
MODEL STRUCTURES AND BEHAVIORS FOR SOFTWARE PROCESSES

3.1 INTRODUCTION

This chapter presents model structures for software processes starting with a review of elemental components, incorporating them into basic flow structures and building up to larger infrastructures. The structures and their behaviors are process patterns that frequently occur. The recurring structures are model “building blocks” that can be reused. They provide a framework for understanding, modifying, and creating system dynamics models regardless of modeling experience. With access to reusable formulations that have been repeatedly proven, previous work can be understood easier and the structures incorporated into new models with minimal modification.

Below is an overview of terminology related to model structures and behavior:

- *Elements* are the smallest individual pieces in a system dynamics model: levels, rates, sources/sinks, auxiliaries, and feedback connections.
- *Generic flow processes* are small microstructures and their variations comprised of a few elements, and are sometimes called *modeling molecules* [Hines 2000]. They are the building blocks, or substructures from which larger structures are created and usually contain approximately two to five elements. They produce characteristic behaviors.
- *Infrastructures* refer to larger structures that are composed of several microstructures, typically producing more complex behaviors.

- *Flow chains* are infrastructures consisting of a sequence of levels and rates (stocks and flows) that often form a backbone of a model portion. They house the process entities that flow and accumulate over time, and have information connections to other model components through the rates.

Not discussed explicitly in this chapter are *archetypes*. They present lessons learned from dynamic systems with specific structures that produce characteristic modes of behavior. The structures and their resultant dynamic behaviors are also called patterns. Whereas molecules and larger structures are the model building blocks, archetypes interpret the generic structures and draw dynamic lessons from them. Senge discusses organizational archetypes based on simple causal loop diagrams in *The Fifth Discipline* [Senge 1990].

An object-oriented software framework is convenient for understanding the model building blocks and their inheritance relationships described in this chapter. Consider a class or object to be a collection of model elements wired in a way that produces characteristic behavior. Figure 3.1 shows the model structures in a class hierarchy with inheritance. Object instances of these generic classes are the specific structures used for software process modeling (e.g., software artifact flows, project management policies, personnel chains, etc.).

The specific structures and their respective dynamic behaviors are the inherited attributes and operations (likened to services or methods). The hierarchy in Figure 3.1 also shows multiple inheritance since some infrastructures combine structure and behavior from multiple generic classes. Not shown are the lower levels of the hierarchy consisting of specific software process instances that all inherit from this tree.

The simplest system is the rate and level combination, whereby the level accumulates the net flow rate (via integration over time). It can be considered the super class.

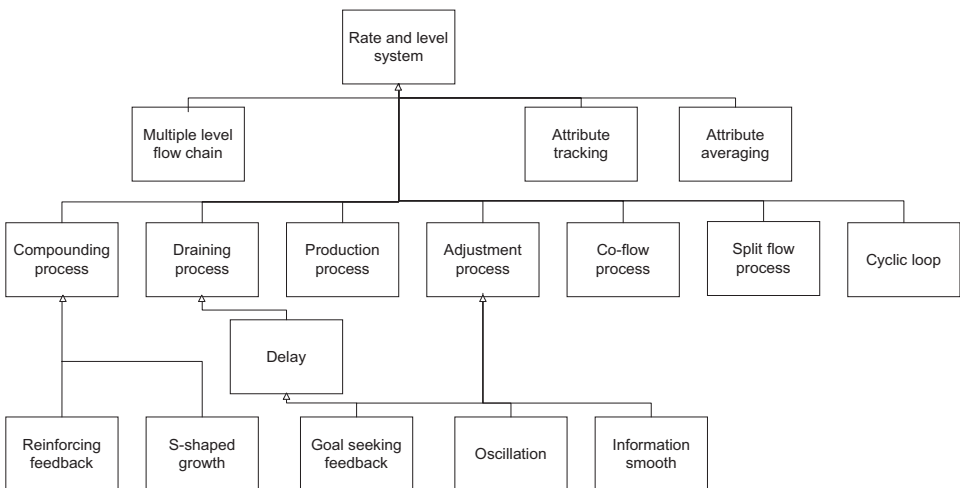


Figure 3.1. Class hierarchy for model structures.

The next level of structures include the generic flow processes, which are all slight variants on the rate and level system. Each of them adds some structure and produces unique characteristic behavior. For example, the compounding process adds a feedback loop from the level to the rate with an auxiliary variable that sets the rate of growth. The new behavior derived from this structure is an exponential growth pattern.

This hierarchy only includes systems explicitly containing rates and levels. There are also structures using auxiliary variables instead of levels that can produce similar dynamics. One example is the adjustment process to reach a goal that could operate without levels. These instances will be identified in their respective sections, and there are also some stand-alone infrastructures presented without levels at all. Normally, these structures would be enhanced with levels and rates in full models. Only the simplest of software process models would not contain any levels and would be of very limited use in assessing dynamic behavior.

Simulation toolsets and this book provide modeling molecules, infrastructures, and whole models that can be reused and modified. The modeler is encouraged to leverage these opportunities, as applicable, for potentially quicker development. For more on how the object-oriented modeling concept can be extended and automated for modelers, see the exercises at the end of this chapter and further discussion in Chapter 7.

A related technique is *metamodeling* [Barros et al. 2006a], which is a domain-specific methodology for creating software process structures. They are system dynamics extensions providing a high-level representation of application domain models for model developers. See [Barros et al. 2006b] for examples of metamodels created specifically for software process modeling, and the annotated bibliography for more references.

Next in this chapter is a review of the basic model elements. Then generic flows and infrastructures will be described. Specific structures for software process models and some behavioral examples will be presented. Virtually all of the structures are derived from one or more generic structures. Each structure will be presented with a diagram, summary of critical equations, and optionally behavioral output if it is new or unique.

3.2 MODEL ELEMENTS

The basic elements of system dynamics models previously described in Chapters 1 and 2 are levels, flows, sources/sinks, auxiliaries, and connectors. These are briefly reviewed below with lists of sample instantiations for software processes.

3.2.1 Levels (Stocks)

Levels are the state variables representing system accumulations. Typical state variables are software work artifacts, defect levels, personnel levels, or effort expenditure. Examples of software process level instances include:

- Work artifacts like tasks, objects, requirements, design, lines of code, test procedures, reuse library components, or documentation pages—these can be new,

reused, planned, actual, and so on. Sublevels like high-level design could be differentiated from low-level design.

- Defect levels—these can be per phase, activity, severity, priority or other discriminators. Note that the term “error” is sometimes used in models instead of defect; the terms are interchangeable unless clarified otherwise for a particular application.
- Personnel levels—often segregated into different experience or knowledge pools (e.g., junior and senior engineers)
- Effort and cost expenditures
- Schedule dates
- Personnel attributes such as motivation, staff exhaustion, or burnout levels
- Process maturity
- Key process areas
- Process changes
- Others

Other accumulations include financial and other business measures for a project (or ongoing organizational process). Relevant levels may include:

- Revenue
- Cumulative sales
- Market share
- Customers
- Orders
- Inventory
- Others

However, the value of software does not always reduce to monetary-related figures. In some instances, the value is derived from other measures such as in military systems where threat deterrence or strike capability is desired. Product value attributes such as quality, dependability, reliability, security, and privacy come into play. Thus, there are many more potential level instances that may play a part in value-based software engineering applications.

If one considers what tangible level items can be actually counted in a software process, levels are naturally aligned with artifacts available in files, libraries, databases, controlled repositories, and so on. Applying the snapshot test from Chapter 2 would lead to the identification of these artifact collections as levels. A process that employs configured baselines for holding requirements, design, code, and so on provides low-hanging fruit (process and project data) for model calibration and validation. Similarly, a trouble report database provides time trends on the levels of found defects. Thus, standard software metrics practices conveniently support system dynamics modeling. Refer to the GQM discussions in Chapter 2 to better align the metrics and modeling processes.

3.2.1.1 Sources and Sinks

Recall that sources and sinks represent levels or accumulations outside the boundary of the modeled system. Sources are infinite supplies of entities and sinks are repositories for entities leaving the model boundary. Typical examples of software process sources could be requirements originating externally or outsourced hiring pools. Sinks could represent delivered software leaving the process boundary or personnel attrition repositories for those leaving the organization. More examples include:

- Sources of requirements or change requests (product and process)
- Software delivered to customers and the market in general
- Software artifacts handed off to other organizations for integration or further development
- Employee hiring sources and attrition sinks
- Others

3.2.2 Rates (Flows)

Rates in the software process are necessarily tied to the levels. Levels do not change unless there are flow rates associated with them. Each of the level instances previously identified would have corresponding inflow and outflow rates. Their units are their corresponding level unit divided by time. A short list of examples include:

- Software productivity rate
- Software change rate
- Requirements evolution
- Defect generation
- Personnel hiring and deallocation
- Learning rate
- Process change rate
- Perception change
- Financial burn rate
- Others

3.2.3 Auxiliaries

Auxiliaries describe relationships between variables and often represent “score-keeping” measures. Examples include communication overhead as a function of people, or tracking measures such as progress (percent of job completion), percent of tasks in certain states, calculated defect density, other ratios, or percentages used as independent variables in dynamic relationships. Example variables include:

- Overhead functions
- Percent of job completion

- Quantitative goals or planned values
- Constants like average delay times
- Defect density
- Smoothed averages
- Others

3.2.4 Connectors and Feedback Loops

Information linkages can be used for many different purposes in a software process model. They are needed to connect rates to levels and auxiliaries. They are used to set the rates and provide inputs for decision making. Rates and decision functions, as control mechanisms, often require feedback connectors from other variables (usually levels or auxiliaries) for decision making. Examples of such information include:

- Progress and status information for decision making
- Knowledge of defect levels to allocate rework resources
- Conditions and pressures for adjusting processes
- Linking process parameters to rates and other variables (e.g., available resources and productivity to calculate the development rate)
- Others

Feedback in the software process can be in various forms across different boundaries. Learning loops, project artifacts, informal hall meetings, peer review reports, project status metrics, meetings and their reports, customer calls, newsletters, outside publications, and so on can provide constructive or destructive feedback opportunities. The feedback can be within projects, between projects, between organizations, and with other various communities.

3.3 GENERIC FLOW PROCESSES

Generic flow processes are the smallest essential structures based on a rate/level system that model common situations and produce characteristic behaviors. They consist of levels, flows, sources/sinks, auxiliaries, and, sometimes, feedback loops. Most of the generic processes in this section were previously introduced and some were shown as examples in Chapter 2. Each generic process can be used for multiple types of applications.

This section highlights the elemental structures and basic equations for each generic process. They are elaborated with specific details and integrated with other structures during model development. The inflow and outflow rates in the simplified equations represent summary flows when there are more than one inflow or outflow.

3.3.1 Rate and Level System

The simple rate and level system (also called stock and flow) is shown in Figure 3.2 and was first introduced in Chapter 2. It is a flow process from which all of the prima-

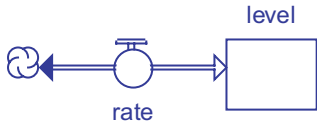


Figure 3.2. Rate and level system.

ry structures are derived. This system has a single level and a bidirectional flow that can fill or drain the level. It can be considered a super class for subsequent structures in this chapter, because each one builds on top of this basic structure with additional detail and characteristic behavior.

This system can also be represented with separate inflows and outflows, and very frequently in a modeling application it is best to consider the flows separately. However, they can always be algebraically combined into a single net flow for the simplest representation and, thus, the generic structure here is shown with a single flow.

The equations for this system are the standard-level integration equations automatically produced by the simulation software when the level and rate are laid down. As a review, the generated equation that applies for any such level is the following:

$$\text{level}(\text{time}) = \text{level}(\text{time} - dt) + \text{rate} \cdot dt$$

When there are separate inflows and outflows the equation becomes

$$\text{level}(\text{time}) = \text{level}(\text{time} - dt) + (\text{inflow rate} - \text{outflow rate}) \cdot dt$$

The inflow and outflow rates in the above equation represent summary flows if there are more than one inflow or outflow. No specific behavior is produced except that that level accumulates the net flow over time. The behavior of interest comes after specifying the details of the rate equation(s) and potential connections to other model components as shown in the rest of the generic flow processes.

3.3.2 Flow Chain with Multiple Rates and Levels

The single rate and level system can be expanded into a flow chain incorporating multiple levels and rates. It can be used to model a process that accumulates at several points instead of one, and is also called a cascaded level system. Figure 3.3 shows an example. The levels and rates always alternate. One level’s outflow is the inflow to an-

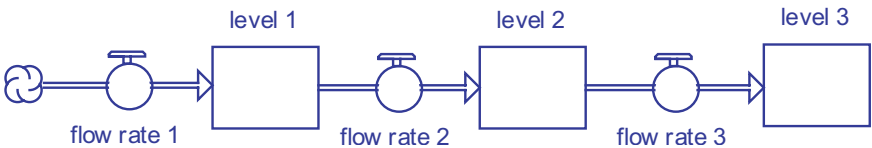


Figure 3.3. Flow chain.

other level, and so on down the chain. Each additional level in a system requires at least one more rate to be added accordingly between successive levels. The number of contained levels determines the order of a system. The dynamics get far more complex as the system order increases.

The end of a multiple level flow chain may also empty into a sink instead of having an end accumulation, and each end flow can also be bidirectional. A generic flow chain within itself does not produce characteristic behavior, like the single rate and level system. It requires that other structures and relationships be specified first.

The flow chain is a primary infrastructure for software process models. Frequently, it is a core structure laid down before elaborating detailed model relationships. Subsequent sections will show specific flow chains and their variants for software artifacts, personnel, defects, and so on.

Multiple levels and rates can be in conservative or nonconservative chains. Flow chains are normally conservative, such as in Figure 3.3, whereby the flow is conserved within directly connected rates and levels in a single chain. There are also nonconserved flows with separate but connected chains. These may be more appropriate at times and will be described with examples for specific software process chains in later sections.

3.3.3 Compounding Process

The compounding structure is a rate and level system with a feedback loop from the level to an input flow, and an auxiliary variable representing the fractional amount of growth per period, as in Figure 3.4. A compounding process produces positive feedback and exponential growth in the level as previously described in Chapter 2. The growth feeds upon itself. Modeling applications include user bases, market dynamics, software entropy, cost-to-fix trends, interest compounding, social communication patterns (e.g., rumors, panic), and so on. The growth fraction need not be constant and can vary over time. It can become a function of other parameters. When growth declines instead of increasing continuously, an overshoot and collapse or S-shaped growth model can be employed with additional structure (see subsequent sections).

