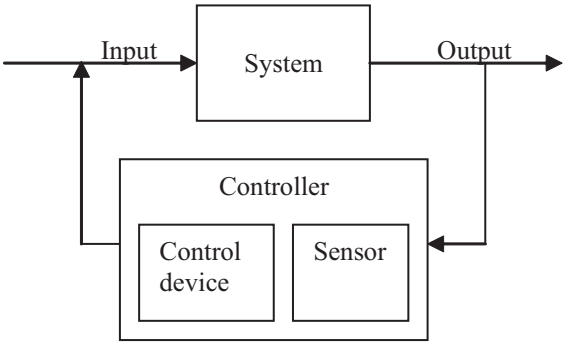

Figure 1.3. Closed system with controller elements.

machines used to develop and evolve the software. Process standards include methods, policies, procedures, and so on. Even the process life-cycle model used for the project can be included (see Section 1.3.2). Requirements include functional requirements, project constraints like budget and schedule, support environment requirements, evolution requirements, and more. Process control actions that management takes based on measurements may affect any of these inputs.

Substituting software process elements into the generic system description produces Figures 1.4, keeping the controller aggregated at the top level representing internal process management.

However, the management controller only represents endogenous process mechanisms local to the development team. These are self-initiated control mechanisms. In reality, there are external, or exogenous feedback forces from the operational environ-

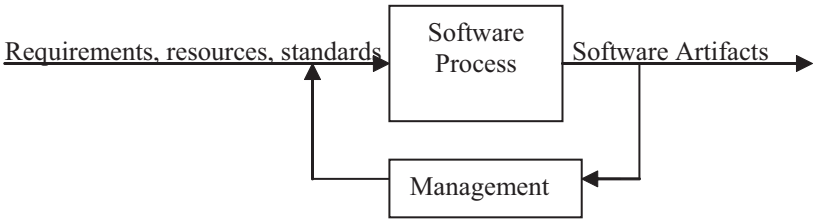

Figure 1.4. Software process control system with management controller.

ment of the software—global feedback. The feedback can be user change requests from the field, other stakeholder change mandates, market forces, or virtually any external source of requirements evolution or volatility. The exogenous feedback is a very important effect to understand and try to control. An enhanced picture showing the two sources of feedback is in Figure 1.5.

The outer, global feedback loop is an entire area of study in itself. Of particular note is the work of Manny Lehman and colleagues on software evolution, which is highlighted in Chapter 5 and referenced in several other places (also see their entries in Appendix B).

These feedback mechanisms shown with control systems notation are implemented in various ways in the system dynamics models shown later. Feedback is represented as information connections to flow rates (representing policies) or other parameters that effect changes in the systems through connected flow rates.

1.3.1 Using Simulation Models for Project Feedback

Projects can proactively use simulation models to adapt to change, thereby taking advantage of feedback to improve through models. This is one way to implement operational control through simulation. A simulation model can be used for metrics-based feedback during project execution since its input parameters represent project objec-

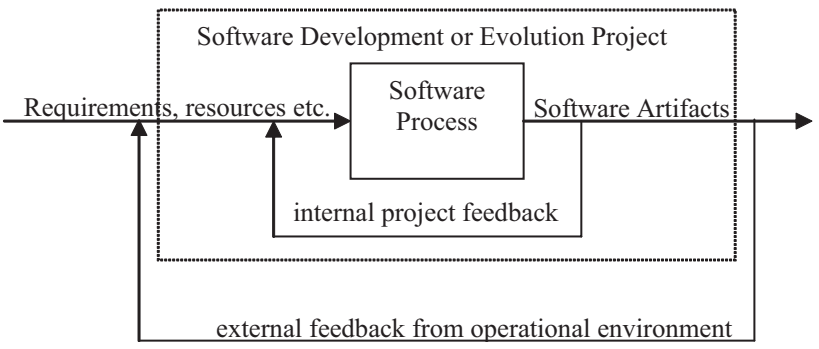

Figure 1.5. Software process control system with internal and external feedback.

tives, priorities, available components, or personnel. It serves as a framework for project rescoping and line management to reassess risks continuously and support replanning.

Figure 1.6 shows a project rescoping framework utilizing metrics feedback and simulation. By inputting parameters representing changed conditions, one can assess whether the currently estimated cost and schedule are satisfactory and if action should be taken. Either rescoping takes places or the project executes to another feedback milestone, where the model is updated with actuals to date and the cycle repeats.

1.3.2 System Dynamics Introductory Example

Table 1.1 is a heads-up preview of system dynamics model elements used throughout this book. The capsule summary may help to interpret the following two examples before more details are provided in Chapter 2 (this table is a shortened version of one in Chapter 2). We are jumping ahead a bit in order to introduce a simple Brooks's Law model. Novices may also want to consult the system dynamics introduction in Chapter 2 to better understand the model elements.

Throughout this text and in other references, levels are synonymous with "stocks" and rates are also called "flows." Thus, a stock and flow representation means an elaborated model consisting of levels and rates.

A simple demonstration example of modeling process feedback is shown in the Figure 1.7 system diagram. In this model, the software production rate depends on the number of personnel, and the number of people working on the project is controlled via a feedback loop. The linear *software production rate* is expressed as

$$\text{software production rate} = \text{individual productivity} \cdot \text{personnel}$$

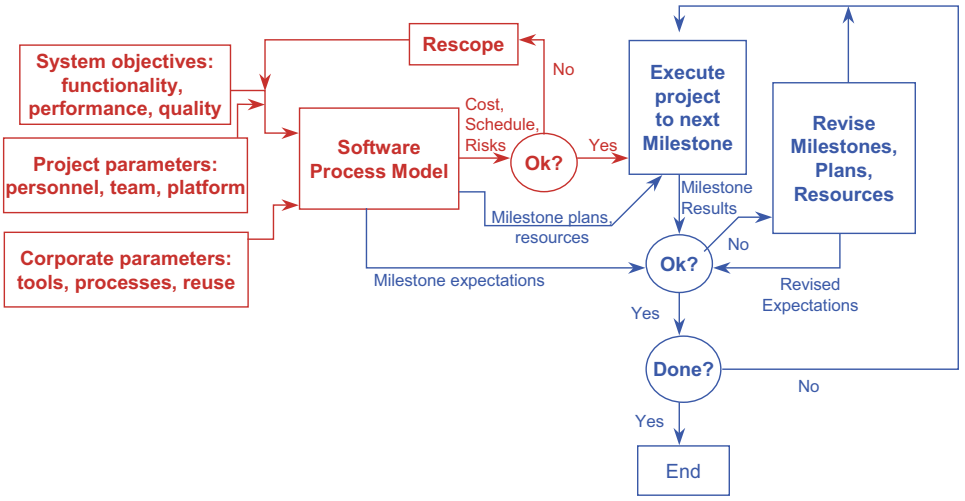

Figure 1.6. Project rescoping framework.

Table 1.1. System dynamics model elements (capsule summary)

Element	Notation	Description
Level		A level is an accumulation over time, also called a stock or state variable. They are functions of past accumulation of flow rates. Examples include software tasks of different forms, defect levels, or personnel levels.
Source/Sink		Sources and sinks indicate that flows come from or go to somewhere external to the process. They represent infinite supplies or repositories that are not specified in the model. Examples include sources of requirements, delivered software, or employee hiring sources and attrition sinks.
Rate		Rates are also called flows. They are the "actions" in a system. They effect the changes in levels and may represent decisions or policy statements. Examples include software productivity rate, defect generation rate, or personnel hiring and deallocation rates.
Auxiliary		Auxiliaries are converters of input to output, and help elaborate the detail of stock and flow structures. Auxiliaries often represent "score-keeping" variables and may include percent of job completion, quantitative goals or planned values, defect density, or constants like average delay times.
Information Link		Information linkages are used to represent information flow (as opposed to material flow). Links can represent closed-path feedback loops between elements. They link process parameters to rates and other variables. Examples include progress and status information for decision making, or knowledge of defect levels to allocate rework resources.

The management decision that utilizes feedback from the actual work accomplished is the following equation for personnel allocation:

personnel allocation rate = if (completed software < planned completion) then 1 else 0

This highly oversimplified feedback decision says to add a person whenever actual progress is less than planned. If actual progress meets or exceeds the planned progress, than no changes are made to the staff (zero people are added). Presumably, this management policy would keep a project on track, assuming a productivity increase when

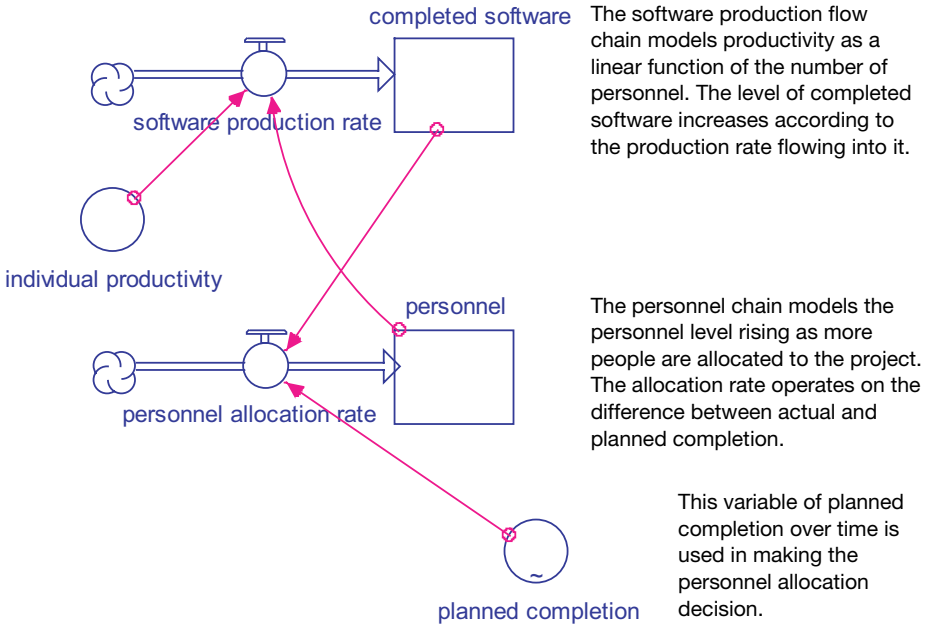

Figure 1.7. System dynamics representation of simple project feedback.

someone is added. It does not consider other ripple effects or nonlinearities of adding extra people, which is addressed in the next example.