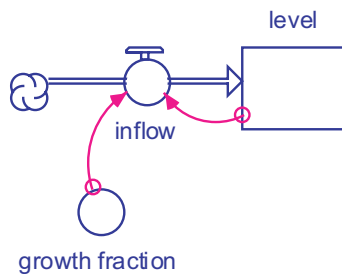


Figure 3.4. Compounding process. Equation:

$$\text{inflow} = \text{level} \cdot \text{growth fraction}$$

3.3.4 Draining Process

Draining can be represented similarly as the compounding process, except the feedback from the level is to an outflow rate and the auxiliary variable indicates how much is drained in the level, as in Figure 3.5. The variable may denote the fractional amount drained per period or it might represent a delay time, whereby the equation divides the level by the delay (see Section 3.4.3 on delays). Both representations produce the same behavior.

Draining is a common process that underlies delays and exponential decays. Delayed outflows and various orders of delays were shown in Chapter 2. In exponential decay, the draining fraction is often interpreted as a time constant that the level is divided by. Personnel attrition, promotion through levels, software product retirement, skill loss, and other trends can be modeled as draining processes.

3.3.5 Production Process

A production process represents work being produced at a rate equal to the number of applied resources multiplied by the resource productivity. Figure 3.6 shows the inflow to a level that represents production dependent on another level in an external flow chain representing a resource. Sometimes, the external resource is modeled with an auxiliary variable or fixed constant instead of a level. This software production infrastructure was introduced in Chapter 2. It can also be used for production of other assets such as revenue generation as a function of sales or several others.

3.3.6 Adjustment Process

An adjustment process is an approach to equilibrium. The structure for it contains a goal variable, a rate, level, and adjusting parameter, as in Figure 3.7. The structure models the closing of a gap between the goal and level. The change is more rapid at first and slows down as the gap decreases. The inflow is adjusted to meet the target goal, and the fractional amount of periodic adjustment is modeled with a constant (or possibly a variable).

The adjusting parameter may represent a fraction to be adjusted per time period (as in the figure) or it might represent an adjustment delay time. The only difference is that

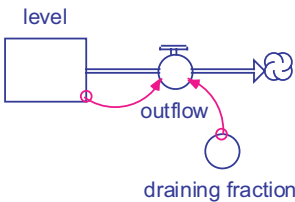


Figure 3.5. Draining process. Equation:

$$\text{outflow} = \text{level} \cdot \text{draining fraction}$$

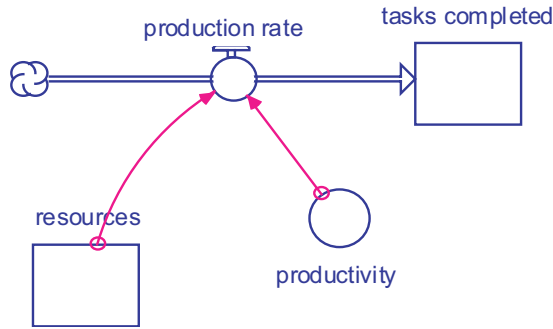


Figure 3.6. Production process. Equation:
 $\text{production rate} = \text{resources} \cdot \text{productivity}$

in the rate equation the goal gap is multiplied by an adjustment fraction or it is divided by an adjustment delay time.

This basic structure is at the heart of many policies and other behaviors. It can be used to represent goal-seeking behavior such as hiring and other goal-seeking negative feedback introduced in Chapter 2. The time constant in negative feedback is the reciprocal of the adjustment fraction (similar to the reciprocal of the draining process fraction parameter becoming the time constant for exponential decay). The structures can also model perceptions of quantities such as progress, quality, or reputation. The time constant (adjustment fraction reciprocal) represents the information delay in forming the perception.

Note that the most fundamental adjustment structure to close a goal gap does not require a level as in the figure. The current value of what is being controlled to meet a goal can be another variable instead of a level.

3.3.7 Coflow Process

A Coflow is a shortened name for a coincident flow, a flow that occurs simultaneously through a type of slave relationship. The coflow process has a flow rate synchronized

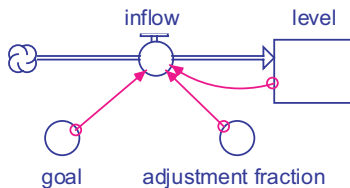


Figure 3.7. Adjustment process. Equation:
 $\text{inflow} = (\text{goal} - \text{level}) \cdot \text{adjustment fraction}$
 or
 $\text{inflow} = (\text{goal} - \text{level}) / \text{adjustment delay time}$

with another host flow rate, and normally has a conversion parameter between them as shown in Figure 3.8. This process can model the coflows of software artifacts and defects. It can also be used for personnel applications such as learning or frustration, resource tracking such as effort expenditure, or tracking revenue as a function of sales.

3.3.8 Split Flow Process

The split flow process in Figure 3.9 represents a flow being divided into multiple sub-flows or disaggregated streams. It contains an input level, more than one output flow, and typically has another variable to determine the split portions. It is also possible that the split fractions may be otherwise embedded in the rate equations.

The outflows may be modeled as delays (not shown in this figure) or they may be functions of other rates and variables. The operative equations must conserve the entire flow. Sometimes, the split flows can start from external sources instead of the original input level. Applications include defect detection chains to differentiate found versus escaped defects (i.e., defect filters), or personnel flows to model dynamic project resource allocation at given organizational levels.

3.3.9 Cyclic Loop

A cyclic loop representing entities flowing back through a loop is shown in Figure 3.10. The difference from nonclosed chains is that a portion of flow goes back into an originating level. The rate determining the amount of entities flowing back can take on any form, but must not violate flow physics, such as making the level go negative. This structure is appropriate to represent iterative software development processes, artifact rework, software evolution, and other phenomena.

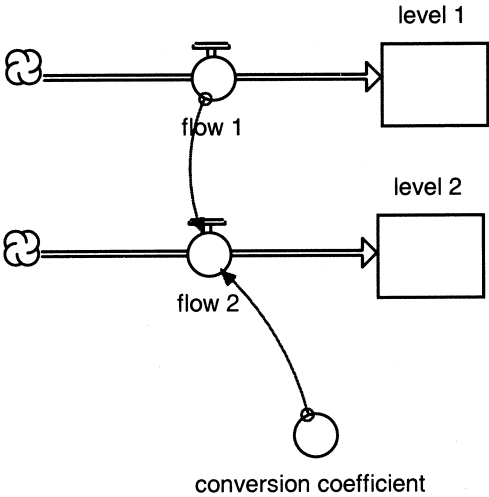


Figure 3.8. Coflow process. Equation:
 $\text{flow 2} = \text{flow 1} \cdot \text{conversion coefficient}$

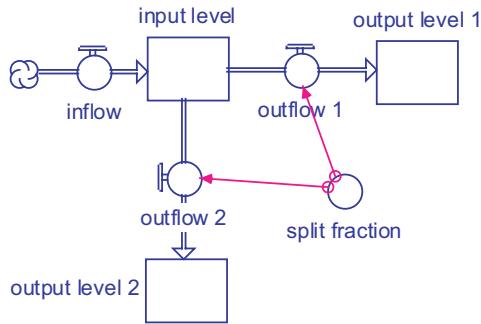


Figure 3.9. Split flow process. Equation:

$$\begin{aligned} \text{outflow 1} &= f(\text{split fraction}) \\ \text{outflow 2} &= f(1 - \text{split fraction}) \end{aligned}$$

3.4 INFRASTRUCTURES AND BEHAVIORS

The infrastructures in this section are based on one or more of the generic flow types with additional structural details. The additional structure typically leads to characteristic dynamic behaviors. A few of the structures herein do not cause specific dynamic behaviors, but instead are used for intermediate calculations, converters, or instrumentation of some kind. This section also builds on the introduction to general system behaviors from Chapter 2 and provides some software process examples.

3.4.1 Exponential Growth

Growth structures are based on the generic compounding flow process. Exponential growth was also introduced in Chapter 2. Figure 3.11 shows the exponential growth infrastructure and the equation that is equivalent to the compounding process. Exponential growth may represent market or staffing growth (up to a point usually representing

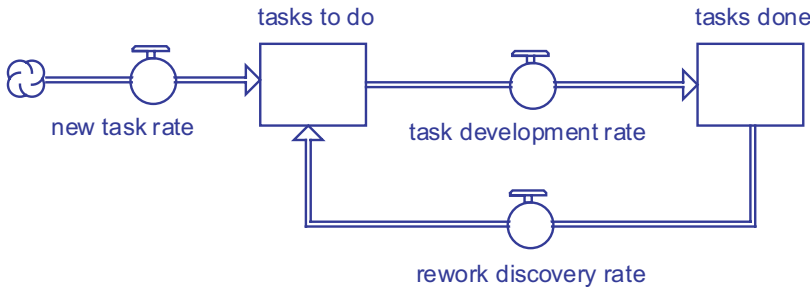


Figure 3.10. Cyclic loop process. The backflow rate formula has no special restrictions.

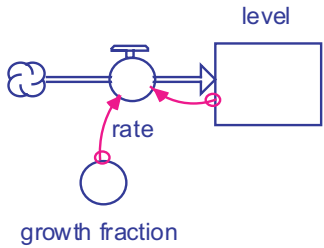


Figure 3.11. Exponential growth infrastructure. Equation:

$$\text{rate} = \text{level} \cdot \text{growth fraction}$$

saturation), software entropy, the steep rise of user populations, Internet traffic, defect fixing costs over time, computer virus infection levels, and so on.

As an example, exponential growth can model the user base of new innovative programs. The rate of new users depends on word-of-mouth reports from current users, the Internet, and other media channels. As the user base increases, so does the word-of-mouth and other communication of the new software capabilities. The amount of new users keeps building and the cycle continues. This was observed with the debuts of Web browsers and peer-to-peer file sharing programs.

Generally there are limits to growth. A system starts out with exponential growth and normally levels out. This is seen in the sales growth of a software product that early on shoots off like a rocket, but eventually stagnates due to satisfied market demand, competition, or declining product quality. The limits to growth for a user base are the available people in a population. For instance, exponential growth of Internet users has slowed in the United States but is still rising in less developed countries. An S-curve is a good representation of exponential growth leveling out. After the following examples of exponential growth, S-curves are discussed in the next section, 3.4.2.

3.4.1.1 Example: Exponential Growth in Cost to Fix

The structure in Figure 3.12 models exponentially increasing cost to fix a defect (or, similarly, the cost to incorporate a requirements change) later in a project lifecycle. It uses a relative measure of cost that starts at unity. The growth fraction used for the output in Figure 3.13 is 85% per year. The same structure can be used to model software entropy growth over time, where entropy is also a dimensionless ratio parameter.

3.4.2 S-shaped Growth and S-Curves

An S-shaped growth structure contains at least one level and provisions for a dynamic trend that rises and another that falls. There are various representations because S-curves may result from several types of process structures representing the rise and fall trends. The structure in Figure 3.14 contains elements of both the compounding process and draining process, for example. It does not necessarily have to contain both

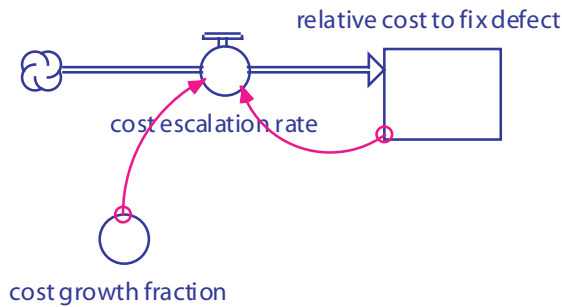


Figure 3.12. Cost-to-fix exponential growth structure.

of these structures, but there must be some opposing trends in the model to account for both the growth and declining portions of the S-curve dynamics. Frequently, there is gap closing that declines over time.

Sigmoidal curves are the result of growth that starts out small, increases, and then decreases again. This is reflected in its shape, which is flatter at the beginning and ends, and steeper in the middle. Frequently, there is a saturation effect in the system that eventually limits the growth rate. This may model technology adoption, staffing and resource usage dynamics, progress, knowledge diffusion, production functions over time, and so on.

An S-curve is seen in the graphic display of a quantity like progress or cumulative effort plotted against time that is S-shaped. S-shaped progress curves are often seen on

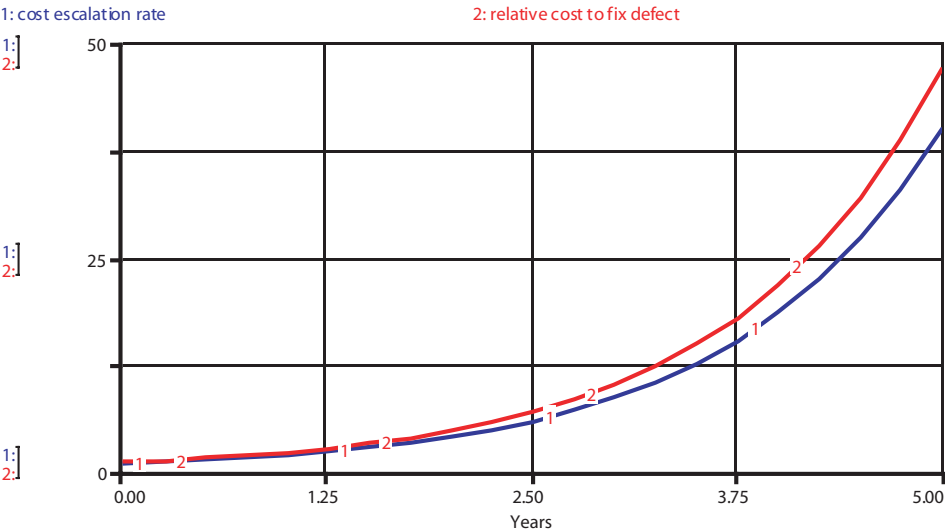


Figure 3.13. Exponential cost-to-fix behavior.

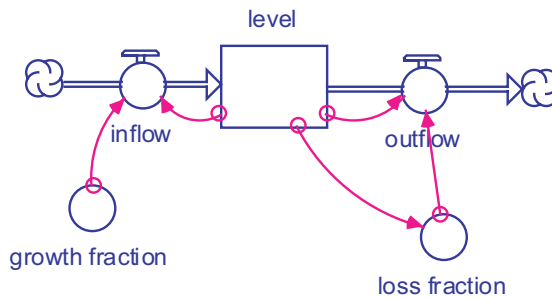


Figure 3.14. S-shaped growth infrastructure. Equations:

$$\text{inflow} = \text{level} \cdot \text{growth fraction}$$

$$\text{outflow} = \text{level} \cdot \text{loss fraction}$$

$$\text{loss fraction} = \text{graph}(\text{level})$$

projects because cumulative progress starts out slowly, the slope increases as momentum gains, then work tapers off. The cumulative quantity starts slowly, accelerates, and then tails off. See the Rayleigh curve example in Section 3.4.10 that produces an S-curve for cumulative effort expenditure.

S-curves are also observed in technology adoption and sales growth. The ROI of technology adoption is also S-shaped, whether plotted as a time-based return or as a production function that relates ROI to investment. See the exercise in this chapter for technology adoption.

S-curves are also seen in other types of software phenomena. For example, a new computer virus often infects computers according to an S-curve shape. The number of infected computers is small at first, and the rate increases exponentially until steps are taken to eliminate its propagation (or it reaches a saturation point). Applications in Chapters 4–6 show many examples of S-curves in cumulative effort, progress, sales growth, and other quantities.

3.4.2.1 Example: Knowledge Diffusion

The model in Figure 3.15 accounts for new knowledge transfer across an organization. It is a variant of the previous S-curve structure with two levels and a single rate that embodies growth and decline processes. It employs a word-of-mouth factor to represent the fractional growth, and a gap representing how many people are left to learn. The diminishing gap models saturation. The S-curve trends are shown in Figure 3.16. The infusion rate peaks at the steepest portion of the cumulative S-curve.

3.4.3 Delays

Delays are based on the generic draining process. Time delays were introduced in Chapter 2 as being present in countless types of systems. Figure 3.17 shows a simple

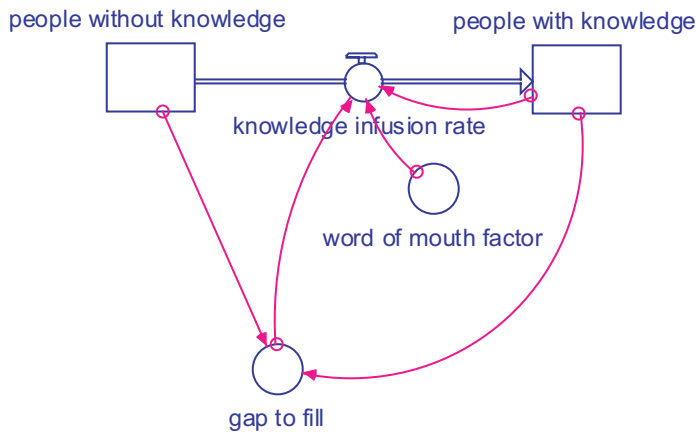


Figure 3.15. Knowledge diffusion structure. Equations:

knowledge infusion rate = (word-of-mouth factor · people with knowledge) · gap to fill
gap to fill = (people with knowledge + people without knowledge) – people with knowledge
word of mouth factor = 0.02

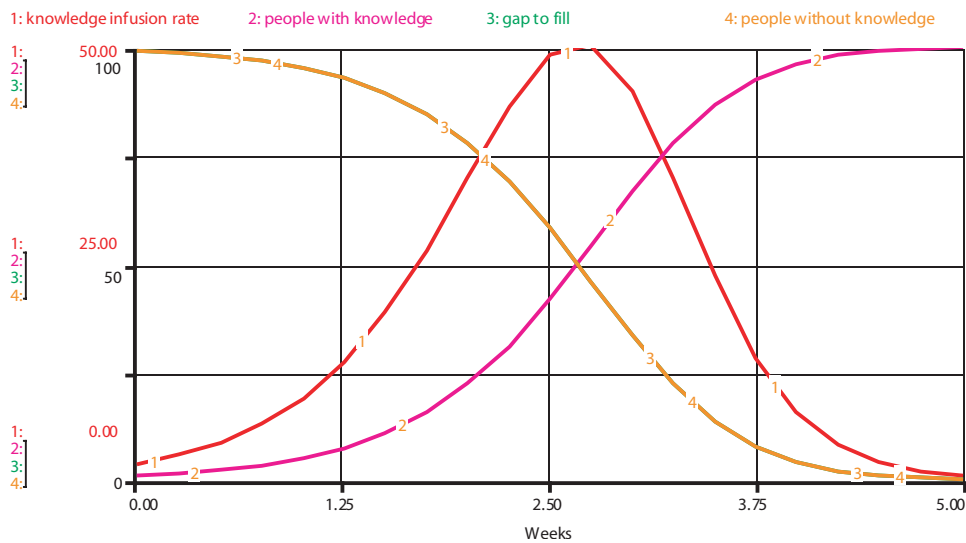


Figure 3.16. Knowledge diffusion S-curve behavior.

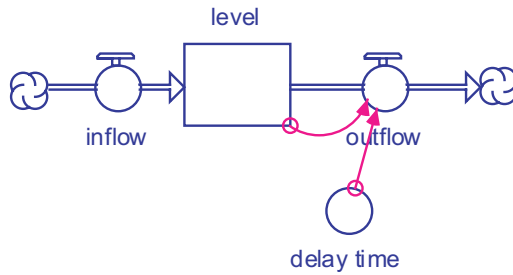


Figure 3.17. Delay structure. Equation:

$$\text{outflow} = \text{level} / \text{delay time}$$

delay structure. Hiring delays are one such common phenomenon in software development. A personnel requisition does not result in the immediate coming onboard of a new hire. There are often delays of weeks or months that should be accounted for (see Chapter 4 for suggested values). There are inherent delays in process data used for decision making, and distorted signals might be used. For example a project monitoring system that examines test reports on a monthly basis could miss an important problem spike early in the month, and visibility would come too late to bring the project back on track. Resource limitations can also induce large lags.

Other delays in development could occur at any manual step, including problem resolution, change approvals, signature cycles (the delay time is said to increase exponentially by 2^n , where n is the number of required signatures), and peer review artifact routing and handoffs. Even the existence of social networks within companies introduces delays, distorted signals, and even dropped information. Some employees act as bottlenecks of information, often knowingly for personal gain, whereas others willingly communicate vital information. Reorganizations and employee attrition also impact information transfer delays.

Communication delay also depends on the number of people. It takes much longer to disseminate information on large teams compared to small teams. This could have a profound impact on project startup, since the vision takes much longer to spread on a large team.

Overcorrection can result from excessive time delays in feedback loops and trying to fix a discrepancy between a goal and the apparent system state. Sometimes, test results or other crucial software assessment feedback comes very late, and often causes unnecessary and expensive rework.

At a detailed level, delays even exhibit seasonal behavior. For example, work is often left unattended on desks during peak vacation months. The month of December is typically a low-production month with reduced communication in many environments, as people take more slack time tending to their holiday plans. These seasonal effects are not usually modeled, but might be very important in some industries (e.g., a model for FedEx accounts for its annual December spike in IT work related to holiday shipping [Williford, Chang 1999]).

3.4.3.1 Example: Hiring Delay

Delays associated with hiring (or firing) are an aggregate of lags including realizing that a different number of people are needed, communicating the need, authorizing the hire, advertising and interviewing for positions, and bringing them on board. Then there are the delays associated with coming up to speed.

A first-order delay is shown in Figure 3.18 using personnel hiring as an example. The average hiring delay represents the time that a personnel requisition remains open before a new hire comes onboard. This example models a project that needs to ramp up from 0 to 10 people, with an average hiring delay of two months. This also entails balancing, or negative feedback, where the system is trying to reach the goal of the desired staffing level.

3.4.3.2 Exponential Decay

Exponential decay results when the outflow constant represents a time constant from a level that has no inflows. The decay declines exponentially toward zero. Both the level and outflow exhibit the trends. An old batch of defects that needs to be worked off is an example situation of decay dynamics. See Figure 3.20 for the structure and equation. It uses the same formula type as for delayed outflow. The graph of this system is a mirror of exponential growth that declines rapidly at first and slowly reaches zero.

Knowledge of time constant phenomena can be useful for estimating constants from real data, such as inferring values from tables or graphs. There are several interpretations of the time constant:

- The average lifetime of an item in a level
- An exponential time constant representing a decay pattern
- The time it would take to deplete a level if the initial outflow rate were constant

The time constant may represent things like the average time to fix a defect, delivery delay, average product lifetime, a reciprocal interest rate, and so on.

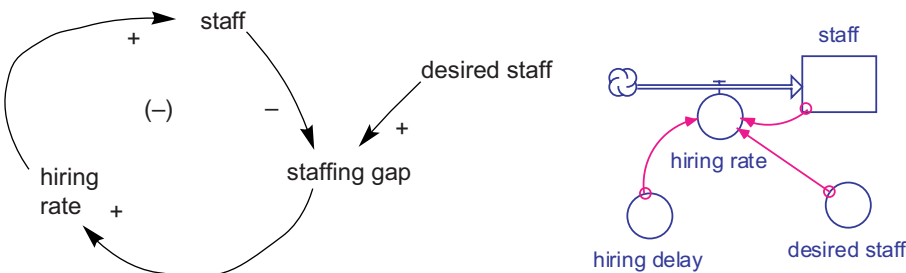


Figure 3.18. Goal-seeking hiring delay structure. Equation:

$$\text{hiring rate} = (\text{desired staff} - \text{staff}) / \text{hiring delay}$$

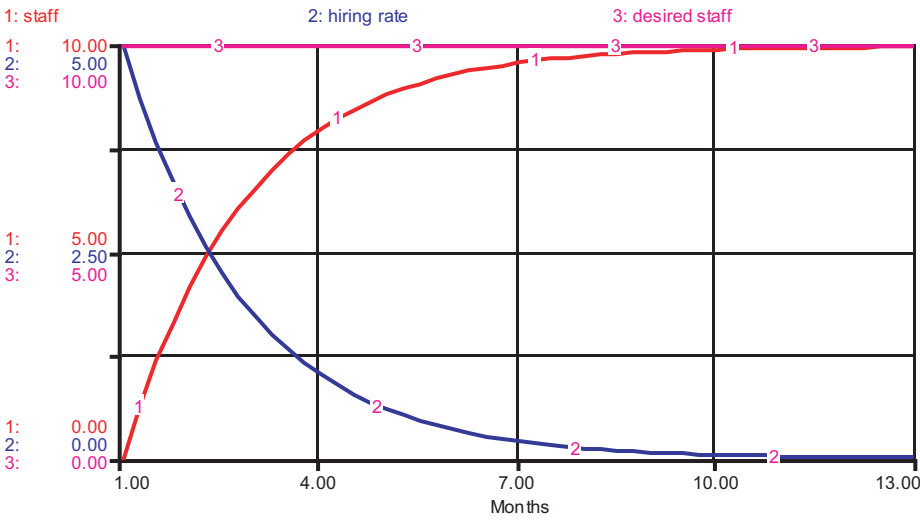


Figure 3.19. First-order hiring delay behavior.

3.4.3.3 Higher-Order (Cascaded) Delays

A higher-order delay behaves like a connected series of first-order delays. For example, a third-order delay can be visualized as the fluid mechanics of three water tanks in series. The first one flows into the second that flows into the third that empties into a sink. The first tank starts out full and the others empty when all the valves are opened. The dynamic level of the first tank is a first-order delay, the second tank level exhibits a second-order delay, and the third tank shows a third-order delay as they empty out.

Figure 3.21 shows the first-, second-, third-, and infinite-order delay responses to a pulse input, and Figure 3.22 shows the responses to a step input. The signals are applied at time = 1 day, and the delays are all 10 days. Figure 3.23 summarizes these delays in a generalized format.

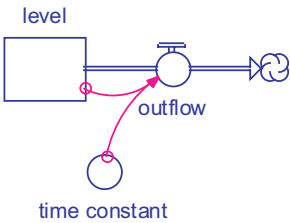


Figure 3.20. Exponential decay. Equation:

$$\text{rate} = \text{level}/\text{time constant}$$

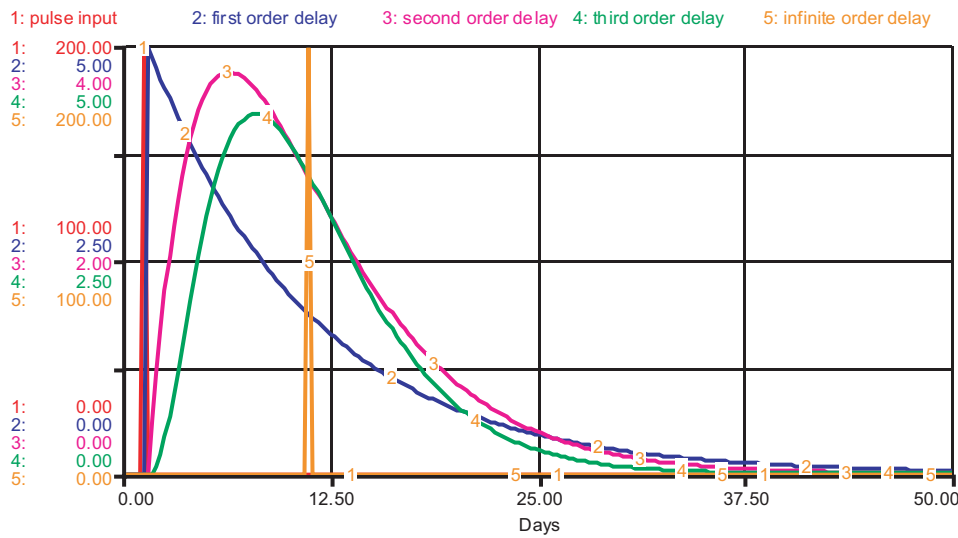


Figure 3.21. Delay responses to impulse.



Figure 3.22. Delay responses to step input.

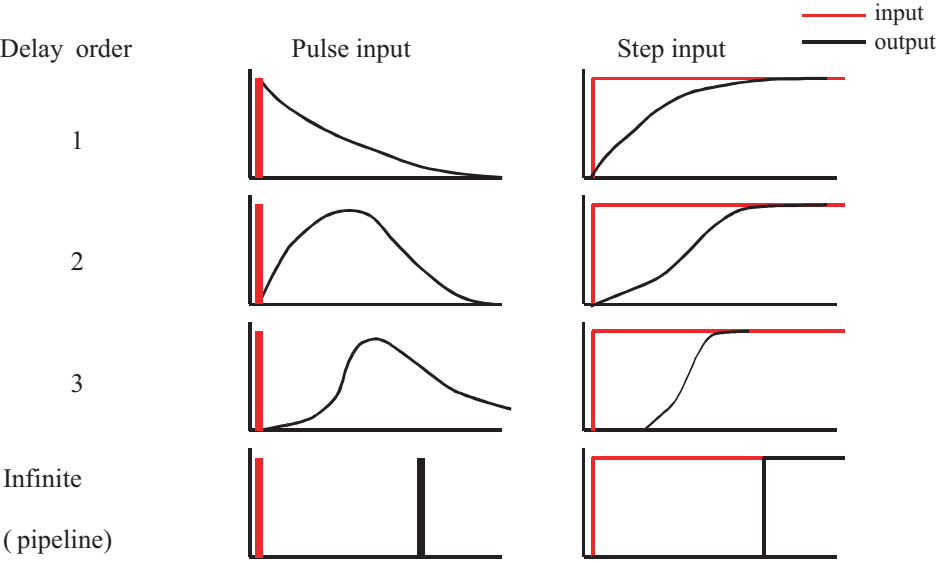


Figure 3.23. Delay summary.