1.4 BROOKS'S LAW EXAMPLE

A small motivational example of modeling Brooks's Law is described here. In the early software engineering classic *The Mythical Man-Month* (it was also updated in 1995), Fred Brooks stated, "Adding manpower to a late software project makes it later" [Brooks 1975]. His explanation for the law was the additional linear overhead needed for training new people and the nonlinear communication overhead (a function of the square of the number of people). These effects have been widely accepted and observed by others. The simple model in Figure 1.8 describes the situation, and will be used to test the law. Model equations and commentary are in Figure 1.9.[4] This model is

[4]Mathematically inclined readers may appreciate differential equations to represent the levels. This representation is *not* necessary to understand the model. The following integrals accumulate the inflow and outflow rates over time to calculate a level at any time t using the standard notation:

$$\text{level} = \text{level}_{t=0} + \int_0^t (\text{inflow} - \text{outflow})\, dt$$

$$\text{requirements} = \text{requirements}_{t=0} - \int_0^t (\text{software development rate})\, dt$$

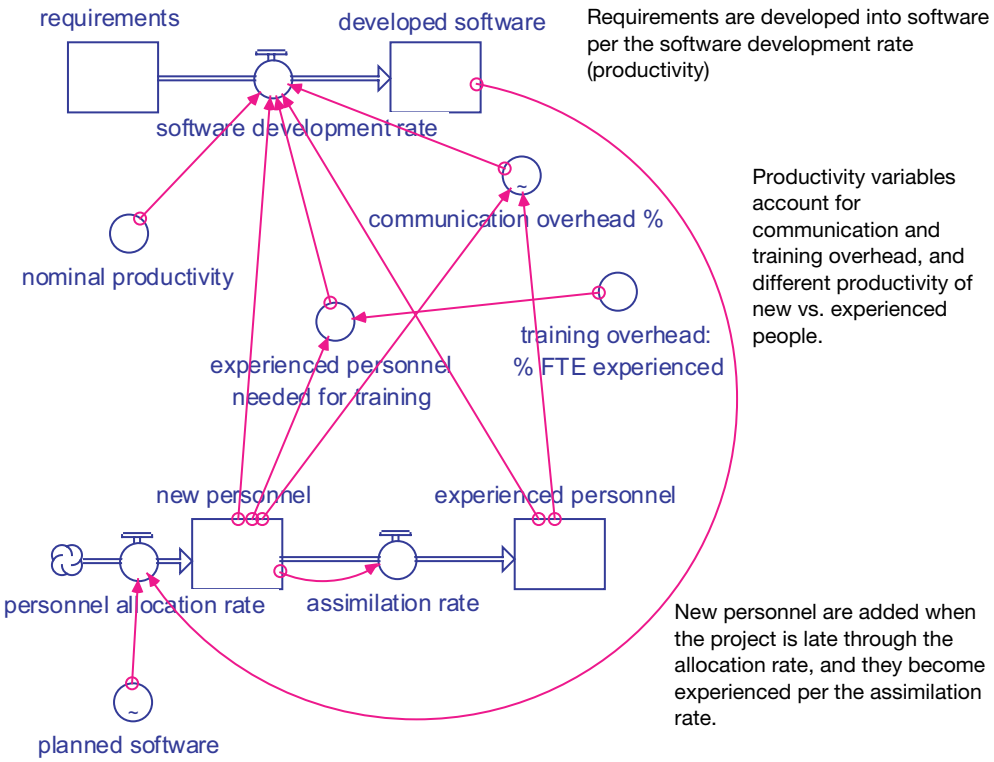

Figure 1.8. Brooks's Law model.

described below assuming no background in systems dynamics. More detail on model notations and equations are discussed in Chapter 2.

The model is conceived around the following basic assumptions:

- New personnel require training by experienced personnel to come up to speed.
- More people on a project entail more communication overhead.
- Experienced personnel are more productive than new personnel, on average.

It is built on two connected flow chains representing software development and personnel. The software development chain assumes a level of *requirements* that needs to

$$\text{developed software} = \text{developed software}_{t=0} + \int_0^t (\text{software development rate})\, dt$$

$$\text{new personnel} = \text{new personnel}_{t=0} - \int_0^t (\text{assimilation rate})\, dt$$

$$\text{experienced personnel} = \text{experienced personnel}_{t=0} + \int_0^t (\text{assimilation rate})\, dt$$

developed_software(t) = developed_software(t - dt) + (software_development_rate) * dt
INIT developed_software = 0

DOCUMENT: This level represents software function points that have been implemented.
INFLOWS:
software_development_rate = nominal_productivity*(1-communication_overhead_%/100.)*(.8*new_personnel+1.2*(experienced_personnel-experienced_personnel_needed_for_training))
DOCUMENT: The development rate represents productivity adjusted for communication overhead, weighting factors for the varying mix of personnel, and the effective number of experienced personnel.

experienced_personnel(t) = experienced_personnel(t - dt) + (assimilation_rate) * dt
INIT experienced_personnel = 20

DOCUMENT: The number of experienced personnel.
INFLOWS:
assimilation_rate = new_personnel/20
DOCUMENT: The average assimilation time for new personnel is 20 days.

new_personnel(t) = new_personnel(t - dt) + (personnel_allocation_rate - assimilation_rate) * dt
INIT new_personnel = 0

DOCUMENT: The number of new project personnel.
INFLOWS:
personnel_allocation_rate = if ((developed_software - planned_software) < - 75 and time <112) then 10 else 0
DOCUMENT: If the gap between developed software and planned software becomes 15% of the plan (75 function points), then add people. The time condition constrains the correction to one time, for demonstration.
OUTFLOWS:
assimilation_rate = new_personnel/20
DOCUMENT: The average assimilation time for new personnel is 20 days.

requirements(t) = requirements(t - dt) + (- software_development_rate) * dt
INIT requirements = 500

DOCUMENT: The project size is 500 function points. This level represents the number left to be implemented.
OUTFLOWS:
software_development_rate = nominal_productivity*(1-communication_overhead_%/100.)*(.8*new_personnel+1.2*(experienced_personnel-experienced_personnel_needed_for_training))
DOCUMENT: The development rate represents productivity adjusted for communication overhead, weighting factors for the varying mix of personnel, and the effective number of experienced personnel.

experienced_personnel_needed_for_training = new_personnel*training_overhead:_%_FTE_experienced/100
DOCUMENT: Training overhead is the effort expended by experienced personnel to bring new people up to speed. It is the number of new personnel * the percent of an experienced person's time dedicated to training.

nominal_productivity = .1
DOCUMENT: The nominal (unadjusted) productivity is .1 function points/person-day.

total_personnel = experienced_personnel+new_personnel

training_overhead:_%_FTE_experienced = 25
DOCUMENT: Percent of full-time equivalent experienced person's time dedicated to training new hires.

communication_overhead_% = GRAPH((experienced_personnel+new_personnel))
(0.00, 0.00), (5.00, 1.50), (10.0, 6.00), (15.0, 13.5), (20.0, 24.0), (25.0, 37.5), (30.0, 54.0)
DOCUMENT: Percent of time spent communicating with other team members as a function of team size. This graph represents the n^2 law in this size region, and was used in the Abdel-Hamid model.

planned_software = GRAPH(time)
(0.00, 0.00), (20.0, 50.0), (40.0, 100), (60.0, 150), (80.0, 200), (100, 250), (120, 300), (140, 350), (160, 400), (180, 450), (200, 500)
DOCUMENT: Plan assumes a constant productivity of 2.5 function points per day.

Figure 1.9. Brooks's Law model equations.

be implemented (shown as the upper left box in Figure 1.8). The *requirements* are transformed into *developed software* at the *software development rate* (rates are shown as the circular pipe-valve symbols). Sometimes, this rate is called "project velocity" (especially by practitioners of agile methods). The level of *developed software* represents progress made on implementing the requirements. Project completion is when *developed software* equals the initial *requirements.* Software size is measured in function points, and the development rate is in function points/day. A function point is a measure of software size that accounts for external inputs and outputs, user interactions, external interfaces, and files used by the system [IFPUG 2004].

The *software development rate* is determined by the levels of personnel in the system: *new project personnel* who come onto the project at the *personnel allocation rate,* and *experienced personnel* who have been assimilated (trained) into the project at the assimilation rate. Further variables in the system are shown as circles in the diagram. Documentation for each model element is shown underneath its equation in Figure 1.9. When constructing the model, the time-based level equations are automatically generated by visually laying out the levels and rates with a simulation tool.

1.4.1 Brooks's Law Model Behavior

The model is a high-level depiction of a level-of-effort project. It allows tracking the *software development rate* over time and assessing the final completion time to develop the requirements under different hiring conditions. When the gap between *developed software* and the *planned software* becomes too great, people are added at the *personnel allocation rate.* All parameters are set to reasonable approximations as described below and in the equation commentary. The overall model represents a nominal case, and would require calibration to specific project environments.

As time progresses, the number of *requirements* decreases since this represents requirements still left to implement. These requirements are processed over time at the *software development rate* and become *developed software,* so *requirements* decline as *developed software* rises. The *software development rate* is constrained by several factors: the *nominal productivity* of a person, the *communication overhead %,* and the effective number of personnel.

The effective number of personnel equals the *new project personnel* plus the *experienced personnel* minus the amount of *experienced personnel needed for training* the new people. The *communication overhead %* is expressed as a nonlinear function of the total number of personnel that need to communicate ($0.06 \cdot n^2$, where n is the number of people). This overhead formulation is described in [Abdel-Hamid, Madnick 1991]. The *experienced personnel needed for training* is the training overhead percentage as a fraction of a full-time equivalent (FTE) experienced personnel. The default of 0.25 indicates that one-quarter of an experienced person's time is needed to train a new person until he/she is fully assimilated.

The bottom structure of the personnel chain models the assimilation of *new project personnel* at an average rate of 20 days. In essence, a new person is trained by one-fourth of an experienced person for an average of 20 days until they become experienced in the project.

The *nominal productivity* is set to 0.1 function points/person-day, with the productivities of new and experienced personnel set to 0.8 · *nominal productivity* and 1.2 · *nominal productivity,* respectively, as a first-order approximation (the fractions roughly mimic the COCOMO II cost model experience factors [Boehm et al. 2000]).

Static conditions are present in the model when no additional people are added. These conditions are necessary to validate a model before adding perturbations (this validation is shown as an example in Chapter 2). The default behavior of the model shows a final completion time of 274 days to develop 500 function points with a constant staff of 20 experienced personnel.

The first experiment will inject an instantaneous pulse of 5 new people when a schedule variance of 15% is reached at about day 110. On the next run, 10 people will be added instead. Figure 1.10 is a sensitivity plot of the corresponding software development rates for the default condition and the two perturbation runs.

Figure 1.10 now shows some interesting effects. With an extra staff of five people (curve #2), the development rate nosedives, then recovers after a few weeks to slightly overtake the default rate and actually finishes sooner at 271 days. However, when 10 people are added the overall project suffers (curve #3). The initial productivity loss takes longer to stabilize, the final productivity is lower with the larger staff, and the schedule time elongates to 296 days. The smooth recovery seen on both are due to the training effect. The extra staff productivity gains in the first case are greater than the communication losses, but going from 20 to 30 people in the second case entails a larger communication overhead compared to the potential productivity gain from having more people.

This model of Brooks's Law demonstrates that the law holds only under certain conditions (Brooks did not mean to imply that the law held in all situations). There is a

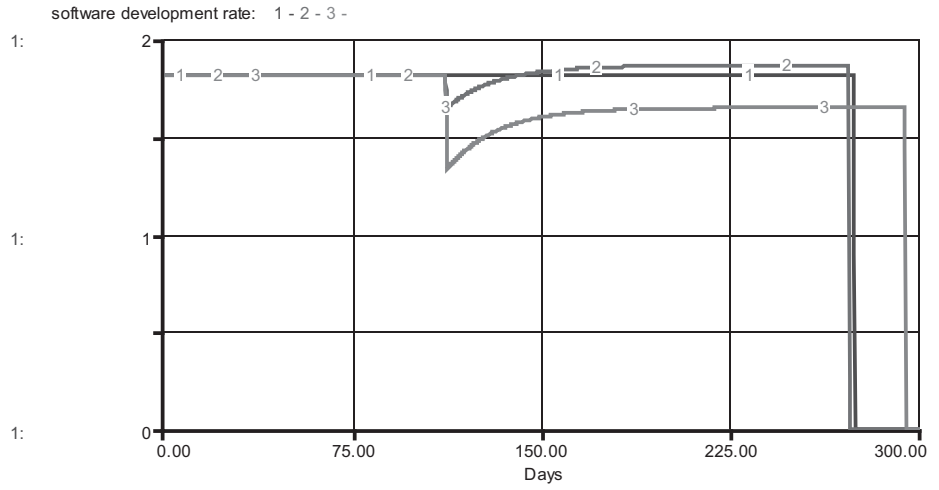

Figure 1.10 Sensitivity of software development rate to varying personnel allocation pulses (1: no extra hiring, 2: add 5 people on 110th day, 3: add 10 people on 110th day).

threshold of new people that can be added until the schedule suffers, showing the trade-off between adding staff and the time in the life cycle. Note that only schedule time was analyzed here, and that effort increased in both cases of adding people. The effort may also be taken into account for a true project decision (though schedule may often be the primary performance index regardless of the cost).