Note that an infinite-order delay (pipeline) replicates the input signal (here a fourth-order delay was sufficient for demonstration purposes). It is called a pipeline delay because it reproduces the input signal but is shifted by the delay time. Information delays are not conservative like material delays.

In most situations, a first- or third-order delay is sufficient to model real processes. A third-order delay is often quite suitable for many processes, and delays beyond the third order are hardly ever used. For example, note that the third-order delay response looks like a Rayleigh curve. The staffing curve with a characteristic Rayleigh shape can be considered the system response to a pulse representing the beginning of a project (the “turn-on” signal or the allocated batch of funds that commences a project).

A cascaded delay is a flow chain with delay structures between the levels, such as in Figure 3.24. It is sometimes called an aging chain. It drains an initial level with a chain such that the final outflow appears different depending on the number of intervening delays (see Figure 3.21). With more levels, the outflow will be concentrated in the peak. Normally, having more than three levels in a cascaded delay will not dramatically change the resulting dynamics. This structure is frequently used to represent a workforce gaining experience.

3.4.4 Balancing Feedback

Balancing feedback (also called negative feedback) occurs when a system is trying to attain a goal, as introduced in Chapter 2. The basic structure and equation are repro-

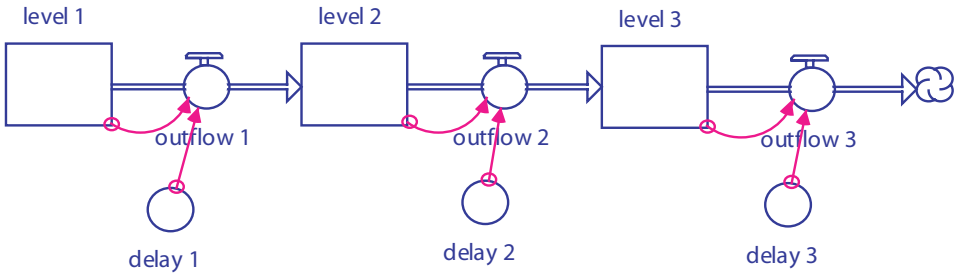


Figure 3.24. Cascaded delay (third order). Equations:

$$\text{outflow 1} = \text{level 1}/\text{delay 1}$$

$$\text{outflow 2} = \text{level 2}/\text{delay 2}$$

$$\text{outflow 3} = \text{level 3}/\text{delay 3}$$

duced in Figure 3.25. It is a simple approach to equilibrium wherein the change starts rapidly at first and slows down as the gap between the goal and actual state decreases. Figure 3.26 shows the response for a positive goal and Figure 3.27 for a zero goal.

Balancing feedback could represent hiring increases. A simple example of balancing feedback is the classic case of hiring against a desired staff level. See the example in Section 3.4.3.1 for a hiring adjustment process, which is also a first-order delay system. Balancing feedback is also a good trend for residual defect levels during testing, when defects are found and fixed; in this case, the goal is zero instead of a positive quantity.

There are different orders of negative feedback and sometimes it exhibits instability. The examples so far show first-order negative feedback. Figure 3.28 shows a notional second-order system response with oscillations. The oscillating behavior may start out with exponential growth and level out. We next discuss some models that produce oscillation.

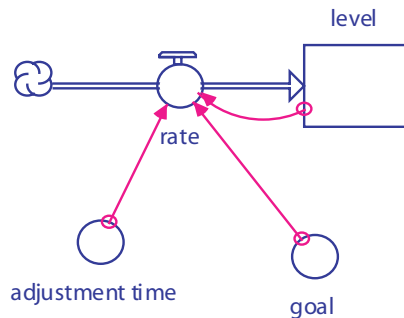


Figure 3.25. Balancing feedback structure. Equation:

$$\text{rate} = (\text{goal} - \text{level})/\text{adjustment time}$$

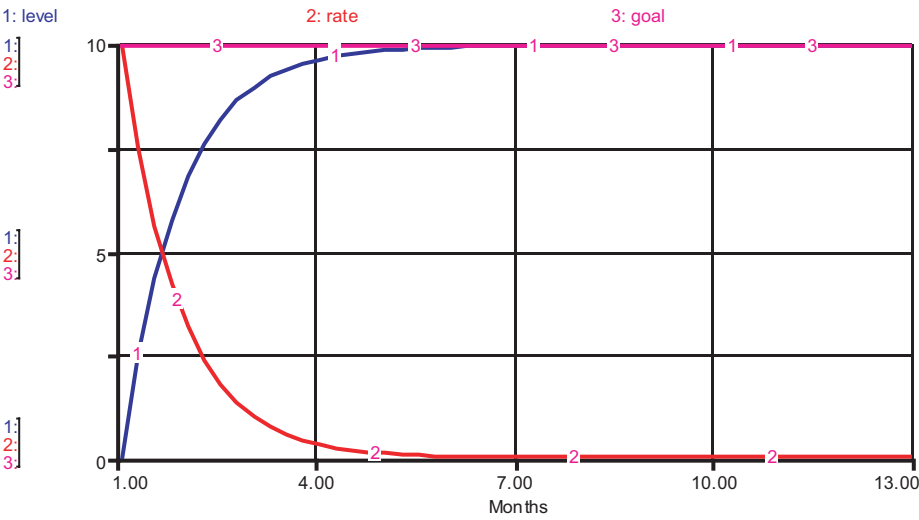


Figure 3.26. Goal-seeking balancing feedback (positive goal).

3.4.5 Oscillation

Oscillating behavior may result when there are at least two levels in a system. A system cannot oscillate otherwise. A generic infrastructure for oscillation is shown in Figure 3.29. The rate 1 and rate 2 factors are aggregates that may represent multiple factors. Normally, there is a parameter for a target goal that the system is trying to reach,

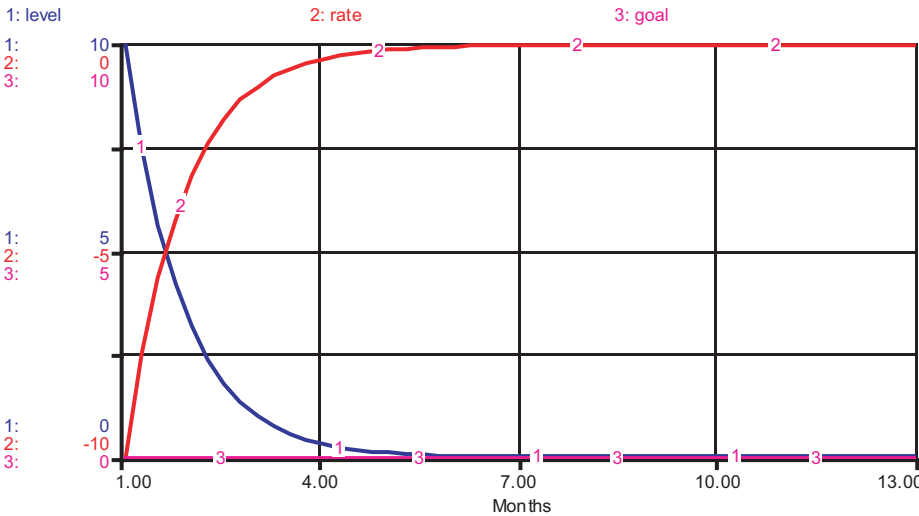


Figure 3.27. Goal-seeking balancing feedback (zero goal).

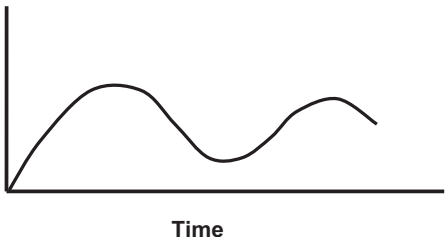


Figure 3.28. Second-order balancing feedback response.

and the system is unstable as it tries to attain the goal. Unsynchronized and/or long delays frequently cause the instability as the gap between actual and desired is being closed.

Such a system does not necessarily oscillate forever. Control can take over with changes in policy, which may or may not be reflected in a modified process structure. For example, there may be instability due to iteratively increasing demands, so management has to descope the demands. The oscillations will dampen and the system eventually level out when control takes over.

There are many different forms that the equations can take. Examples related to software production and hiring are shown below with their equations.

3.4.5.1 Example: Oscillating Personnel Systems

Instability may result when demand and resources to fill the demand are not synchronized. Figure 3.30 shows a simplified structure to model oscillating production and hiring trends. Figure 3.31 shows an oscillating system with more detail, demonstrating

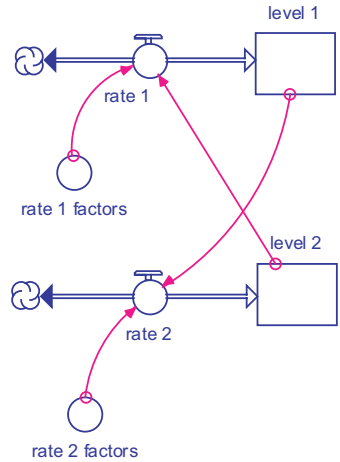


Figure 3.29. Oscillating infrastructure.

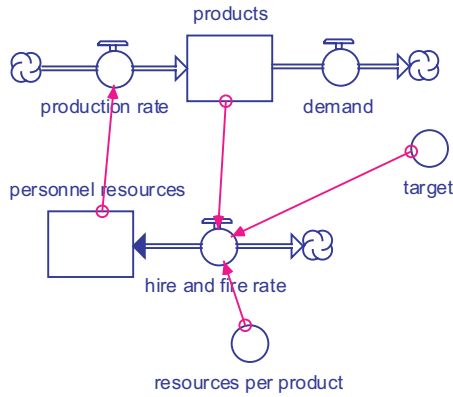


Figure 3.30. Oscillating production and hiring—simplified. Equation:

$$\text{hire and fire rate} = (\text{target} - \text{products}) \cdot \text{resources per product}$$

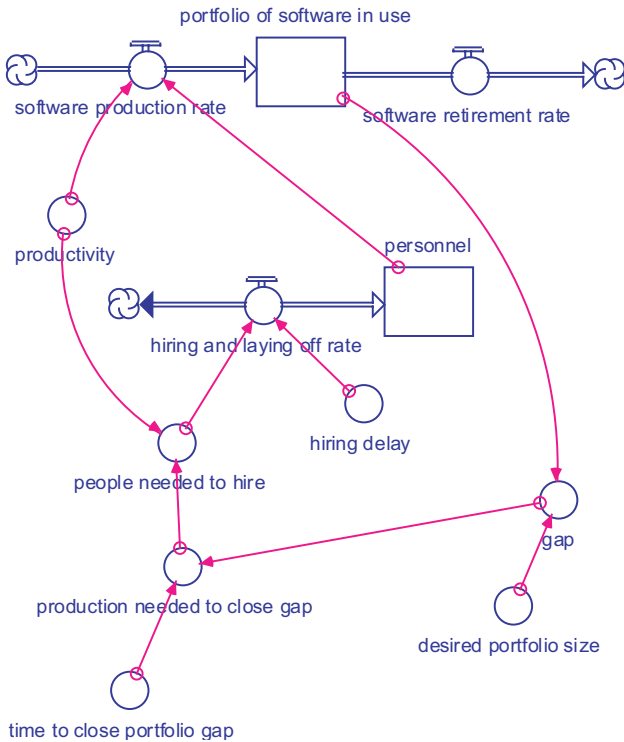


Figure 3.31. Oscillating production and hiring. Equations:

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

$$\text{hiring and laying off rate} = \text{people needed to hire} / \text{hiring delay}$$

$$\text{production needed to close gap} = \text{gap} / \text{time to close portfolio gap}$$

$$\text{people needed to hire} = \text{production needed to close gap} / \text{productivity}$$

hiring instability. An organization is trying to staff to an appropriate level to meet current demands for creating software to be used. If the gap for new software becomes too large, then new hiring is called for. Conversely, there are times when there is not enough work to go around and organizational layoffs must be undertaken. Figure 3.32 shows resulting dynamic behavior for the system.

A similar system at the organizational level would consider the number of software projects being undertaken. The inflow to the stock of projects would be the new project rate representing the approval and/or acquisition of new projects chartered to be undertaken. The outflow would be the project completion rate. Unstable oscillating behavior could result from trying to staff the projects. This chain could also be connected to the one in Figure 3.31 representing the software itself.

3.4.6 Smoothing

Information smoothing was introduced in Chapter 2 as an averaging over time. Random spikes will be eliminated when trends are averaged over a sufficient time period. The smoothed variables could represent information on the apparent level of a system as understanding increases toward the true value, as they exponentially seek the input signal. A generic smoothing structure is used to gradually and smoothly move a quantity toward a goal. Thus, it can be used to represent delayed information, perceived quantities, expectations, or general information smoothing.

Smoothing can be modeled as a first-order negative feedback loop as shown in Figure 3.33. There is a time delay associated with the smoothing, and the change toward the final value starts rapidly and decreases as the discrepancy narrows between the present and final values.

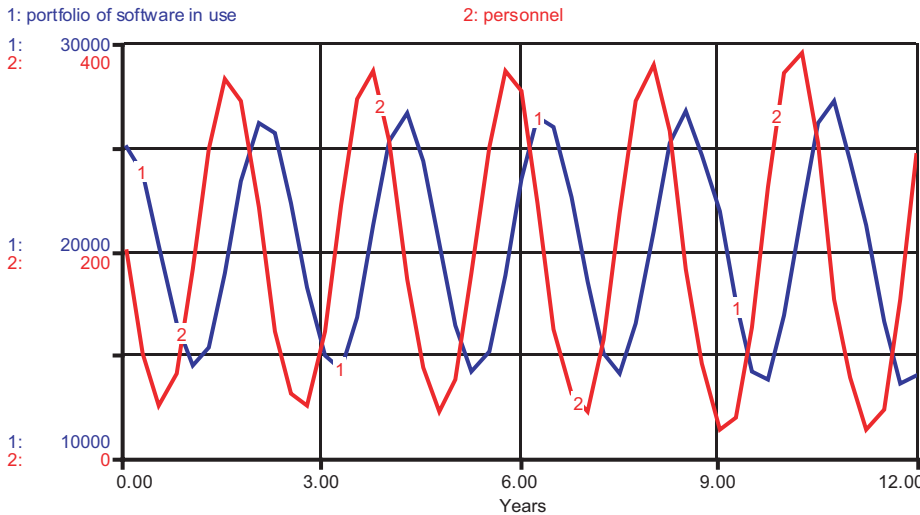


Figure 3.32. Oscillating portfolio and personnel dynamics.