There is a trade-off between the number of added people and the time in the life cycle. The model can be used to experiment with different scenarios to quantify the operating region envelope of Brooks's Law. A specific addition of people may be tolerable if injected early enough, but not later. The project time determines how many can be effectively added.

The model uses simplified assumptions and boundaries, and can be refined in several ways. The parameters for communication overhead, training overhead, assimilation rate, and other formulations of personnel allocation are also important to vary for a thorough sensitivity analysis. A myriad of different combinations can and should still be tested. This model is analyzed in more detail in Chapter 2 to illustrate the principles of sensitivity analysis, and Chapter 6 has an advanced exercise to extend it.

This exercise has demonstrated that a small-scale and simple model can help shed light on process phenomena. Even though the model contains simple formulations, the dynamic time trends would be very time-consuming for a person to manually calculate and all but impossible to mentally figure.

Based on the insight provided, we may now clarify Brooks's Law. *Adding manpower to a late software project makes it later if too much is added too late.* That is when the additional overhead is greater than productivity gains due to the extra staff, as subject to local conditions.

This model is a microcosm of the system dynamics experience. Simplifying assumptions are made that coincide with the simple purpose of this model. Many aspects of the model can be challenged further, such as the following:

- Determining the adequacy of the model before experimenting with it
- Boundary issues: where do requirements come from, and why is there no attrition of people
- Accounting for new requirements coming in after the project has begun
- Varying the starting levels of new and experienced personnel
- Alternate representations to model learning by new hires
- Effects of different hiring scenarios (e.g., step or ramp inputs rather than a single pulse)
- Different relative productivities for experienced and new personnel
- Different communication overhead relationship
- Team partitioning when more people are added
- The policy of determining schedule variance and associated actions

All of these issues will be addressed in subsequent chapters that discuss this Brooks's Law model and others.

1.5 SOFTWARE PROCESS TECHNOLOGY OVERVIEW

Software process technology encompasses three general and overlapping areas: process modeling, process life-cycle models, and process improvement. These areas address highly critical issues in software engineering. A typical example of overlap is modeling of life-cycle processes to achieve process improvement. This book involves all three areas, but the major focus is process modeling with the given technique of system dynamics. Lessons learned from the modeling will give insight into choosing life-cycle models and improving processes. Background on each of these related areas follows in the next subsections.

The term *process model* is overloaded and has meaning in all three contexts. A simulation process model is an executable mathematical model like a system dynamics model, which is the default meaning for process model in this book. However, life-cycle approaches that define the steps performed for software development are sometimes called process models, so the term life-cycle process is used here. Finally, frameworks for process improvement such as Capability Maturity Models, Six Sigma, or ISO 9000 described in Section 1.3.3 are sometimes called process models.

1.5.1 Software Process Modeling

A high-level overview of the software process modeling field is contained in this section, and the remainder of the book expounds on process modeling with system dynamics. Process modeling is representing a process architecture, design, or definition in the abstract. Concrete process modeling has been practiced since the early days of software development [Benington 1956]. Abstract software process modeling started in earnest after Leon Osterweil's landmark paper titled "Software Processes are Software Too" [Osterweil 1987]. The title of Osterweil's statement neatly captures some important principles. As software products are so complex and difficult to grasp, processes to build them should be described as rigorously as possible (with the same precision and rigor used for the software itself). "Process programming" refers to systematic techniques and formalisms used to describe software processes. These techniques to produce process objects are very much the same as those used for software artifacts. The resulting process objects are instantiated and executed on projects. Hence, process descriptions should be considered software and process programming should be an essential focus in software engineering.

Since Osterweil's argument that we should be as good at modeling the software process as modeling applications, software process modeling has gained very high interest in the community among both academic researchers and practitioners as an approach and technique for analyzing complex business and policy questions.

Process modeling can support process definition, process improvement, process management, and automated process-driven environments. Common purposes of simulation models are to provide a basis for experimentation, to predict behavior, answer "what if" questions, or teach about the system being modeled [Kellner et al. 1999]. Such models are usually quantitative. As a means of reasoning about software processes, process models provide a mechanism for recording and understanding processes,

and evaluating, communicating, and promoting process improvements. Thus, process models are a vehicle for improving software engineering practice.

A software process model focuses on a particular aspect of a process as currently implemented (as-is), or as planned for the future (to-be). A model, as an abstraction, represents only some of the many aspects of a software process that potentially could be modeled, in particular those aspects relevant to the process issues and questions at hand.

Models can be used to quantitatively evaluate the software process, implement process reengineering and benchmark process improvement since calibrated models encapsulate organizational metrics. Organizations can experiment with changed processes before committing project resources. A compelling reason for utilizing simulation models is when the costs, risks, or logistics of manipulating the real system of interest are prohibitive (otherwise one would just manipulate the real system).

Process models can also be used for interactive training of software managers; this is sometimes called "process flight simulation." A pilot training analogy is a handy one to understand the role of simulation. Before pilots are allowed to fly jet planes or space vehicles, they are first required to spend thousands of hours in flight simulation, perfecting their skills and confronting different scenarios. This effective form of risk minimization is needed due to the extremely high risks of injury, death, and damaged vehicles. Unfortunately, software managers are too often required to confront project situations without really training for them (despite the very high risks). Why not use simulation to prepare them better?

A model result variable is a primary output of a process simulation. Typical result variables for software process simulation include effort, cycle time, defect levels, staffing requirements over time, return on investment (ROI), throughput, productivity, and queue lengths. Many of these variables may be reported for the entire simulation or any portion thereof. Some result variables are best looked at as continuous functions (e.g., staffing requirements), whereas others only make sense at the end of the process being simulated (e.g., ROI). A more detailed overview of the what, why, and how of software process modeling can be found in [Kellner et al. 1999].

1.5.1.1 Modeling Approaches

To support the aforementioned objectives, process models must go beyond static representation to support process analysis. Unfortunately, there is no standard approach for process modeling. A variety of process definition and simulation methods exist that answer unique subsets of process questions.

Five general approaches to representing process information, as described and reviewed in [Curtis et al. 1992], are programming models (process programming), functional models, plan-based models, Petri-net models (role interaction nets), and system dynamics. The suitability of a modeling approach depends on the goals and objectives for the resulting model. No current software process modeling approach fully satisfies the diverse goals and objectives previously mentioned. Most of these approaches are still used. Over the years, experience has shown that Petri-net models do not scale very well for software processes [Osterweil 2005].

An earlier technique for process modeling is the Articulator approach [Scacchi-Mi 1993]. They created a knowledge-based computing environment that uses artificial intelligence scheduling techniques from production systems for modeling, analyzing, and simulating organizational processes. Classes of organizational resources are modeled using an object-oriented knowledge-representation scheme. Simulation takes place through the symbolic performance of process tasks by assigned agents using tools and other resources.

Other simulation modeling variants that have been applied to software processes include state-based process models, general discrete event simulation, rule-based languages, scheduling approaches, queueing models, and project management (CPM and PERT).

Process modeling languages and representations usually present one or more perspectives related to the process. Some of the most commonly represented perspectives are functional, behavioral, organizational, and informational. They are analogous to different viewpoints on an observable process. Although separate, these representations are interrelated and a major difficulty is tying them together [Curtis et al. 1992], [Kellner 1991]. Ideally, a single approach combines process representation, guidance, simulation, and execution capabilities.

A discrete process modeling approach that combines several language paradigms was developed by Kellner, Raffo, and colleagues at the Software Engineering Institute (SEI) [Kellner 1991], [Raffo 1995]. They used tools that provide state transitions with events and triggers, systems analysis, and design diagrams and data modeling to support representation, analysis, and simulation of processes. This approach enables a quantitative simulation that combines the functional, behavioral, and organizational perspectives.

The two main techniques to gain favor over the last decade are system dynamics and discrete-event simulation. A comparison of the relative strengths and weaknesses between the two are provided in [Kellner et al. 1999], and subsequent chapters in this book discusses their trade-offs for some applications. Recently, agent-based approaches have also been used, and they are expected to increase in popularity [Smith et al. 2006].

Various hybrid approaches have also been used. System dynamics and discrete-event models were combined in a hybrid approach in [Martin-Raffo 2001], [Martin 2002]. Another hybrid two-level modeling approach in [Donzelli, Iazeolla 2001] combines analytical, continuous and discrete-event methods. At the higher abstraction level, the process is modeled by a discrete-event queuing network, which models the component activities (i.e. service stations), their interactions, and the exchanged artifacts. The implementation details of the introduced activities are given at the lower abstraction level, where the analytical and continuous methods are used.

Little-JIL is a graphical method for process programming inspired by Osterweil. It defines processes that coordinate the activities of autonomous agents and their use of resources during the performance of a task [Wise et al. 2000], [Wise et al. 2006]. It uses pre-conditions, post-conditions, exception handlers and a top-down decomposition tree. Programs made with Little-JIL are executable so agents can be guided through a process and ensure that their actions adhere to the process. It is a flexible and

adaptive language initially used to coordinate agents in software engineering [Wise et al. 2000], though recently the process formalism has been applied to other domains besides software demonstrating "ubiquituous process engineering" [Osterweil 2006].

1.5.1.1.1 STATIC VERSUS DYNAMIC MODELING. Traditional analysis approaches are static and cannot capture the dynamic feedback loops that cause real-world complexity. Static modeling techniques assume that time plays no role. Factors are invariant over the duration of a project or other applicable time horizon. However, the dynamic behavior of a system is one of the complexities encountered in real systems. Dynamic models are used when the behavior of the system changes over time and is of particular interest or significance.

For example, static cost and schedule estimation models such as COCOMO II derivatives [Boehm et al. 2000], SEER [Galorath 2005], True-S (formerly PRICE-S) [PRICE 2005], or others provide valuable insight for understanding software project trade-off relationships, but their static form makes it difficult to help reason about complex dynamic project situations. These may include hybrid processes with a mix of life cycles, agile and open-source development methods, or schedule optimization focusing on critical-path tasks. These situations may require understanding of complex feedback processes involving such interacting phenomena as schedule pressure, communication overhead, or numerous others.

1.5.1.1.2 SYSTEM DYNAMICS. The discipline of modeling software processes with system dynamics started with Tarek Abdel-Hamid developing his seminal model of software development for his Ph.D. at MIT. His dissertation was completed in 1984, and he wrote some intervening articles before publishing the book *Software Project Dynamics*[5] with Stuart Madnick in 1991 [Abdel-Hamid, Madnick 1991]. Since then, over 100 publications have directly dealt with software process modeling with system dynamics (see Appendix B for an annotated bibliography of these).

System dynamics is one of the few modeling techniques that involves quantitative representation. Feedback and control system techniques are used to model social and industrial phenomena. Dynamic behavior, as modeled through a set of interacting equations, is difficult to reproduce through modeling techniques that do not provide dynamic feedback loops. The value of system dynamics models is tied to the extent that constructs and parameters represent actual observed project states.

System dynamics addresses process improvement and management particularly well. It also helps facilitate human understanding and communication of the process, but does not explicitly consider automated process guidance or automated execution support. Some system dynamicists stress that *understanding* of system behavior and structure is the primary value of system dynamics as opposed to strict prediction.

As will be detailed in subsequent sections, software process modeling with system dynamics has been used in a large number of companies, governmental agencies

[5]Abdel-Hamid's model and book describe project dynamics for a single fixed process. The title of this book, *Software Process Dynamics,* conveys a more general view of processes and provides assets to analyze a variety of processes, including those outside of project boundaries.

around the world, university research and teaching, and other nonprofit institutions, and has been deployed in marketable software engineering toolsets. It has been used in numerous areas for software processes. The reader is encouraged to read Appendix B to understand the breadth of applications and work in this area.

1.5.1.1.2.1 Why Not Spreadsheets? Many people have tried to use spreadsheets to build dynamic models and failed. Spreadsheets are designed to capture discrete changes rather than a continuous view. It is extremely difficult to use them to model time-based relationships between factors and simulate a system continuously over time. Additionally, they are not made to easily generate different scenarios, which are necessary to analyze risks.

Using spreadsheets for dynamic modeling is a very tedious process that requires each time step to be modeled (e.g., four rows or columns are needed to simulate a week using a standard time step of 0.25). Additionally, this method is actually modeling a difference equation, not a continuous-time model. There is much peril in this approach and it is very easy for errors to creep in with all the related time equations. Some of the basic laws of system dynamics can innocently be violated such as requiring a stock or level in each feedback loop.

But spreadsheets can be important tools for modeling and used in conjunction with simulation tools. They can be used for a variety of purposes, most notably for analyzing outputs of many simulation runs, optimizing parameters, and for generating inputs for the models (such as a stream of random variables). Some system dynamics tools include these capabilities and some do not (a list of available tools is in Chapter 2).

1.5.1.2 Process Model Characterization

Software process modeling can be used in a wide variety of situations per the structure of Table 1.2. The applications shown are representative examples of major models and, thus, are only a sliver of all possible applications. The matrix structure was originated by Marc Kellner as a framework for characterizing modeling problems and relevant modeling work in terms of scope (what is covered) and purpose (why) for the annual Software Process Simulation (ProSim) workshops. System dynamics can be used to address virtually every category, so the table is incompletely populated (it would grow too big if all applications in Appendix B were represented). The matrix can also be used to place the result variables of a simulation study in the proper categorization cell.

The primary purposes of simulation models are summarized below:

- *Strategic management* questions address the long-term impact of current or prospective policies and initiatives. These issues address large portfolios of projects, product-line approaches, long-term practices related to key organizational strengths for competitive advantage, and so on.
- *Planning* involves the prediction of project effort, cost, schedule, quality, and so on. These can be applied to both initial planning and subsequent replanning. These predictions help make decisions to select, customize, and tailor the best process for a specific project context.

Table 1.2. Process modeling characterization matrix and representative major models

	Scope				
Purpose	Portion of Life Cycle	Development Project	Multiple, Concurrent Projects	Long-Term Product Evolution	Long-Term Organization
Strategic Management				Global Feedback [Wernick, Lehman 1999] Progressive and Anti-Regressive Work [Ramil et al. 2005] Value-Based Product [Madachy 2005]	Acquisition [Greer et al. 2005]
Planning	Architecting [Fakharzadeh, Mehta 1999]	Integrated Project Dynamics [Abdel-Hamid, Madnick 1991] Reliability [Rus 1998] Requirements Volatility [Pfahl, Lebsanft 2000] [Ferreira 2002] Hybrid Project Modeling [Martin 2002] Operational Release Plans [Pfahl et al. 2006]	Hybrid Processes [Madachy et al. 2007]		IT Planning [Williford, Chang 1999]
Control and Operational Management		Risk Management [Houston 2000]	Time-Constrained Development [Powell 2001]		

(*continued*)

Table 1.2. *Continued*

	Scope				
Purpose	Portion of Life Cycle	Development Project	Multiple, Concurrent Projects	Long-Term Product Evolution	Long-Term Organization
Process Improvement and Technology Adoption	Sociological Factors in Requirements Development [Christie, Staley 2000]	Inspections [Madachy 1994b] [Tvedt 1996]			Organizational Improvement [Rubin et al. 1994] [Burke 1996] [Stallinger 2000] [Pfahl et al. 2004] [Ruiz et al. 2004]
Training and Learning		Project Management [Collofello 2000] [Pfahl et al. 2001] [Barros et al. 2006]			

- *Control and operational management* involves project tracking and oversight once a project has already started. Project actuals can be monitored and compared against planned values computed by simulation, to help determine when corrective action may be needed. Monte Carlo simulation and other statistical techniques are useful in this context. Simulation is also used to support key operational decisions such as whether to commence major activities.
- *Process improvement and technology adoption* is supported by simulation both *a priori* or *ex post.* Process alternatives can be compared by modeling the impact of potential process changes or new technology adoption before putting them into actual practice. The purpose is process improvement, in contrast to planning. Simulation can also be used ex post to evaluate the results of process changes or selections already implemented.
- *Training and learning* simulation applications are expressly created for instructional settings. Simulations provide a way for personnel to practice or learn project management, similar to pilots practicing on flight simulators.

The original taxonomy also had a purpose category for "understanding" [Kellner et

al. 1999]. That designation is not used here because all models fit in that category. All simulation models promote understanding in some way or another (training and learning are also particularly close semantically). Though some applications may be more dedicated to "theory building" than actual process or project applications (notably the work of Manny Lehman and colleagues on software evolution theory), those efforts are mapped to their potential application areas. More details on the specific categorizations can be found in [Kellner et al. 1999].

The questions and issues being addressed by a study largely determine the choice of result variables, much like choosing a modeling approach. For example, technology adoption questions often suggest an economic measure such as ROI; operational management issues often focus on the traditional project management concerns of effort, cycle time, and defect levels. It is common to produce multiple result variables from a single simulation model.

1.5.2 Process Life-Cycle Models

Broadly speaking, a process life-cycle model determines the steps taken and order of the stages involved in software development or evolution, and establishes the transition criteria between stages. Note that a life-cycle model can address portfolios of projects such as in product-line development, not just single projects. An often perplexing problem is choosing an appropriate process for individual situations. Should a waterfall or iterative process be used? Is an agile process a good fit for the project constraints and goals? What about incremental development, spiral risk-driven processes, the personal software process, specific peer review methods, and so on? Should the software be developed new or should one buy COTS or use open-source components? Dynamic simulation models can enable strategic planners and managers to assess the cost, schedule, or quality consequences of alternative life-cycle process strategies.

The most commonly used life-cycle models are described in the following subsections. Knowing about the alternative processes is important background to better understand process behavior and trade-offs in the field. From a pragmatic modeling perspective, it also helps to understand the different life cycles in order to have a good modeling process for given situations. Chapter 2 describes the modeling process as being iterative in general, but aspects from other life cycles may also be important to consider. The spiral process in particular is a good fit for modeling projects (also see Chapter 2).

The proliferation of process models provides flexibility for organizations to deal with the wide variety of software project situations, cultures, and environments, but it weakens their defenses against some common sources of project failure, and leaves them with no common anchor points around which to plan and control. In the iterative development section, milestones are identified that can serve as common anchor points in many different process models.

1.5.2.1 Waterfall Process

The conventional waterfall process is a single-pass, sequential approach that progresses through requirements analysis, design, coding, testing, and integration. Theoretical-

ly, each phase must be completed before the next one starts. Variations of the waterfall model normally contain the essential activities of specification, design, code, test, and, frequently, operations/maintenance. Figure 1.11 shows the commonly accepted waterfall process model.

The first publication of a software process life-cycle model was by Winston Royce [Royce 1970]. In this original version of the waterfall, he offered two primary enhancements to existing practices at the time: (1) feedback loops between stages and (2) a "build-it-twice" step parallel with requirements analysis, akin to prototyping (see next section). The irony is that the life-cycle model was largely practiced and interpreted without the feedback and "build-it-twice" concept. Since organizations historically implement the waterfall sequentially, the standard waterfall definition does not include Royce's improvements.[6]

The waterfall was further popularized in a version that contained verification, validation, or test activity at the end of each phase to ensure that the objectives of the phase were satisfied [Boehm 1976]. The phases are sequential, none can be skipped, and baselines are produced throughout the phases. No changes to baselines are made unless all interested parties agree. The economic rationale for the model is that to achieve a successful product, all of the subgoals within each phase must be met and that a different ordering of the phases will produce a less successful product.

However, the waterfall model implies a sequential, document-driven approach to product development, which is often impractical due to shifting requirements. Some other critiques of the waterfall model include: a lack of user involvement, ineffectiveness in the requirements phase, inflexibility to accommodate prototypes, unrealistic separation of specification from design, gold plating, inflexible point solutions, inability to accommodate reuse, and various maintenance-related problems.

Some of the difficulties of the waterfall life-cycle model have been addressed by extending it for incremental development, parallel development, evolutionary change, automatic code generation, and other refinements. Many of these respond to the inadequacies of the waterfall by delivering executable objects early to the user and increasing the role of automation. The following sections describe selected enhancements and alternatives to the waterfall model. Note that process models can be synthesized into hybrid approaches when appropriate.

1.5.2.2 Incremental Development

The incremental approach is an offshoot of the waterfall model that develops a system in separate increments versus a single-cycle approach. All requirements are determined up front and subsets of requirements are allocated to individual increments or releases. The increments are developed in sequential cycles, with each incremental release adding functionality. The incremental approach reduces overall effort and provides an initial system earlier to the customer. The effort phase distributions are different and the overall schedule may lengthen somewhat. Typically, there is more effort in requirements analysis and design, and less for coding and integration.

[6]His son, Walker Royce, has modernized life-cycle concepts in [Royce 1998], but also defends the original waterfall model and describes how practitioners implemented it in wrong ways that led to failures.

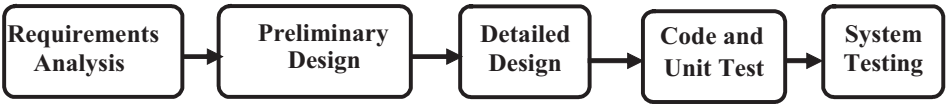

Figure 1.11. Conventional waterfall process.

The incremental process is sometimes called staged delivery, since software is provided in successively refined stages. Requirements analysis and preliminary architectural design is done once up front for all increments. Once the design is complete, the system can be implemented and delivered in stages. Some advantages of staged delivery include: (1) it allows useful functionality to be put into the hands of the customer much earlier than if the entire product were delivered at the end, (2) with careful planning it may be possible to deliver the most important functionality earliest, and (3) it provides tangible signs of progress early in the project. Almost all large projects use some form of incremental development.

1.5.2.3 Evolutionary Development

Evolutionary life-cycle models were developed to address deficiencies in the waterfall model, often related to the unrealistic expectation of having fully elaborated specifications. The stages in an evolutionary model are expanding increments of an operational product, with evolution being determined by operational experience. Evolutionary approaches develop software in increments, except that requirements are only defined for the next release. Such a model helps deliver an initial operational capability and provides a real-world operational basis for evolution; however, the lack of long-range planning often leads to trouble with the operational system not being flexible enough to handle unplanned paths of evolution.

With evolutionary prototyping, development is begun on the most visible aspects of the system. That part of the system is demonstrated, and then development continues on the prototype based on the feedback received. This process is useful when requirements are changing rapidly, when the customer is reluctant to commit to a set of requirements, or when neither the developer nor the customer understands the application. Developers find the model useful when they are unsure of the optimal architecture or algorithm to use. The model has the advantage of producing steady visible signs of progress, which can be especially useful when there is a strong demand for development speed.