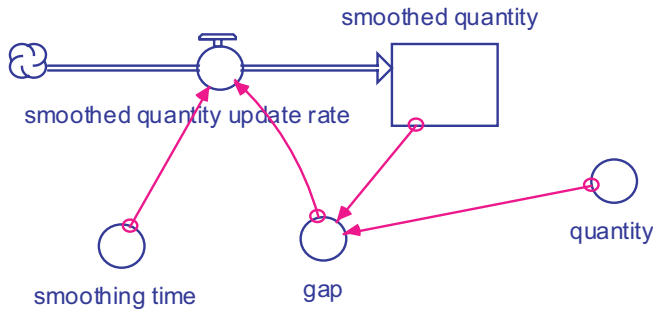


Figure 3.33. Information smoothing structure. Equations:
$$\text{smoothed quantity update rate} = \text{gap} / \text{smoothing time}$$
$$\text{gap} = \text{quantity} - \text{smoothed quantity}$$

The smoothed quantity update rate closes the gap over time. The gap is the difference between a goal and the smoothed quantity. There are performance trade-offs with the choice of smoothing interval. A long interval produces a smoother signal but with a delay. A very short interval reduces the information delay, but if too short it will just mirror the original signal with all the fluctuations.

3.4.6.1 Example: Perceived Quality

An example structure for information smoothing is shown in Figure 3.34, which models the perceived quality of a system. The same structure can be used to model other perceived quantities as smoothed variables. The operative rate equation for smoothing of quality information expresses the change in perceived quality as the difference be-

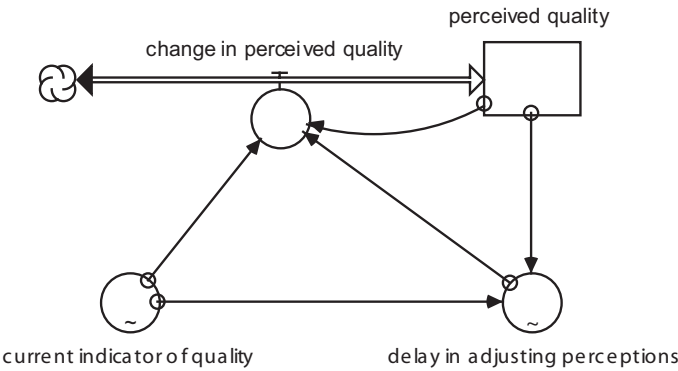


Figure 3.34. Perceived quality information smoothing. Equation:
$$\text{change in perceived quality} = (\text{current indicator of quality} - \text{perceived quality}) / \text{delay in adjusting perceptions}$$

tween current perceived quality and the actual quantity divided by a smoothing delay time.

The delay relationship is a graphical function that reflects an assumption about asymmetry in the adjustment of perceptions. It is very easy to modify perceptions downward, but far harder and more time-consuming to modify them upward. If a software product suddenly gets bad press about serious defects, then even if the problems are fixed it will take a long time for the public perception to change positively.

The graph in Figure 3.35 shows how the asymmetric delays impact the quality perception. When the actual quality indicator falls, then the perceived quality closely follows it with a short delay (bad news travels fast). But it is much harder to change the bad perception to a good one. It is seen that the delay is much greater when the actual quality improves, as it takes substantially longer time for the revised perception to take place.

3.4.7 Production and Rework

The classic production and rework structure in Figure 3.36 accounts for incorrect task production and its rework. Work is performed, and the percentage done incorrectly flows into undiscovered rework. Rework occurs to fix the problems at a specified rate. The work may cycle through many times. This structure is also related to the cyclic loop, except this variant has separate sources and sinks instead of directly connected flow chains. This is an important structure used in many models, including Abdel-Hamid’s integrated project model.

A number of other structures can be combined with this, such as using the production structure for the task development rate. The rate for discovering task rework and the fraction correct can be calculated from other submodels as well. Another variation

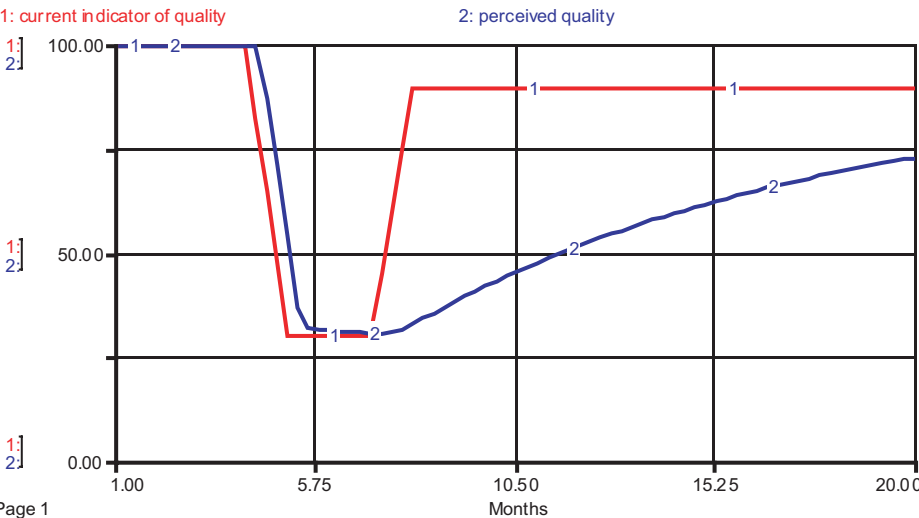


Figure 3.35. Output of perceived quality information smoothing.

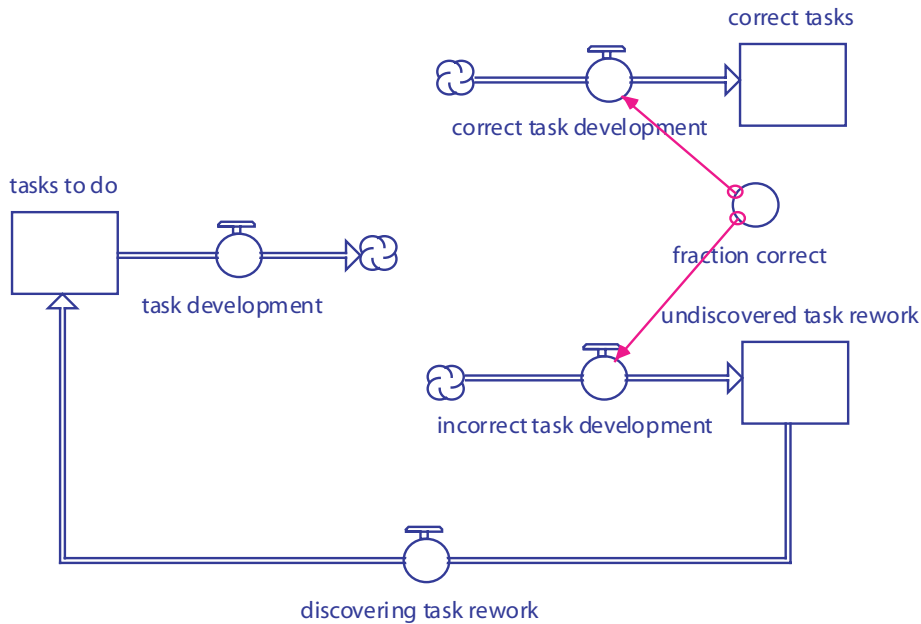


Figure 3.36. Production and rework structure. Equations:
 $\text{correct task development} = \text{task development} \cdot \text{fraction correct}$
 $\text{incorrect task development} = \text{task development} \cdot (1 - \text{fraction correct})$

would have discovered rework flow into a separate level than the originating one. This would allow for the differentiation of productivity and quality from the original tasks. A number of other variations are also possible. Structures for working off defects specifically (instead of tasks) are provided in later sections.

3.4.8 Integrated Production Structure

The infrastructure in Figure 3.37 combines elements of the task production and human resources personnel chains. Production is constrained by both productivity and the applied personnel resources external to the product chain. The software development rate equation is typically equivalent to the standard equation shown for the production generic flow process. As shown earlier in Figure 3.6, the level of personnel available is multiplied by a productivity rate.

3.4.9 Personnel Learning Curve

The continuously varying effect of learning is an ideal application for system dynamics. Figure 3.38 shows a classic feedback loop between the completed tasks and productivity.

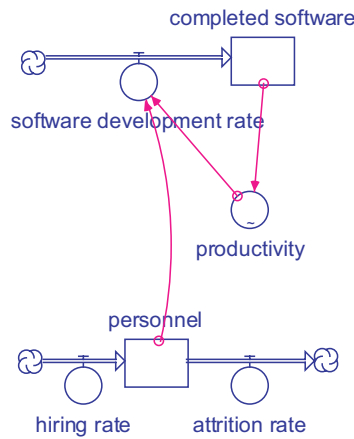


Figure 3.37. Production infrastructure with task and personnel chains. Equation:
$$\text{software development rate} = \text{personnel} \cdot \text{productivity}$$

One becomes more proficient at a task after repeated iterations of performing that task. This figure shows a representation in which the learning is a function of the percent of job completion. The learning curve can be handily expressed as a graph or table function.

Another formulation would eliminate the auxiliary for percentage complete and have a direct link between tasks completed and the learning curve. The learning would be expressed as a function of the volume of tasks completed. The learning curve is a

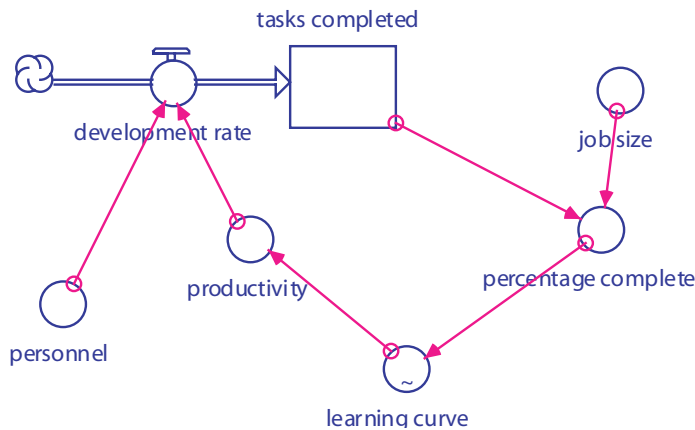


Figure 3.38. Learning curve. Equations:
$$\text{learning curve} = f(\text{percentage complete})$$
$$\text{productivity} = f(\text{learning curve})$$

complex area of research in itself. See Chapter 4 for a more detailed look at modeling this phenomenon and alternative formulations.

The learning curve function for this model is a graph relationship in Figure 3.39. It is expressed as a unitless multiplier to productivity as a function of project progress. It exhibits the typical S-curve shape whereby learning starts slow and the rate of learning increases and then drops off when reaching its peak near the end.

3.4.10 Rayleigh Curve Generator

The Rayleigh generator in Figure 3.40 produces a Rayleigh staffing curve. It contains essential feedback that accounts for the work already done and the current level of elaboration on a project. The output components of the Rayleigh generator are in Figure 3.41. The familiar hump-shaped curve for the staffing profile is the one for effort rate. The manpower buildup parameter sets the shape of the Rayleigh curve. Note also how cumulative effort is an S-shape curve. Since it integrates the effort rate, the steepest portion of the cumulative effort is the peak of the staffing curve.

This modeling molecule can be extended for many software development situations. The generator can be modified to produce various types of staffing curves, including those for constrained staffing situations, well-known problems, and incremental development. The Rayleigh curve is also frequently used to model defect levels. The defect generation and removal rates take on the same general shape as the effort rate curve. The Rayleigh curve and its components are further elaborated and used in Chapters 5 for defect modeling and in Chapter 6 for project staffing.

The standard Rayleigh curve in this section can be modified in several ways, as shown later in this book. It can become nearly flat or highly peaked per the buildup parameter, it can be scaled for specific time spans, clipped above zero on the beginning

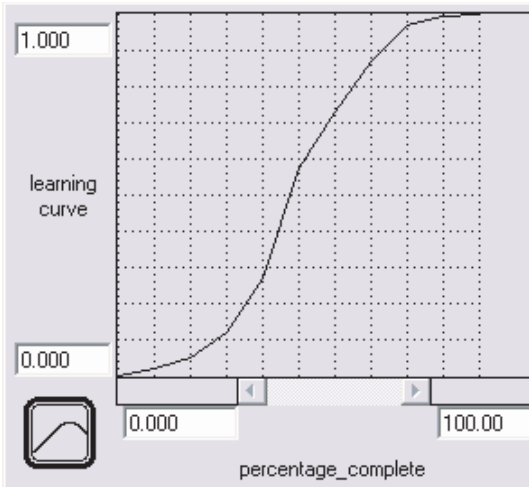


Figure 3.39. Learning curve function.

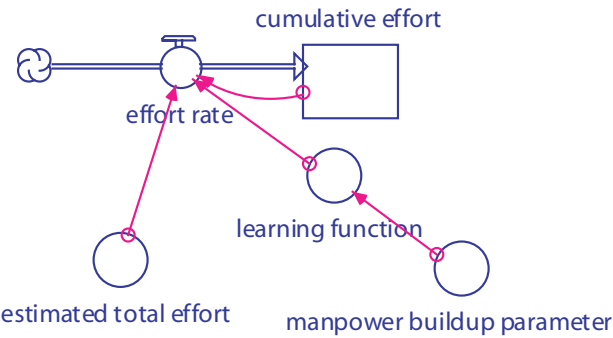


Figure 3.40. Rayleigh curve generator. Equations:
 $\text{effort rate} = \text{learning function} \cdot (\text{estimated total effort} - \text{cumulative effort})$
 $\text{learning function} = \text{manpower buildup parameter} \cdot \text{time}$

or end of the curve, offset by time, or superimposed with other Rayleigh curves. Applications in Chapters 5 and 6 illustrate some of these modifications.

3.4.11 Attribute Tracking

Important attributes to track are frequently calculated from levels. The structure in Figure 3.42 is used to track an attribute associated with level information (i.e., the state variables). This example calculates the normalized measure defect density by dividing

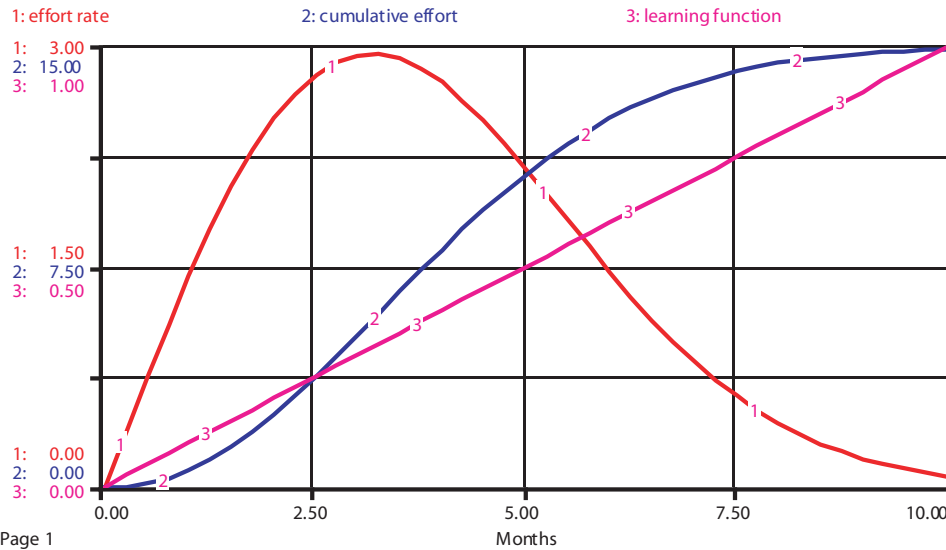


Figure 3.41. Rayleigh curve outputs.

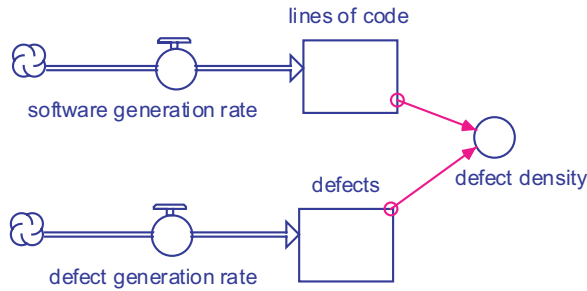


Figure 3.42. Attribute tracking infrastructure (defect density example). Equation:
$$\text{defect density} = \text{defects} / \text{lines of code}$$

the software size by the total number of defects. The defect density attribute can be used as an input to other model portions, such as a decision structure.

3.4.12 Attribute Averaging

A structure for attribute averaging (similar to attribute tracking) is shown in Figure 3.43. It calculates a weighted average of an attribute associated with two or more levels. This example calculates the weighted average of productivity for two pools of personnel. It can be easily extended for more entities to average across and for different weighting schemes.

3.4.13 Effort Expenditure Instrumentation

Effort or cost expenditures are coflows that can be used whenever effort or labor cost is a consideration. Frequently, this coflow structure serves as instrumentation only to

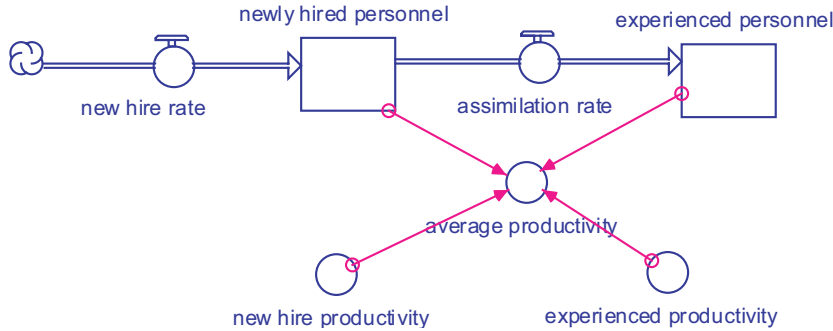


Figure 3.43. Attribute averaging infrastructure (software productivity example). Equation:
$$\text{average productivity} = [(\text{newly hired personnel} \cdot \text{new hire productivity}) + (\text{experienced personnel} \cdot \text{experienced productivity})] / (\text{newly hired personnel} + \text{experienced personnel})$$

obtain cumulative effort and does not play a role in the dynamics of the system. It could be used for decision making in some contexts. In Figure 3.44, the task production rate represents an alias computed in another part of the model. Using an alias is convenient for isolating the instrumentation, or the variable can be directly connected, depending on the visual diagram complexity. Effort accumulation rates can be calibrated against productivity for every activity modeled. Cost accumulation is easily calculated with labor rates.

If the number of people is represented as a level (set of levels) or another variable, then the effort expenditure rate (same as manpower rate) will be the current number of people per time period, as shown in Figure 3.45. The accumulated effort is contained in the level. In this example, the traditional units are difficult to interpret, as the rate takes on the value of the current personnel level. The rate unit still works out to effort per time.

3.4.14 Decision Structures

Example infrastructures for some important decision policies are described in this section.

3.4.14.1 Desired Staff

The structure in Figure 3.46 determines how many people are needed in order to meet a scheduled completion date. It is useful in project models where there is a fixed schedule and staffing decisions to be made. It takes into account the work remaining, allowable time to complete, and the current productivity. It then computes the desired project velocity to complete the project on time and how many people are needed for it. Desired project velocity has the units of tasks per time, such as use cases per week. This structure can be easily modified to determine the number of people to meet other quantitative goals besides schedule.

This is similar to a reverse of the production structure in that it figures out the desired productivity and staff level. This particular structure contains no levels, but levels could be used in place of some of the parameters. The maximum function for remaining dura-

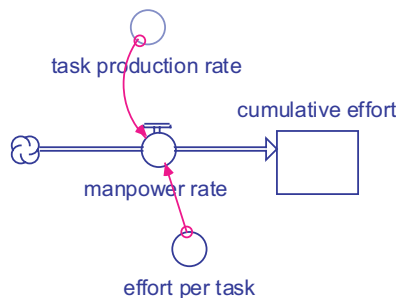


Figure 3.44. Effort expenditure coflow. Equation:

$$\text{manpower rate} = \text{task production rate} \cdot \text{effort per task}$$

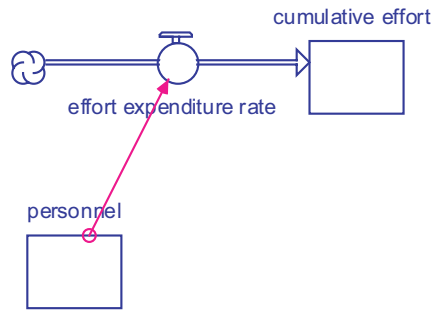


Figure 3.45. Effort expenditure linked to personnel level. Equation:
 $\text{effort expenditure rate} = \text{personnel}$

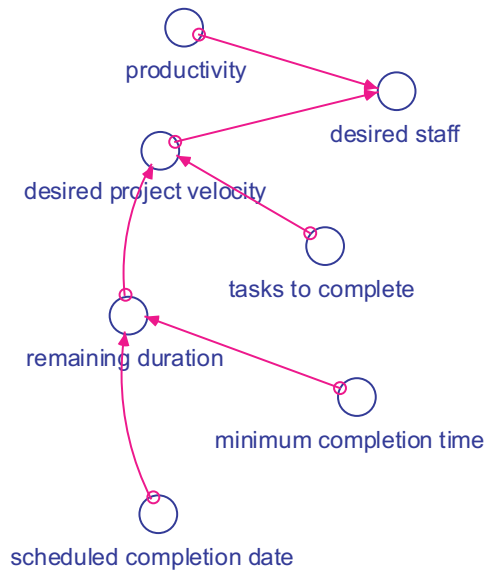


Figure 3.46. Desired staff structure. Equations:
 $\text{desired staff} = \text{desired project velocity} / \text{productivity}$
 $\text{desired project velocity} = \text{tasks to complete} / \text{remaining duration}$
 $\text{remaining duration} = \text{maximum (minimum completion duration, scheduled completion date - time)}$

tion ensures that time does not become so small that the desired staff becomes unreasonably large. A perceived productivity could also be used in place of actual productivity.

3.4.14.2 Resource Allocation

Project and organizational management need to allocate personnel resources based on relative needs. The infrastructure in Figure 3.47 supports that decision making. Tasks with the greatest backlog receive proportionally greater resources (“the squeaky wheel gets the grease”). This structure will adjust dynamically as work gets accomplished, backlogs change, and productivity varies. Many variations on the structure are possible (see the chapter exercises).

3.4.14.3 Scheduled Completion Date

The structure in Figure 3.48 represents the process of setting a project completion date, and it can be easily modified to adjust other goals/estimates. A smoothing is used to

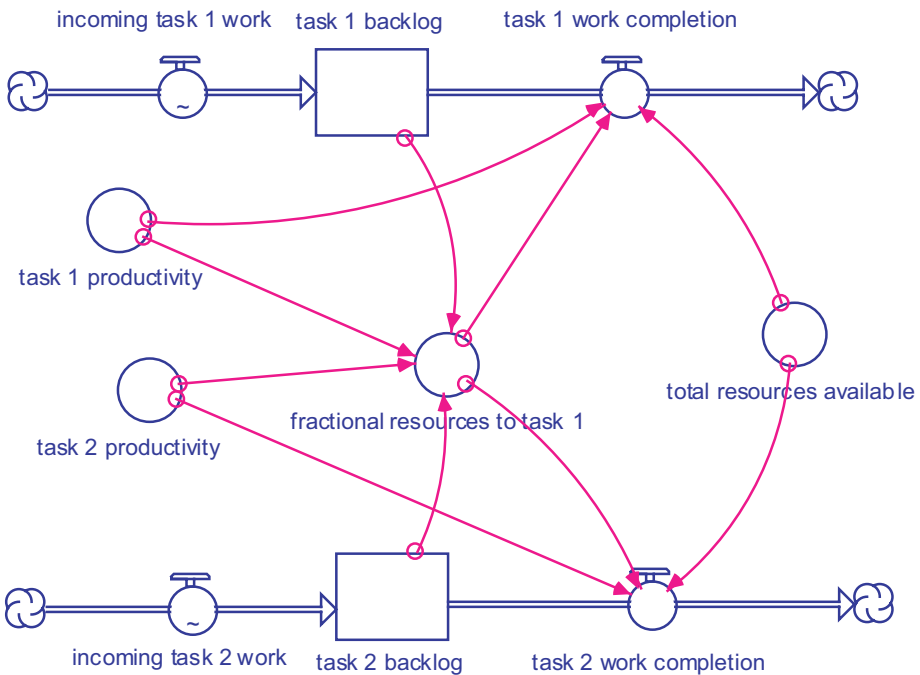


Figure 3.47. Resource allocation infrastructure. Equations:

$$\text{fractional resources to task 1} = \frac{(\text{task 1 backlog} / \text{task 1 productivity})}{[(\text{task 1 backlog} / \text{task 1 productivity}) + (\text{task 2 backlog} / \text{task 2 productivity})]}$$

$$\text{task 1 work completion} = (\text{total resources available} \cdot \text{fractional resources to task 1}) \cdot \text{task 1 productivity}$$

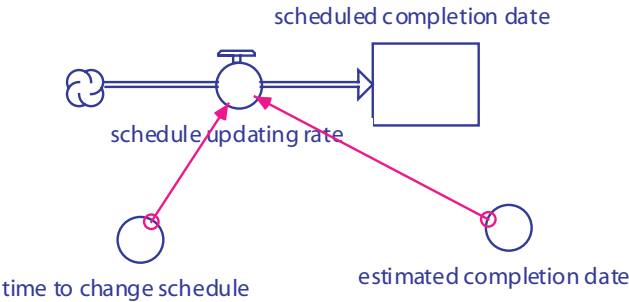


Figure 3.48. Scheduled completion date. Equation:
$$\text{schedule updating rate} = (\text{estimated completion date} - \text{scheduled completion date}) / \text{time to change schedule}$$

adjust the scheduled completion date toward the estimated completion date. The estimated date could be calculated elsewhere in a model.

3.4.14.4 Defect Rework Policies

The capability to assess cost/quality trade-offs is inherent in system dynamics models that represent defects as levels, and include the associated variable effort for rework and testing as a function of those levels. Figure 3.49 shows an example with an implicit

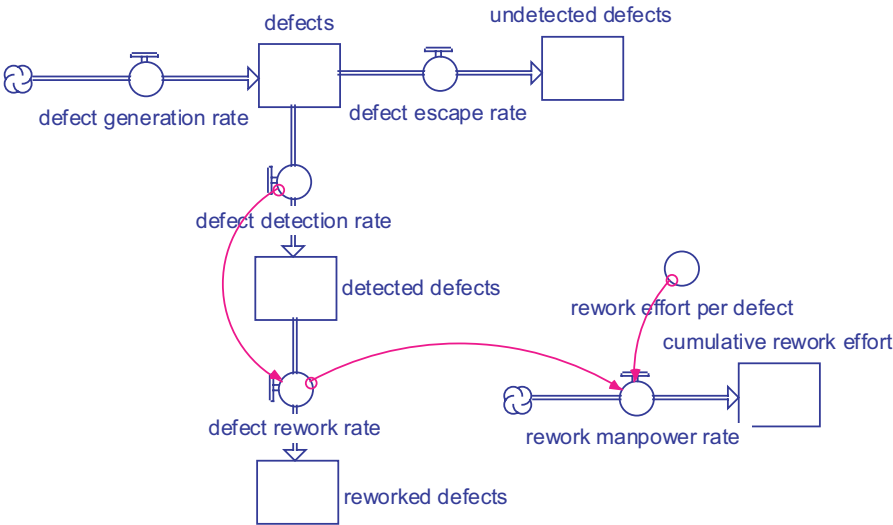


Figure 3.49. Rework all defects right away structure. Equation:
$$\text{rework manpower rate} = \text{defect rework rate} \cdot \text{rework effort per defect}$$

it policy to rework all errors as soon as they are detected. The resources are assumed to be available to start the work immediately with no constraints. The rework rate drains the detected defect level based on an average fixing delay. Not shown is a detection efficiency parameter to proportionally split the defect flow into escaped and detected errors. The effort expended on rework is also tracked with this infrastructure. An alternative would be to model the rate by dividing the level of detected defects by an average rework (delay) time.

Figure 3.49 is a simplistic model that does not always represent actual practice. More often, there are delays and prioritization of defects, and/or schedule tradeoffs. An alternative structure is to have the rework rate constrained by the personnel resources allocated to it, as shown in Figure 3.50. More defect examples are shown in Chapter 5.

There are usually staffing constraints, even when the policy is to rework all defects immediately. In this case the resource constraints have to be modeled. The structure in Figure 3.49 has a level for the available personnel.