The main disadvantage of the model is that it is not known at the beginning of the project how long it will take to create an acceptable product. The developer will not know how many iterations will be necessary. Another disadvantage is that the approach lends itself to becoming a code-and-fix development.

In evolutionary delivery, a version of the product is developed; it is shown to the customer, and the product is refined based on customer feedback. This model draws from the control obtained with staged delivery and the flexibility afforded by evolutionary prototyping. With evolutionary delivery, the developer's initial emphasis is on

the visible aspects of the system and its core, which consists of lower-level system functions that are unlikely to be changed by customer feedback. Plugging holes in the system's foundation occurs later on.

1.5.2.4 Iterative Development

An iterative process uses multiple development cycles to produce software in small, usable pieces that build upon each other. It is often thought that iterative development is a modern concept; however, it was first documented in 1975 [Basili, Turner 1975]. Development proceeds as a series of short evolutionary iterations instead of a longer single-pass, sequential waterfall process for fixed requirements. The difference with incremental development is that iterative approaches do not lay out all the requirements in advance for each cycle. Each iteration may resemble a very short waterfall, and prototyping is an essential aspect for the vast majority of applications containing a user interface. The first iteration usually implements a core capability with an emphasis on addressing high-risk areas, stabilizing an architecture, and refining the requirements. The capabilities and architecture are built upon in successive iterations.

Iterative development helps to ensure that applications meet user needs. It uses a demonstration-based approach in order to refine the user interface and address risky areas [Royce 1998]. Programming starts early and feedback from real end users is integral to the evolution of a successful application. For example, working slices of the software architecture will be assessed early for areas of high risk (e.g. security, throughput, response time).

When defining the user interface for a new application, the project team will gather requirements and translate them quickly into an initial prototype. Next, the team will step through the prototype to validate the results and make changes. By rapidly iterating and refining "live" code, the final application is ensured to meet the needs and expectations of the users. This approach is how we successively developed user-intensive e-commerce systems at C-Bridge Internet Solutions [Madachy 2001].

Some implementation examples of an iterative process include the Rational Unified Process (RUP) [Kruchten 1998], the USC Model-Based Architecting and Software Engineering (MBASE) framework (see Section 1.5.2.5.2), and most forms of agile methods. A typical profile of activities in iterative development is shown in Figure 1.12, adapted from RUP documentation. The milestone acronyms at the top stand for Life-Cycle Objectives (LCO), Life-Cycle Architecture (LCA), and Initial Operating Capability (IOC). The milestones and their criteria are described in [Boehm 1996] and [Royce 1998].

This life-cycle process reduces the entropy and large breakage associated with the waterfall. An overall functionality is delivered in multiple iterative pieces with the highest payback features first, and the overall scope is delivered in a shorter time frame compared to using the waterfall. Incremental application releases allow early return on investment and ease the burden of change management in the user's environment. The project loops through multiple development cycles until the overall scope of the application is achieved.

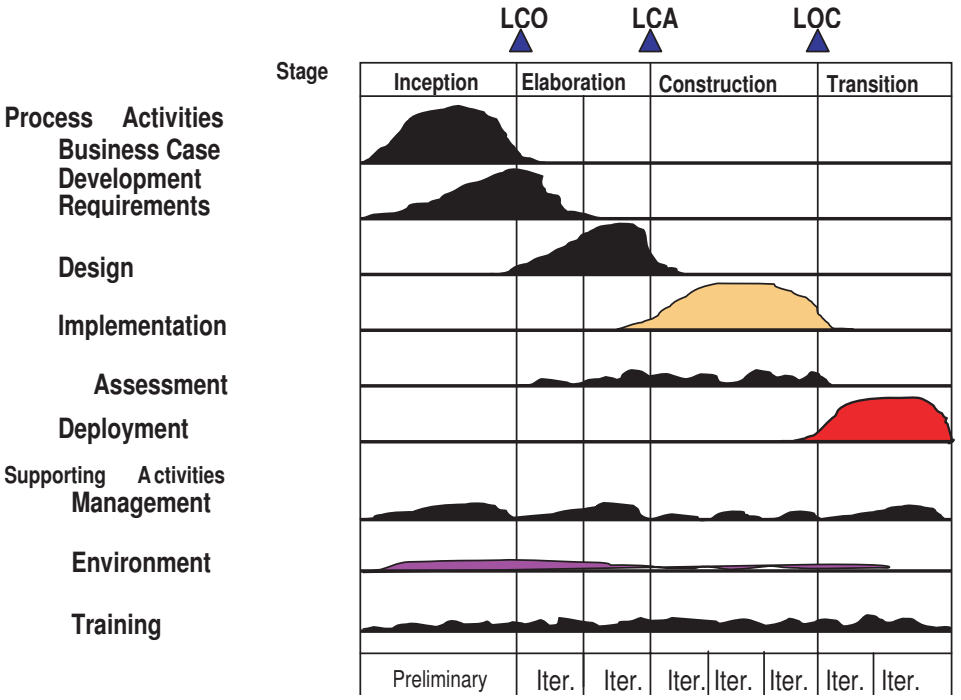

Figure 1.12. Activity profiles for typical iterative processes.

Business trends continue to place more emphasis on being first to market. Iterative software development processes are well suited for schedule-driven projects, and provide users with working systems earlier than waterfall processes.

1.5.2.5 Spiral Model

The spiral model is a risk-driven variation of evolutionary development that can function as a superset process model. It can accommodate other models as special cases and provides guidance to determine which combination of models best fits a given situation [Boehm 1988]. It is called risk-driven because it identifies areas of uncertainties that are sources of project risk and structures activities based on the risks. The development proceeds in repeating cycles of determining objectives, evaluating alternatives, prototyping and developing, and then planning the next cycle. Each cycle involves a progression that addresses the same sequence of steps for each portion of the product and for each level of elaboration. Development builds on top of the results of previous spirals.

A major distinction of the spiral model is having risk-based transition criteria between stages in contrast to the single-shot waterfall model in which documents are

the criteria for advancing to subsequent development stages. The spiral model prescribes an evolutionary process, but explicitly incorporates long-range architectural and usage considerations in contrast to the basic evolutionary model. The original version of the spiral model is shown in Figure 1.13. The process begins at the center, and repeatedly progresses clockwise. Elapsed calendar time increases radially with additional cycles.

A typical cycle of the spiral model starts with identification of objectives (for the portion of the product being elaborated), alternative means of implementing the portion of the product (e.g., design A, design B, buy) and constraints imposed on the application of the alternatives such as cost or schedule budgets. Next is an evaluation of the alternatives relative to the objectives and constraints. Areas of uncertainty are identified that are significant sources of risk, and the risks are evaluated by prototyping, simulation, benchmarking, or other means.

What comes next is determined by the remaining relative risks. Each level of software specification and development is followed by a validation step. Each cycle is completed by the preparation of plans for the next cycle and a review involving the primary people or organizations concerned with the product. The review's major objective is to ensure that all concerned parties are mutually committed to the approach for the next phase.

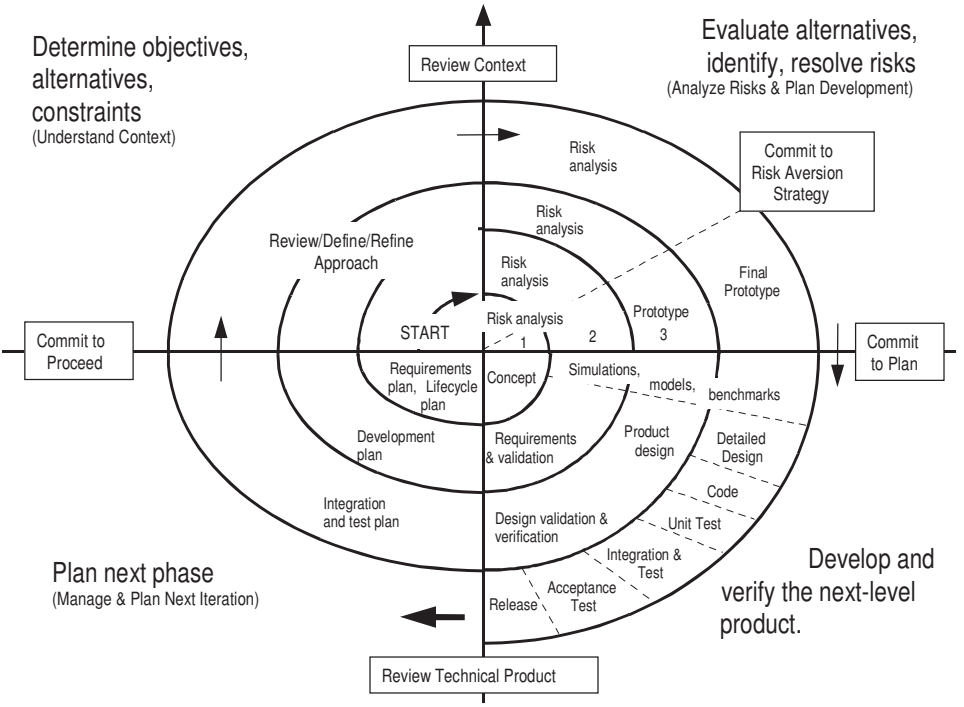

Figure 1.13: Original spiral model.

1.5.2.5.1 WinWin Spiral Model. The original spiral model of software development begins each cycle by performing the next level of elaboration of the prospective system's objectives, constraints, and alternatives. A primary difficulty in applying the spiral model has been the lack of explicit process guidance in determining these objectives, constraints, and alternatives. The more recently developed WinWin spiral model [Boehm et al. 1998] uses the Theory W (win–win) approach [Boehm, Ross 1989] to converge on a system's next-level objectives, constraints, and alternatives. This Theory W approach involves identifying the system's stakeholders and their win conditions, and using negotiation processes to determine a mutually satisfactory set of objectives, constraints, and alternatives for the stakeholders.

Figure 1.14 illustrates the WinWin spiral model. The original spiral had four sectors, beginning with "Establish next-level objectives, constraints, alternatives." The two additional sectors in each spiral cycle, "Identify Next-Level Stakeholders" and "Identify Stakeholders' Win Conditions," and the "Reconcile Win Conditions" portion of the third sector, provide the collaborative foundation for the model. They also fill a missing portion of the original spiral model, the means to answer, "Where do the next-level objectives and constraints come from, and how do you know they're the right ones?" The refined spiral model also explicitly addresses the need for concurrent analysis, risk resolution, definition, and elaboration of both the software product and the software process.

An elaborated cycle of the WinWin spiral includes the following:

- Identify the system or subsystem's key stakeholders.
- Identify the stakeholders' win conditions for the system or subsystem.

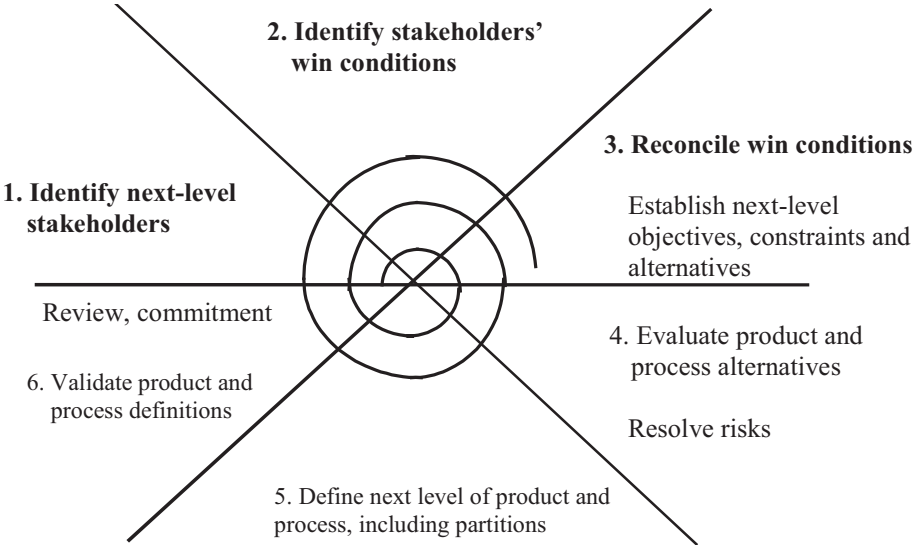

Figure 1.14. WinWin Spiral (additions to spiral model shown in bold).