3.5 SOFTWARE PROCESS CHAIN INFRASTRUCTURES

This section provides flow chain infrastructures related to software processes consisting mostly of cascaded levels for software tasks, defects, people, and so on. These in-

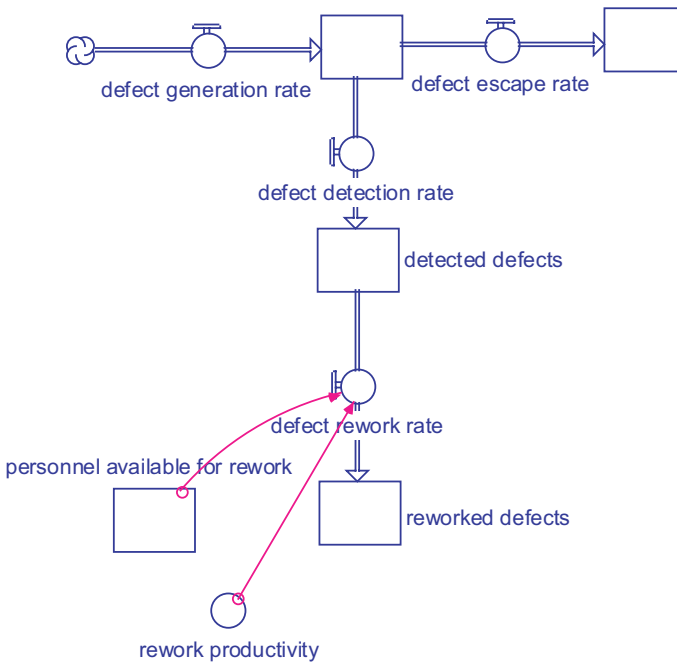


Figure 3.50. Personnel-constrained rework structure. Equation:

$$\text{defect rework rate} = \text{personnel available for rework} \cdot \text{rework productivity}$$

frastructures can be used as pieces in a comprehensive software process model, or could serve as standalone base structures for isolated experimentation. The modeling challenge lies in balancing simplicity with the complexity imposed by integrating infrastructures into a meaningful whole.

Basic flows pervasive in software processes are discussed in the following sections. When applying system dynamics, the question must be asked, What is flowing? Determination of what kinds of entities flow through a software process is of primary importance, and this section will provide applied examples. For simplicity and conciseness, the flow chains in this section do not show all the information links that can represent feedback or other variable connections.

The applied examples include actual implementations of main process chains from some major models. The chains shown are isolated from their encompassing models to ease their conceptual understanding, and are idealized in the sense that levels with corresponding inflow and outflow rates are shown without information links. (The variable names shown in these examples are largely left intact from the original authors with some very slight changes to enhance understanding.) The connections would likely distract focus away from the main chains. Comparison of the chains for products, personnel, and defects illustrates specific model focuses and varying aggregation levels. The structures of the different models reflect their respective goals.

A number of executable models are also identified and provided for use. They may be experimented with as is or be modified for particular purposes. Before running a new or revised model, all input and output flow rates must be specified and initial values of the levels are also needed.

3.5.1 Software Products

Software development is a transformational process. The conserved chain in Figure 3.51 shows a typical sequential transformation pass from software requirements to design to code to tested software. Each level represents an accumulation of software artifacts. The chain could be broken up for more detail or aggregated to be less granular.

The tasks are abstracted to be atomic units of work uniform in size. Note that the unit of tasks stays the same throughout the chain, and this simplification is reasonable for many modeling purposes. In reality, the task artifacts take on different forms (e.g., textual requirements specifications such as use case descriptions to UML design notation to software code). Operationally, a task simply represents an average size module in many models.

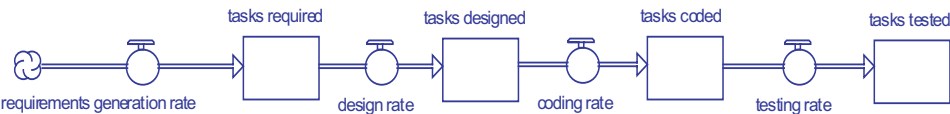


Figure 3.51. Product transformation chain.

3.5.1.1 Conserved Versus Nonconserved Product Flow

Software artifact sequences can be modeled as conserved flows, where each level has the same unit, or in nonconserved flow chains where product transformation steps are modeled using distinct artifact types. Each level in the process has different units in nonconserved chains. One of the elegant aspects of system dynamics is the simplification of using conserved flows. Hence, many models employ a generic “software task” per the examples shown above.

However, the process and modeling goals may dictate that sequential artifacts be modeled in their respective units. For example an agile process may be modeled with the nonconserved and simplified product transformation as in Figure 3.52. This structure only represents a single cycle or iteration. Additional structure, like that of a cyclic loop, is needed to handle multiple passes through the chain. Other variations might have use cases instead of user stories or classes implemented instead of software code; the acceptance tests could have been converted from user stories instead of code, and so on.

The conversion between artifact types may be modeled as an average transformation constant. For example, the parameter *user story to design task conversion* could be set to three design tasks per user story. A variable function could also be used in the conversion. A discrete model may even use individual values for each software artifact, whereby each user story entity is associated with a specific number of design tasks.

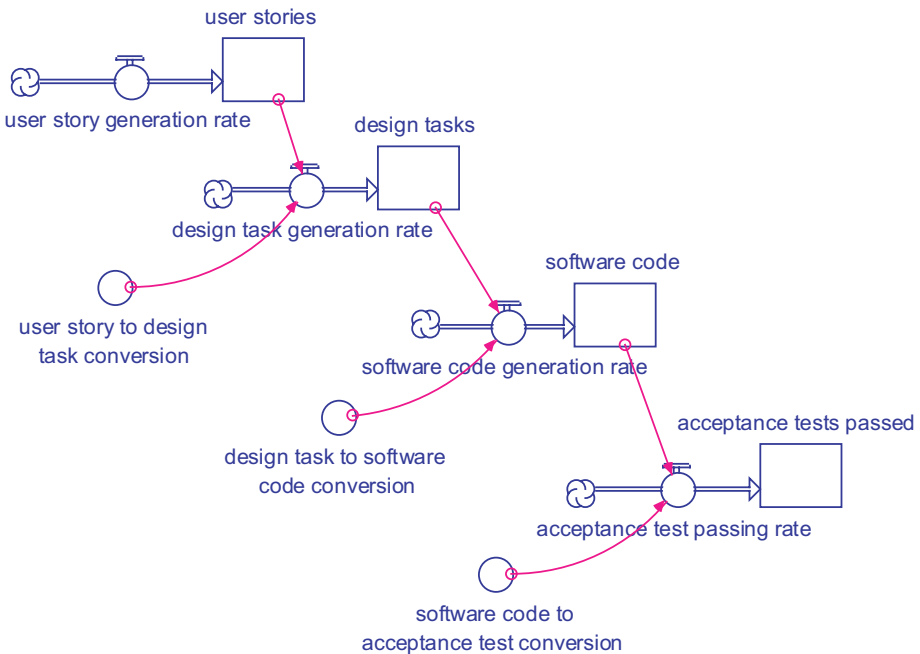


Figure 3.52. Nonconserved product flow.

3.5.1.2 Example Software Product Chains

The following three examples of software product chains illustrate successive “boring in” to the process. The different levels of detail correspond to the goals of the particular models, and are reflected in the different number of levels modeled. First is a highly aggregated chain of three levels representing software tasks in a general project model. Second is a chain of six levels to account for individual project phases and inspection points. Finally, a chain of eight intraphase levels is used to investigate even further down into the inspection process itself.

Abdel-Hamid’s integrated project model [Abdel-Hamid, Madnick 1991] represents software tasks at a top-level, highly aggregated view. Figure 3.53 shows only three levels being used. The software development rate covers both design and coding. More information on the surrounding information connections and the rest of the Abdel-Hamid model are provided in Chapters 4–6 with associated models. The skeleton product chain in Figure 3.53 is elaborated in Chapter 5.

The product chain from [Madachy 1994b] in Figure 3.54 includes more detailed phases for the purpose of evaluating inspections throughout the lifecycle. The more aggregated view in the Abdel-Hamid model (Figure 3.53) combining design and coding and with no requirements stage, would not provide requisite visibility into how inspections effect process performance in each phase. The inspection model from [Madachy 1994b] is detailed in Chapter 5.

The detailed chain from [Tvedt 1995] in Figure 3.55 includes the subphases of the inspection process. Each of the primary phases in Figure 3.54 could be broken down to this level of detail. The purpose of this disaggregation is to explore the steps of the inspection process itself in order to optimize inspections.

The comparison of the Madachy and Tvedt chains for inspection modeling reflects the different modeling goals. Per the GQM example in Chapter 2, the different goals result in different levels of inspection process detail in the Madachy model versus the Tvedt model. The top-level view of the Madachy model to investigate the overall project effects of incorporating inspections results in more aggregation than Tvedt’s detailed focus on optimizing the inspection process. Tvedt’s model is another layer down within the inspection process, i.e. a reduced level of abstraction compared to the Madachy model. There is finer detail of levels within the product chain.

Inspection parameters in the Madachy model were used and calibrated at an aggregate level. For example, an auxiliary variable, *inspection efficiency* (the percentage of defects found during an inspection), represents a net effect handled by several components in the detailed Tvedt inspection model. The Madachy model assumes a given

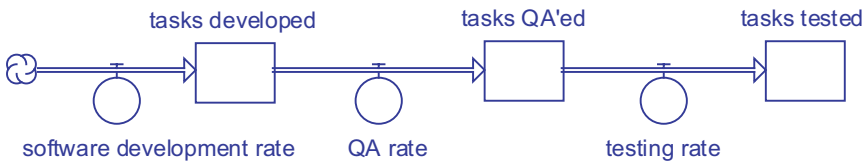


Figure 3.53. Product chain from [Abdel-Hamid, Madnick 1991].

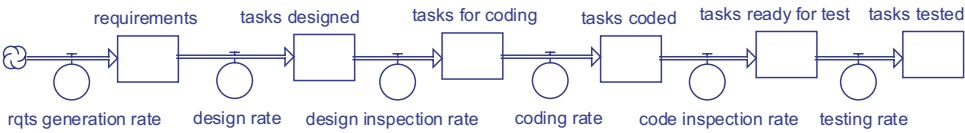


Figure 3.54. Product chain from [Madachy 1994b].

value for each run, whereas the Tvedt model produces it as a major output. A possible integration of the two models would involve augmentation of the Madachy model by inserting the Tvedt model subsystem where *inspection efficiency* is calculated.

3.5.2 Defects

This section shows ways to represent defects, including their generation, propagation, detection, and rework. Defects are the primary focus but are inextricably tied to other process aspects such as task production, quality practices, process policies to constrain effort expenditure, various product and value attributes, and so on.

3.5.2.1 Defect Generation

Below are several ways to model the generation of defects. They can be tied directly to software development or modeled independently.

3.5.2.1.1 DEFECT COFLOWS. An obvious example of coflows in the software process is related to one of our common traits per the aphorism “to err is human.” People make errors while performing their work. Defects can originate in any software phase. They are simultaneously introduced as software is conceived, designed, coded,

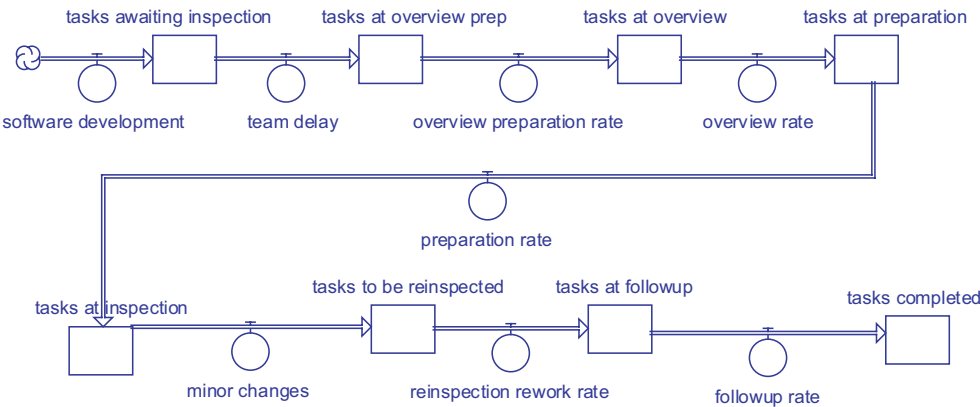


Figure 3.55. Product chain from [Tvedt 1995].

integrated etc.* Defects can also be created during testing or other assessment activities, such as a test case defect that leads to erroneous testing results.

A convenient way of expressing the defect generation rate multiplies the defect density, or the number of defects normalized by size (defects/SLOC, defects/function point, etc.), by the production rate, such as in Figure 3.56. With this, the defect generation rate is associated with the software development rate. The defect density used in this example could be constant or variable. This structure can be applied to any process subactivity; for example, design defect generation tied to a design rate or code defect generation as a function of the coding rate. Modeling the dynamic generation or detection of defects over time provides far greater visibility than static approaches, by which defect levels are described as the final resulting defect density.

3.5.2.1.2 RAYLEIGH DEFECT GENERATION. The generation of defects can also be modeled independently of task production. The Rayleigh curve is one traditional method of doing so. See Section 3.4.10 for a Rayleigh curve generator that can be modified for defects (and the corresponding chapter exercise).

3.5.2.2 Defect Detection (Filters)

It is instructive to consider the flow of defects throughout life-cycle phases with each potential method of detection as a filter mechanism. Reviews and testing are filters used to find defects. Some defects are stopped and some get through. The split-flow generic process is a perfect representation of filtering and easy to understand.

A natural way to model the detection of defects is to introduce *defect detection efficiency*, which is a dimensionless parameter that quantifies the fraction of total defects found. Figure 3.57 shows the defect chain split into detected and undetected subchains. Conservation of flow dictates that the overall incoming flow must be preserved among the two outflows from the initial defect level.

Process defect detection efficiencies (also called *yields*) should be readily available in organizations that have good software defect and review metrics across the life cycle. Typically, these may vary from about 50% to 90%. See Chapter 5 for additional references and nominal detection efficiencies, which can be used if no other relevant data is available.

3.5.2.3 Defect Detection and Rework

The high-level infrastructure in Figure 3.58 adds the rework step after detection. This infrastructure represents a single phase or activity in a larger process and could be part of an overall defect flow chain. The defect generation rate could be expressed as a coflow with software artifact development. The split in the chain again represents that some defects are found and fixed at the detection juncture, while the others are passed on and linger in the undetected level. Those that are detected are then reworked.

*The cleanroom process philosophy (i.e., “you can be defect free”) is not incongruent with this assumption. Defects are still introduced by imperfect humans, but cleanroom methods attempt to detect all of them before system deployment.