- Negotiate win–win reconciliations of the stakeholders' win conditions.
- Elaborate the system or subsystem's product and process objectives, constraints, and alternatives.
- Evaluate the alternatives with respect to the objectives and constraints. Identify and resolve major sources of product and process risk.
- Elaborate the definition of the product and process.
- Plan the next cycle and update the life-cycle plan, including partition of the system into subsystems to be addressed in parallel cycles. This can include a plan to terminate the project if it is too risky or infeasible. Secure the management's commitment to proceed as planned.

1.5.2.5.2 MBASE Framework. At the USC Center for Systems and Software Engineering we have been developing the Model-Based[7] Architecting and Software Engineering (MBASE) framework. MBASE addresses debilitating dynamics of software development that often lead to project downfalls by embodying a risk-driven iterative approach, heavily involving project stakeholders per the WinWin spiral model, and evolving a system architecture compatible with changes and future evolution. Its milestones can be used in many different process life-cycle models. For an iterative process, the milestones are used to achieve stakeholder concurrence on the current project state.

MBASE has many features in common with the RUP, and is fully compatible with the iterative life cycle described in Walker Royce's book *Software Project Management—A Unified Approach* [Royce 1998]. Chapter 5 of this book has an application model for the MBASE architecting process.

1.5.2.6 Prototyping

Prototyping is a process whereby the developer creates a quick initial working model of the software to be built to support requirements definition. The objective is to clarify the characteristics and operation of a system using visual representations and sample working sessions. Partial requirements gathering is done first with stakeholders. Structured workshop techniques are good for this. A quick design is produced, focusing on software aspects visible to the user, and the resulting prototype is evaluated to refine the requirements. Prototyping produces a full-scale model and functional form of a new system, but only the part of interest. This approach may be used when there is only a general set of objectives, or other uncertainties exist about the desired form of an interface or algorithmic efficiencies.

Prototyping is highly valuable to help elicit requirements by incorporating user feedback. A phenomenon that often occurs when users have difficulty elucidating what they want is "I'll Know It When I See It" (IKIWISI). Not until they see some exploratory or sample screens can they point and say "that looks good, that's what I want

[7]The term "Model" in the name is a general descriptor referring to a large variety of models used in software (e.g., success models, development standards, and many others), and is not at all limited to simulation models or life-cycle models.

now." Prototyping is used heavily in modern user-interactive systems, and is an essential component of Rapid Application Development (RAD) and iterative development.

There are drawbacks to prototyping. Often, canned scenarios are used to demonstrate a particular functionality, and compromises are inevitably made to accommodate the quick development. A customer may think that the prototype is more robust than it is in reality, and may not understand why it cannot be used in the actual system. Prototyping has similar aspects to simulation. Neither produces the software product per se, but they are complementary ways to address project risk. Prototyping helps with product risks and process simulation helps with process risk.

1.5.2.7 Agile Methods

Agile methods are a set of iterative processes that have gained much favor in recent years. They have been advocated as an appropriate "lightweight" programming paradigm for high-speed and volatile software development [Cockburn 2001]. Some examples of agile methods include Extreme Programming (XP), SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, and Pragmatic Programming.

As opposed to more plan-driven approaches espoused by the Capability Maturity Models (see Section 1.5.3.1), agile development is less document-oriented and more focused on programming. The so-called "agile manifesto" [Agile 2003] identifies an agile method as one that adopts the four value propositions and twelve principles in the agile manifesto. All of the concepts in the value propositions below are important and valid but, in general, agilists value the ones on the left over the right:

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

The twelve principles behind the agile manifesto are:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

The title of a relevant article characterizes agile methods as "Agile Software Development: It's about Feedback and Change" [Williams, Cockburn 2003]. In this sense, agile methods go hand in hand with using system dynamics to model feedback and adapt to change. Agile methods are a manifestation of fine-tuned process feedback control. Agile methods have drawbacks and trade-offs like other methods, and the interested reader can consult [Boehm, Turner 2004] for a balanced treatment.

1.5.2.8 Using Commercial-off-the-Shelf (COTS) Software or Open-Source Software

Software development with third-party components has become more prevalent as applications mature. The components help provide rapid functionality and may decrease overall costs. These components may be purchased as COTS or obtained from open-source repositories. COTS is sold by vendors and is generally opaque, whereas open source is from the public domain with full access to its source code. Both process strategies pose new risks and different life-cycle activities.

COTS-based or COTS-intensive systems utilize commercial packages in software products to varying degrees. Using COTS has unique challenges. Developers do not have access to the source code and still have to integrate it into their systems. Development with COTS requires assessment of the COTS offerings, COTS tailoring, and, quite possibly, glue-code development to interface with the rest of the software.

Frequently, the capability of the COTS or open-source software determines systems requirements, instead of the converse. Another challenge is timing the product "refresh" as new releases are put out. A system with multiple COTS will have different release schedules from different vendors. See [Abts 2003] for a comprehensive (static) COTS cost model. Chapter 5 describes some dynamic COTS application models. Chapter 7 discusses current and future directions of developing COTS-based systems and open-source software development, and makes important distinctions between their life-cycle impacts.

1.5.2.9 Reuse and Reengineering

Reuse-based development uses previously built software assets. Such assets may include requirements, architectures, code, test plans, QA reviews, documents, and so on. Like COTS, reuse has the potential to reduce software effort and schedule. Though ex-

tra costs are incurred to make software reusable, experience has shown that the investment is typically recouped by about the third round of reuse. Reuse is frequently incorporated in other life-cycle models, and a number of process models dedicated to reuse and reengineering have been published. Some are risk-driven and based on the spiral model, while others are enhanced waterfall process models. A process for reusing software assets can also improve reliability, quality, and user programmability.

Reengineering differs from reuse in that entire product architectures are used as starting points for new systems, as opposed to bottom-up reuse of individual components. A software process model for reuse and reengineering must provide for a variety of activities and support the reuse of assets across the entire life cycle. It must support tasks specific to reuse and reengineering, such as domain analysis and domain engineering.

1.5.2.10 Fourth-Generation Languages and the Transform Model

Fourth-generation languages (4 GLs) and transformational approaches specify software at a high level and employ automatic code generation. They automatically convert a formal software specification into a program [Balzer et al. 1983]. Such a life-cycle model minimizes the problems of code that has been modified many times and is difficult to maintain. The basic steps involved in the transform process model are: (1) formally specify the product, (2) automatically transform the specification into code (this assumes the capability to do so), (3) iterate if necessary to improve the performance, (4) exercise the product, and (5) iterate again by adjusting the specification based on the operational experience. This model and its variants are often called operational specification or fourth-generation language (4GL) techniques. Chapter 5 has an example application model for 4GLs.

1.5.2.11 Iterative Versus Waterfall Life-Cycle Comparison

A modern iterative life-cycle process, when practiced right, is advantageous to a conventional waterfall life cycle for a majority of applications. It is particularly recommended for user-interactive systems, where the IKIWISI principle applies. The difference between a modern iterative process versus a conventional waterfall process in terms of project progress is idealized in Figure 1.15, which is adapted from [Royce 1998]. Progress is measured in terms of the percent of software demonstrable in target form.

Figure 1.15 also calls out the significant contrasting dynamics experienced between the two life-cycle processes. Notable are the larger breakages and elongated schedule of the waterfall life cycle. No software is produced during requirements analysis, so nothing is demonstrable until design or coding. During design, there is apparent progress on paper and during reviews, but the system architecture is not really assessed in an implementation environment. Not until integration starts can the system be tested to verify the overall structure and the interfaces. Typically there is a lot of breakage once integration starts, because that is the first time that unpredictable issues crop up and interfaces are finally fleshed out. The late-surfacing problems stretch out the integration and testing phase and resolving them consumes many resources. Last-minute

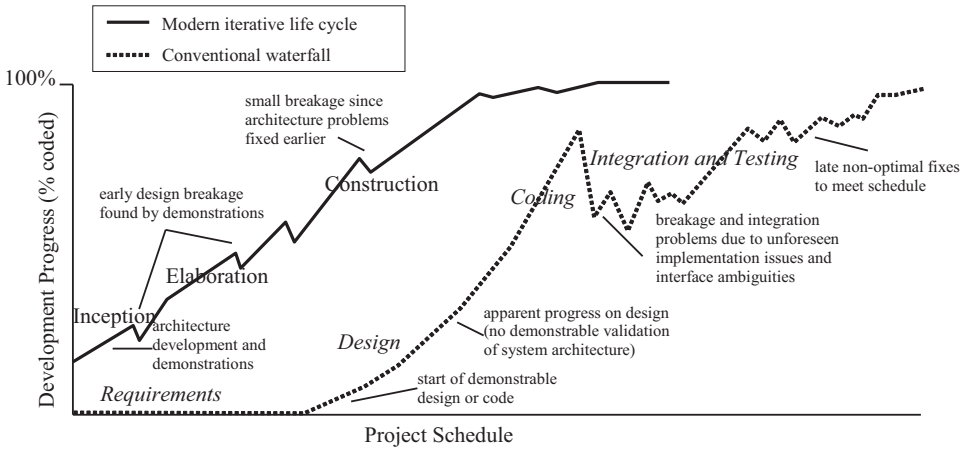

Figure 1.15. Progress dynamics for iterative life cycle versus waterfall life cycle (adapted from Walker Royce).

fixes are necessary to meet an already slipped schedule, so they are usually quick and dirty, with no time to do things optimally.

The iterative life cycle focuses on developing an architecture first, with frequent demonstrations that validate the system. The demonstrations may be in the form of prototypes or working slices that validate the proposed architecture. The working demonstrations help to mitigate risk by identifying problems earlier when there are still options to change the architecture. Some breakage is inevitable, but the overall life-cycle impact is small when the problems are addressed at this stage. The system architecture is continually being integrated via the executable demonstration of important scenarios. There is minimal risk exposure during construction since there are many fewer unknowns, so the final integration and transition proceed smoothly.

1.5.3 Process Improvement

In order to address software development problems, the development task must be treated as a process that can be controlled, measured, and improved [Humphrey 1989]. This requires an organization to understand the current status of their processes, identify desired process change, make a process improvement plan, and execute the plan. All of these process improvement activities can be supported with simulation.

Industrial initiatives to reengineer and improve software processes through metrics is helping to motivate modeling efforts. Initially, the Software Engineering Institute Capability Maturity Model (SEI CMM) process maturity framework was an important driver of these efforts [Paulk et al. 1994], but it has been superseded by the CMM-Integrated (CMM-I) [SEI 2003], which brings systems engineering into the process improvement fold among other improvements. Frameworks for process enhancements are described next.

1.5.3.1 Capability Maturity Models

The Software Capability Maturity Model (CMM or SW-CMM) was developed by the Software Engineering Institute (SEI) to improve software engineering practices [Paulk et al. 1994]. The more recent CMMI ("I" stands for integration) is a broader framework based on experience with the SW-CMM that includes systems engineering practices [SEI 2003]. They are both models of how organizations can mature their processes and continually improve to better meet their cost, schedule, and quality goals. They are frameworks of management, engineering, and support best practices organized into process areas. Capabilities in the process areas can be improved through generic practices outlined in the frameworks.

The SEI was initially tasked by the U.S. government to develop the CMM, and it has become a de-facto standard adopted by commercial and defense companies worldwide. CMMI has been a collaborative effort among members of industry, government, and the SEI. CMMI is also consistent and compatible with ISO/IEC 15504, an emerging standard on software process assessment.

The CMM framework identifies five levels of organizational maturity, where each level lays a foundation for making process improvements at the next level. The framework utilizes a set of key practices and common features. The CMM staged representation offers a road map for improvement one predetermined step at a time. Process areas are grouped at maturity levels that provide organizations with a sound approach for process improvement. The staged representation prescribes the order of implementation for each process area according to maturity levels.

CMMI also allows for either the traditional staged improvement or a continuous improvement path within a specific process area. The continuous representation offers a more flexible approach to process improvement. It is designed for organizations that would like to choose a particular process area or set of process areas. As a process area is implemented, the specific practices and generic practices are grouped into capability levels.

A primary intent of CMMI is to provide an integrated approach to improving an organization's systems engineering and software functions. CMMI covers the same material as the SW-CMM, although some topics like risk management and measurement and analysis are given more extensive treatment that reflects the engineering community's learning over the past 10 years. In CMMI, systems engineering is defined as "The interdisciplinary approach governing the total technical and managerial effort required to transform a set of customer needs, expectations, and constraints into a product solution and support that solution throughout the product's life."

Related maturity models include the Software Acquisition CMM (SA-CMM), the People Capability Maturity Model, the FAA-iCMM, and others. Though the CMMI is intended to supercede the SW-CMM, the SW-CMM will still be in use for a while. The subsequent sections are predicated on the software CMM but apply just as well to CMMI.

At level one in the maturity models, an organization is described as ad-hoc, whereby processes are not in place and a chaotic development environment exists. The second level is called the repeatable level. At this level, project planning and tracking is established and processes are repeatable. The third level is termed the defined level be-

cause processes are formally defined. Level 4 is called the managed level, whereby the defined process is instrumented and metrics are used to set targets. Statistical quality control can then be applied to process elements. At level 5, the optimized level, detailed measurements and analysis techniques are used to guide programs of continuous process improvement, technology innovation, and defect prevention. The CMMI key process areas and their general categories are shown in Table 1.3.

To make the process predictable, repeatable, and manageable, an organization must first assess the maturity of the processes in place. By knowing the maturity level, organizations can identify problem areas to concentrate on and effect process improvement in key practice areas. The SEI has developed assessment procedures based on the maturity models and standardized the assessments using ISO/IEC 15504.

Software process modeling is essential to support improvements at levels 4 and 5 through process definition and analysis. Process models can represent the variation among processes, encapsulate potential solutions, and can be used as a baseline for benchmarking process improvement metrics.

Table 1.3. CMMI key process areas

	Process area category			
Level	Engineering	Project Management	Process Management	Support
5—Optimizing			• Organizational Innovation & Deployment	• Causal Analysis and Resolution
4—Managed		• Quantitative Project Management	• Organizational Process Performance	
3—Defined	• Requirements Development • Technical Solution • Product Integration • Verification • Validation	• Integrated Project Management • Risk Management • Integrated Supplier Management	• Organizational Process Focus • Organizational Process Definition • Organizational Training	• Decision Analysis and Resolution • Organizational Environment for Integration
2—Repeatable	• Requirements Management	• Project Planning • Project Monitoring and Control • Supplier Agreement Management		• Configuration Management • Process & Product Quality Assurance • Measurement and Analysis
1—Ad hoc	No process areas defined for this level			

Table 1.4 identifies some general uses of simulation at the different levels and the type of data available to drive the models. Though dynamic process simulation is not absolutely necessary at the lower levels, some process modeling capability is needed to advance in levels 4 and 5. As maturity increases, so does the predictive power of the organizational models. Highly mature organizations can test their improved processes via simulation rather than through costly trial and error in the field.

The distinction between open and closed feedback systems described in Section 1.3 typifies the difference between an ad-hoc CMM Level 1 organization and higher maturity organizations in which past process performance data is used to control the current process. The essence of CMM levels 4 and 5 can be summarized as using process feedback. This implies that a mature organization would desire some model of that feedback. See Chapter 5 for an example of process maturity modeling.

Table 1.4. Simulation uses at the CMM levels

CMM Level	Simulation Uses	Calibration Data Available
5—Optimizing	• Testing new processes offline before project implementation and performing process trade-off studies • Modeling impacts of new technology adoption • Comparing defect prevention techniques • Interdepartmental collaboration and widespread usage of simulation • Organizational modeling • Product line analysis • Model change management	• Continuous stream of detailed data tailored for organizational goals • Results of previous simulations and decisions against actuals
4—Managed	• Time-based process predictions to determine planned values • Determination of process control limits through variance analysis • More widespread usage of simulation among software and managerial staff	• Finer detail and more consistent data • Specific process and product measures targeted to environment • Previous model estimates
3—Defined	• Defect modeling • Peer review optimization • Training • Increased validity and confidence in process models allow supplanting some expert judgment	• Phase and activity cost and schedule data • Size, requirements, and defect data • Peer review data
2—Repeatable	• Cost and schedule modeling • Earned value setting • Evaluating requirements	• Project-level cost and schedule • Expert judgment
1—Ad hoc	• Improve awareness of process dynamic behavior • Role playing games	

1.5.3.2 Other Process Improvement and Assessment Frameworks

ISO 9000 and ISO/IEC 15504 are other process assessment frameworks. These consist of standards and guidelines as well as supporting documents that define terminology and required elements, such as auditing and assessment mechanisms. Neither CMM, ISO 9000, nor ISO/IEC 15504 are product standards. They focus on processes under the assumption that if the production and management system is right, the product or service that it produces will also be correct. Six Sigma is a defined process for improving products and processes.

1.5.3.2.1 ISO 9000. The International Organization for Standards (ISO) published ISO 9000 in 1987 for quality management certification [ISO 2005]. ISO 9000 is primarily concerned with the establishment of a quality management system. The definition of quality is captured in twenty "must address" clauses and enforced by a formal, internationally recognized, third-party audit and registration system. ISO 9000 requires the organization to "walk the talk": say what it does, do what it says, and be able to demonstrate it. It requires the organization to write specific procedures defining how each activity in their development process is conducted.

Processes and procedures must address control, adaptability, verification/validation, and process improvement. Organizations must always be able to show objective evidence of what has been done, how it was done, and the current status of the project and product, and it must be able to demonstrate the effectiveness of its quality system.

An accreditation schema for software, motivated by ISO 9000's apparent lack of applicability to the software industry, led to the TickIT method, which is limited to the United Kingdom and Sweden. A more powerful assessment-based certification standard for software was developed under the informal name of SPICE. It was eventually formalized as the ISO 15504 standard described next.

1.5.3.2.2 ISO/IEC 15504. The aim of the ISO/IEC 15504 standard is to perform process assessment, process improvement, and capability determinations [SEI 2005]. ISO/IEC 15504 combines the CMM and ISO 9000 approaches into a single mechanism for developing a quality system for software. It embodies the reference framework of the ISO 9000 approach with the capability assessment and process maturity features of CMM. In addition, it establishes a migration path for existing assessment models and methods that wish to come into the 15504 domain. It intentionally does not specify a particular assessment methodology. Its architecture is designed to permit and encourage the development of methods and models that serve specific domains or markets. See [Stallinger 2000] for a system dynamics modeling application for ISO/IEC 15504.

1.5.3.2.3 Six Sigma. Six Sigma is a methodology initially used to manage process variations that cause defects, defined as unacceptable deviation from the mean or target, and to systematically work toward managing variation to eliminate those defects. Six Sigma has now grown beyond defect control. The objective of Six Sigma is to deliver high performance, reliability, and value to the end customer. See

[Motorola 2006] for details, but there are also many other online sources, books, and guides.

The methodology was pioneered at Motorola in 1986 and was originally defined as a metric for measuring defects and improving quality, and a methodology to reduce defect levels below 3.4 defects per million opportunities. Hence, "Six Sigma" refers to the ends of a probability distribution that correspond to these odds. The principles and techniques of using probability distributions (see Appendix A) are important to understand when implementing Six Sigma.

When deploying Six Sigma for improvement in an organization, there are clearly defined roles for executive and project sponsors, black belts, green belts, yellow belts, and so on (the belts correspond to levels of Six Sigma training and skill). On a project, a common road map for developing new processes or products is Define, Measure, Analyze, Design/Build, Verify (DMADV). The road map for improving products or processes that already exist is known as Define, Measure, Analyze, Improve, Control (DMAIC).

Six Sigma is a natural application for software process modeling to quantify and forecast effects of variation on new processes and designs. Key factors that drive uncertainty can be illuminated with simulation to better explain the effects of variation without expensive trials.

1.6 CHALLENGES FOR THE SOFTWARE INDUSTRY

We have just scratched the surface of the potential of dynamic modeling to help improve the software process. Accounting for all the dynamic factors on a software project far outstrips the capability of the human mind. For example, Capers Jones has identified about 250 different factors that can affect schedule, cost, quality, and user satisfaction [Jones 2000]. Clearly, one cannot mentally calculate all the influences from so many factors; thus, we resort to computer simulation technology.

Correspondingly, there are formidable challenges for the industry and pitfalls to avoid. The ability to handle more phenomena and have high confidence in models will help system dynamics make powerful impacts. Simulation models need their own verification, validation, and accreditation just like other software products. There are confidence limits associated with the calibration of models, and it is harder to verify and validate highly detailed models. Methods for model certification and ways to converge on acceptable models must be considered. As will be described later, some possibilities include the rigorous use of the Goal–Question–Metric (GQM) process, getting user buy-in and consensus on key model drivers, model assessment techniques per Chapter 2 and Appendix A, incremental validation, and cross-validation with other types of models. Many of these topics are explored in more detail in Chapter 2 and case studies in subsequent chapters. Chapter 7 discusses some automated model analysis techniques.

The field is continuously getting more complex for decision makers. Software development is becoming even more dynamic with agile methods, increasing use of COTS, complex systems of systems, open-source development via the Internet, dis-

tributed 24/7 global development, new computing devices, model-driven development, Rapid Application Development (RAD) processes, and increasingly shorter increments of development. New management techniques are needed to keep up with the changing environment, as well as available professionals trained in new techniques. Most of these topics are treated later in the book and Chapter 7 discusses their current and future directions. Some present-day decision scenarios are described next to illustrate the wide array of decision contexts and help set the stage.

Planners and managers need to assess the consequences of alternative strategies such as reuse and COTS, fourth-generation languages (4GLs), application generators, architectural refactoring, rapid prototyping, incremental or iterative development, or other process options before actual implementation. And what are the effects of interactions between requirements elicitation, software implementation, testing, process improvement initiatives, hiring practices, and training within any of these strategies?

In projects with multiple COTS packages, one problem is synchronizing releases from several products with different update schedules. How frequently the components should be "refreshed" and the overall software rebuilt is a complex decision. Will different COTS packages integrate well with each other, or do we need to develop special interface glue code? COTS glue code development is modeled in Chapter 5. Will the vendors be able to support future versions and upgrades?

There are countless individual strategies to improve the software process (e.g., formal inspections, object-oriented development, cleanroom engineering, automated testing, iterative development, etc.). But what are the conditional effects of combined strategies? Can they be separated out for analysis? For example, there are overlaps between methods for finding defects. What are the types of defects found by different means and what are the reduced efficiencies of single methods when used in conjunction with others? See Chapter 5 for examples on defects.

Trying to find the right balance between process flexibility and discipline is a large challenge. How much of a project should be performed with an agile approach and how much should be plan-driven? See Chapter 4 for an example application. It would also be valuable to understand the limits of scaling up agile methods to decide when best to use them.

Companies that are looking for ways to accelerate their processes can assess various RAD techniques through simulation. Methods are also needed, for example, to evaluate how many people are required to maintain a software product at a specified level of change, whether a modification to the inspection process will result in higher defect yields, or if a process can be implemented with fewer reviews and still maintain quality.

In today's interconnected world, large, complex systems of systems have their own unique dynamics. Trying to manage a globally distributed project involving many teams and interfaces is a unique challenge. What are the best ways to accomplish this process management in order to get successful systems out the door?

It would be useful to model the complex adaptation behavior of projects and the ripple effects of decisions made. In order to salvage runaway projects, teams sometimes incorporate more reused software and COTS software. Often, managers have to perform midstream corrections to "design to cost" or "design to schedule." This could

mean a reduction in scope by dropping features, or finding ingenious shortcuts to lessen the effective size, such as incorporating middleware from another source (a ripple effect may be the extra time incurred to do so). These shortcuts can often be attributed to the ideas of "top people." Such individuals are invaluable to organizations, but they are in short supply and may cost a lot. The effects of having such people need to be better understood to capitalize on their talents.

People are the most valuable resources in software; as such, a lot of attention should be paid to their training and retention. Having a national or global supply of skilled workers requires cooperation between many organizations and educational institutions. Modeling can help assess different ways to achieve skill development, and to keep people motivated to address these industry challenges (see Chapters 4 and 7 on people resource issues).

These are just a few examples of complex decision scenarios that the software industry is currently grappling with. Things will get even more complex and dynamic as software technology and business conditions keep changing. The rest of the book will address how to achieve better understandings of the process interrelationships and feedback, in order to cope with a fast-changing software world.

1.7 MAJOR REFERENCES

[Abdel-Hamid, Madnick 1991] Abdel-Hamid, T., and Madnick, S., *Software Project Dynamics,* Englewood Cliffs, NJ: Prentice-Hall, 1991.

[Boehm et al. 1998] Boehm, B., Egyed, A., Kwan, J., Port, D., Shah, A., and Madachy, R., Using the WinWin Spiral Model: A Case Study, *IEEE Computer,* July 1998.

[Brooks 1975] Brooks, F., *The Mythical Man-Month,* Reading, MA: Addison-Wesley 1975 (also reprinted and updated in 1995).

[Forrester 1961] Forrester, J. W., *Industrial Dynamics,* Cambridge, MA: MIT Press, 1961.

[Kellner et al. 1999] Kellner, M., Madachy, R., and Raffo, D., *Software Process Simulation Modeling: Why? What? How? Journal of Systems and Software,* Spring 1999.

[Kruchten 1998] Kruchten, P., *The Rational Unified Process,* Reading, MA: Addison-Wesley, 1998.

[Osterweil 1987] Osterweil, L., Software Processes are Software Too, *Proceedings ICSE 9,* IEEE Catalog No. 87CH2432-3, pp. 2–13, March 1987.

[Paulk et al. 1994] Paulk, M., Weber, C., Curtis, B., and Chrissis, M., *The Capability Maturity Model: Guidelines for Improving the Software Process,* Reading, MA: Addison-Wesley, 1994.

[Royce 1998] Royce, W., *Software Project Management—A Unified Approach,* Reading, MA: Addison-Wesley, 1998.

[Senge 1990] Senge, P., *The Fifth Discipline,* New York: Doubleday, 1990.

[Weinberg 1992] Weinberg, G., *Quality Software Management, Volume 1, Systems Thinking,* New York: Dorset House Publishing, 1992.

CHAPTER 1 SUMMARY

Simulation is a practical and flexible technique to support many areas of software processes to improve decision making. It can be used to model different aspects of processes in order to assess them, so that the dynamic (and unintended) consequences of decisions can be understood. Through simulation, one can test proposed processes before implementation. It is effective when the real system is not practical to use or is too expensive. Often, simulation is the only way to assess complex systems. Systems are becoming more complex and computer capabilities continue to improve, so it is expected that the use of simulation will continue to increase over the years in software engineering and other fields.

Systems come in a variety of types (discrete/continuous/mixed, static/dynamic), and there are different techniques for modeling them. This book will focus on the system dynamics technique for modeling dynamic continuous systems.

Systems thinking is an allied discipline that leverages system dynamics. Though it may have several different meanings depending on the perspective, for this context we consider it to be a thinking process that promotes individual and organizational learning.

The software process is a real-time process with feedback control elements. Treating it as such allows us to explicitly consider its inputs, outputs, and feedback loops in order to better understand and control it. System dynamics is well suited for modeling these aspects of the software process control system.

An example of modeling Brooks's Law was introduced. The model illustrates an important management heuristic for late projects that is frequently misunderstood, and sheds light on some reasons why adding people to a late project can make it later. These dynamic effects are increased communication overhead, training losses, and lower productivity for new personnel. This modeling example will be broken down more in Chapter 2.

Software process technology involves process modeling, life-cycle models, and process improvement. A variety of simulation approaches and languages are available. Continuous systems modeling (such as with system dynamics) is generally more appropriate for strategic, macro-level studies. Discrete event modeling is usually better suited for studying detailed processes. Simulation tools have been improving over the years and often are used in a visual manner.

Software process simulation is increasingly being used for a wide variety of purposes. These include strategic management, planning and control, process improvement, and training. Process models can focus on narrow portions of the life cycle, entire projects, multiple projects, product lines, and wide organizational issues. Simulation in software development has been used for decades to support conceptual studies, requirements analysis, assess architectures, support testing, and verification and validation activities.

Process life-cycle models define process activities, their order and dependencies. There are many different life-cycle models available to be used. A risk-driven approach to deciding on an appropriate process will help ensure that the process best matches the project needs. Simulation can be used to assess candidate processes. It is also important to understand the different life-cycle characteristics to make better modeling process decisions (see Chapter 2 on the modeling process). In particular, the WinWin spiral life cycle is a good fit for many modeling situations.

Process frameworks like the CMMI and ISO are important vehicles for process improvement that are fully compatible with simulation. They identify best practices and provide paths for organizations to improve. Modeling and simulation support continuous process improvement every step of the way.

It is an exciting time in the software industry, with many opportunities, but techniques are needed to deal with the ever-increasing dynamic aspects and complexities of software processes. Project time lines are shortening, there are an increasing number of new processes and techniques, new technology keeps coming along, and projects keep increasing in complexity. We also need to keep focused on people and sustain a large enough pool of skilled workers to develop software in an efficient fashion that meets user needs. Simulation can help us see the forest through the trees, find the right balance of methods or people for particular project environments, and, ultimately, help software projects make better decisions.

1.19 EXERCISES

Some of the exercises in this chapter and others ask for analysis with respect to "your environment," which may take on different interpretations. If you are a practitioner, it may refer to the software process environment on your project or in your company, a government acquisition agency, an open-source project, or any software development you wish to study.

If you are a pure student, your assignment may be a specific project, company, or a segment of the software industry for case study. An assignment may also be to do the problems for more than one environment, then compare and analyze the differences.

1.1. List some benefits of using software process simulation in your environment.

1.2. Explain any impediments to using simulation for software processes in your environment. How might these impediments be mitigated?

1.3. What are the advantages and disadvantages of using simulation as an approach to a dynamics problem versus an analytical solution?

1.4. What is the difference between open and closed systems? Can you identify both kinds in your environment? You may use pictures to help illustrate your answer.

1.5. Identify organizational systems that may influence the software process in your enviroment, and especially focus on those that are not normally considered to have anything to do with software development or engineering.

1.6. Define the boundary of your process environment and create a comprehensive list of external factors that may influence the software process.

1.7. If you have experience in an environment in which software process modeling has been used, what have been the results? What went right and what went wrong? Provide lessons learned and suggest how things could have been improved.

1.8. What are the process state variables that are tracked in your environment?

1.9. Can you name any aspects of the software process that are completely static?

1.10. Can you name any aspects of the software process that are completely deterministic?

1.11. Identify some critical issues or problems in your environment that might be addressed by simulation and place them in the process modeling characterization matrix.

1.12. What other types of models and simulations are used in software development?

1.13. Draw a software process control system and expand/instantiate the inputs and outputs for your environment. For example, what is the specific source(s) of requirements, what types of resources are used, what standards are used, and what artifacts are produced?

1.14. Draw a software process control system for your environment, paying attention to the controllers. Does the depiction help show what maturity level you are at?

1.15. Review the table of system dynamics model elements and add more examples from the software process domain.

1.16. Trace the feedback loops in the simple project feedback model and in the Brooks's Law model. For each loop, outline the entire path of feedback, including the information connections, rates, levels, and auxiliaries in the path. Describe how the feedback loops impact the system behaviors.

1.17. In the Brooks's Law model, what are some other ways to add people besides an instantaneous pulse?

1.18. Write your own summary of the differences between the major life-cycle models, and outline them in a table format.

1.19. What are some drawbacks to the life-cycle process(es) currently being used in your environment or elsewhere?

1.20. Explain whether your environment could settle on a single life-cycle model or whether it needs more than one.

1.21. Critically analyze the life-cycle comparison figure for the iterative versus waterfall process. Does it seem reasonable and match your experience? Why or why not?

1.22. Describe the pros and cons of using COTS software. What happens if COTS is no longer supported by the vendor? What must be anticipated before designing COTS into a system?

Advanced Exercises

1.23. Make a comprehensive list of software process measures and assess whether they are continuous or discrete. For those that are deemed discrete, discuss whether they can be modeled as continuous for the purpose of understanding process dynamics. Identify the goals and constraints for that decision.

1.24. Identify some general software development best practices and discuss how they would apply to system dynamics model development (you may refer to Chapter 2 if not familiar yet with the modeling process).

1.25. Identify significant issues involved in verifying and validating system dynamics models (you may also get some ideas from Chapter 2). Give suggestions on how the issues can be dealt with and overcome so that system dynamics models will be accepted more readily by practitioners and researchers.

1.26. Suggest modifications to the simple project feedback model to make it more realistic.

1.27. Suggest modifications to the Brooks's Law model to make it more realistic.

1.28. The Brooks's Law model only varied the number of new personnel, but the time of adding new people is another dimension to the problem. What other factors might come into play in terms of optimizing the schedule finish? Might there be more than one local maximum across the different dimensions?

a) Define an experiment for future implementation to find the optimal solutions in Brooks's Law situations.

b) Sketch the multidimensional space of optimization and make conjectures as to the shape of the optimization curves.

1.29. Do some research and write a report on quantitative software process modeling that occurred before Abdel-Hamid's work. How much of it is still being (re)used? What has been the impact? Have any good practices gone unnoticed?

1.30. Perform a research study on the effects of implementing CMM, CMM-I, ISO, Six Sigma, or other process frameworks. What are the quantitative impacts to the software process?

1.31. Explain any relationships between the iterative activities in Figure 1.12 and the dynamics in Figure 1.15.

1.32. Read *Software Quality Management, Volume 1, Systems Thinking* by Weinberg and write a report on the applicability of the feedback examples in your environment.

1.33. Read *The Fifth Discipline* by Peter Senge and write a report on the applicability to software process improvement. Do any of the system archetypes apply to software processes? Explain.

1.34. Expand on the section regarding challenges for the software industry.