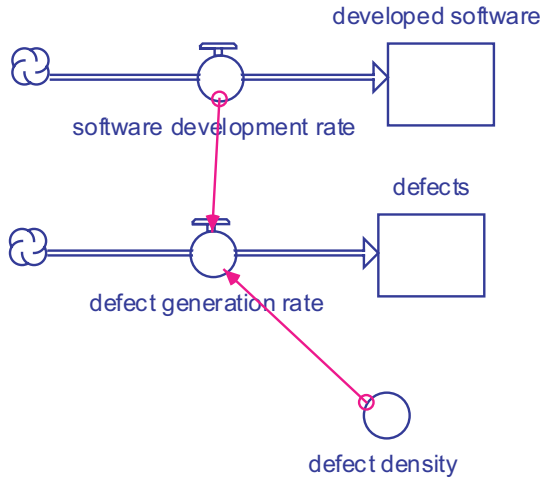


Figure 3.56. Defect generation coflow. Equation:

$$\text{defect generation rate} = \text{software development rate} \cdot \text{defect density}$$

This structure only shows defects being reworked a single time, whereas they may actually recycle through more than once. Another way to model defect rework is to use a variant of the production and rework structure in Figure 3.36, or a cyclic loop flow can be used to model the cycling through of defects. Another addition would model bad fixes, whereby new defects would be generated when old defects were being fixed. Figure 3.58 only shows the detection and rework for part of a defect chain. The defects may also be passed on or amplified as described next.

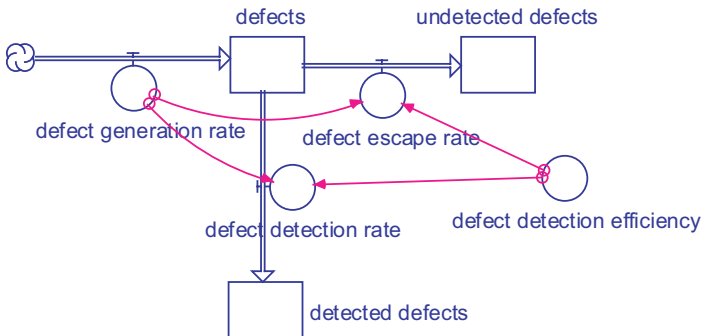


Figure 3.57. Defect detection structure. Equations:

$$\text{defect escape rate} = (1 - \text{defect detection efficiency}) \cdot \text{defect generation rate}$$

$$\text{defect detection rate} = \text{defect detection efficiency} \cdot \text{defect generation rate}$$

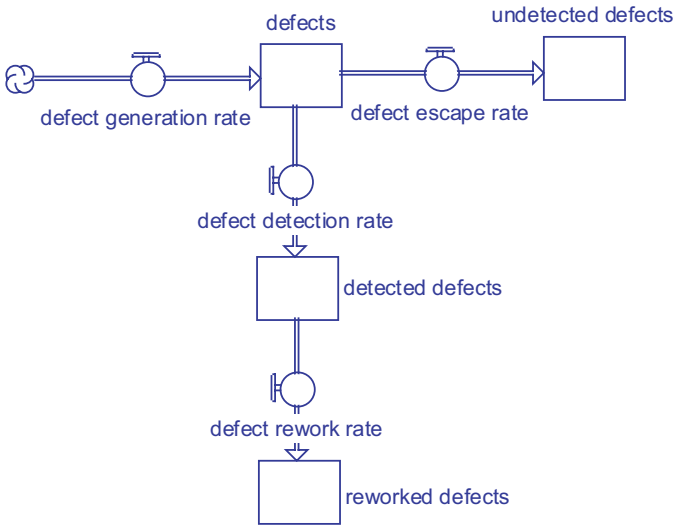


Figure 3.58. Defect detection and rework chain.

3.5.2.4 Defect Amplification

Defects may be passed between phases (corresponding to sequential activities) or possibly amplified, depending on the nature of elaboration. The structure in Figure 3.59 shows an example defect amplification (multiplication) structure from the Madachy inspection model that uses an amplification factor. Abdel-Hamid’s project model used a different approach for error amplification described later in Chapter 5.

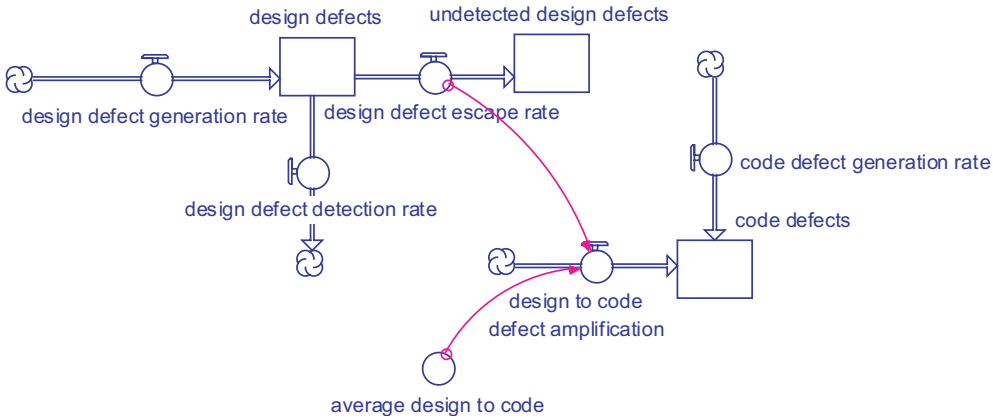


Figure 3.59. Defect amplification structure.

3.5.2.5 Defect Categories

There are many defect taxonomies. Defects can be modeled, assessed, and prevented using different classification schemes and attributes. Such classifications can be critical for defect analysis and prevention. One can model different severities such as low, medium, and high. Defect types associated with their origin could include requirements, design, code, or test defects. A way to model different defect categories is to have separate flow chains for the categories. One flow chain could be for minor defects and another for major defects, for example. The next section will show structures to separately track discovered, undiscovered, and reworked defects.

Figure 3.60 shows an example structure to track different requirements defect types per the Orthogonal Defect Classification (ODC) scheme [Chillarege et al., 2002]. For simplicity, it only shows their generation aspect and not their detection and removal. It could also be enhanced to include further attributes such as defect finding triggers, defect removal activities, impact, target, source, and so on. However, this scheme can get difficult with system dynamics, and discrete modeling might be more suitable (see the Chapter 5 discussion on product attributes and Chapter 7 about combining continuous versus discrete approaches). Also see Chapter 5 for a defect classification scheme and examples of modeling applications for detailed product attributes.

This simplistic scheme decomposes generic requirements defects into subcategories according to their respective fractions. A more detailed model would use unique factors to model the introduction and detection of each type of defect. The timing and shape of the defect curves is another consideration. See the example model in Chapter 6 on defect removal techniques and ODC that uses modified Rayleigh curves.

3.5.2.6 Example Defect Chains

Example defect chains from major models are shown here without connections to external parameters used in the rate formulas. Figure 3.61 shows two separate, unlinked defect propagation chains from the Abdel-Hamid project model. Active errors are those that can multiply into more errors.

Figure 3.62 is from the Madachy inspection model. Note the finer granularity of development phases compared to the Abdel-Hamid project model. The chain splits off for detected versus nondetected error flows. This structure also accounts for amplification of errors from design to code (see also Section 3.5.2.4, Defect Amplification). An information link is shown from design error escape rate to denote this, but not shown is a parameter for the amount of amplification. The code error generation rate refers to newly generated code errors, as opposed to design errors that escape and turn into code errors. They collectively contribute to the overall level of code errors. Without amplification, the chains would be directly connected as conserved flows.

3.5.3 People

Structures for representing personnel chains are provided in this section. These conserved-flow chains traditionally account for successive levels of experience or skill as people transition over time. Levels representing the number of personnel at each stage are mainstays of models that account for human labor. In simplistic models in which

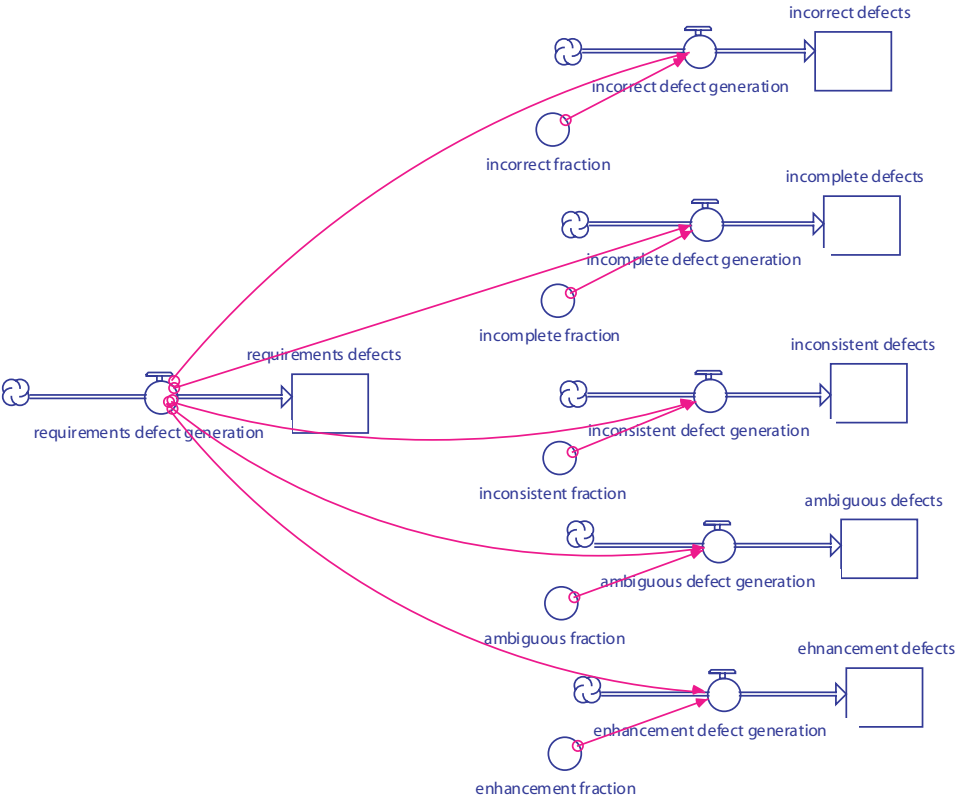


Figure 3.60. Example streams for ODC requirements defects. Equations:

$$\text{incorrect defect generation} = \text{incorrect fraction} \cdot \text{requirements defect generation}$$
$$\text{incomplete defect generation} = \text{incomplete fraction} \cdot \text{requirements defect generation}$$
$$\text{inconsistent defect generation} = \text{inconsistent fraction} \cdot \text{requirements defect generation}$$
$$\text{ambiguous defect generation} = \text{ambiguous fraction} \cdot \text{requirements defect generation}$$
$$\text{enhancement defect generation} = \text{enhancement fraction} \cdot \text{requirements defect generation}$$

dynamic staff levels are not of concern, one can get by with auxiliary variables to represent the staff sizes. More complex analysis of personnel attributes corresponding to different skillsets, experience, performance, other personnel differentiators, and non-monotonic trends requires more detail than auxiliaries or single levels can provide.

3.5.3.1 Personnel Pools

Each level in a personnel chain represents a pool of people. Figure 3.63 shows a three-level model of experience. The sources and sinks denote that the pools for entry-level recruiting and cumulative attrition are not of concern; they are outside the effective organizational boundary for analysis. Varying degrees of detail and enhancements to the

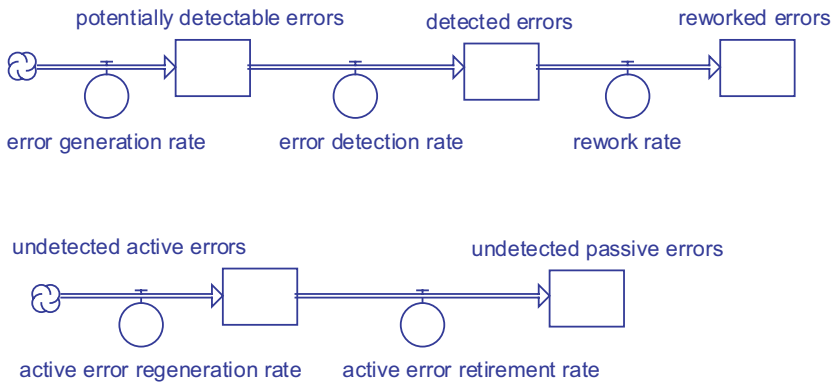


Figure 3.61. Defect chains from [Abdel-Hamid, Madnick 1991].

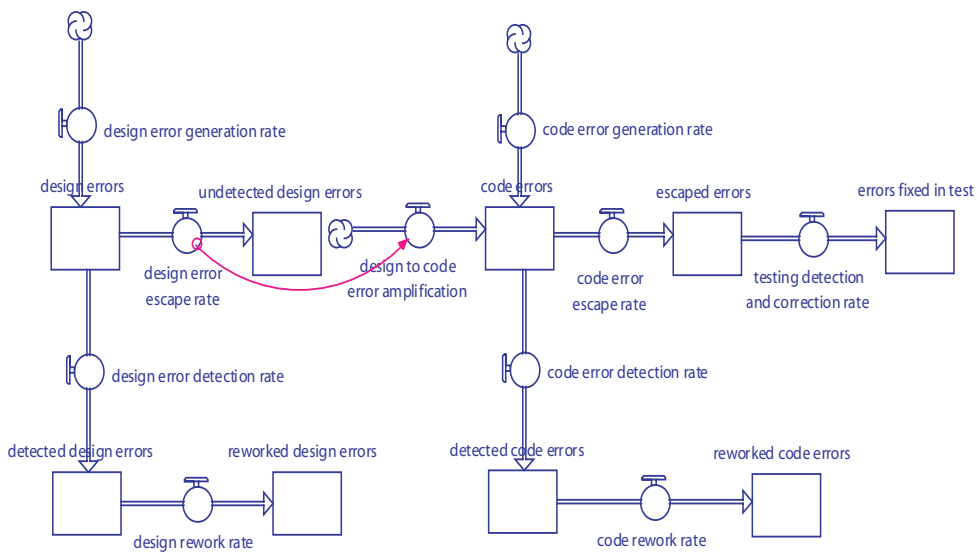


Figure 3.62. Defect chains from [Madachy 1994b].

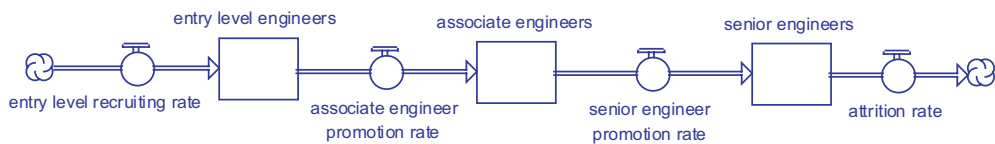


Figure 3.63. Personnel chain (three levels).

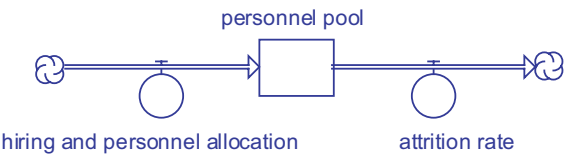


Figure 3.64. Personnel pool from [Madachy 1994b].

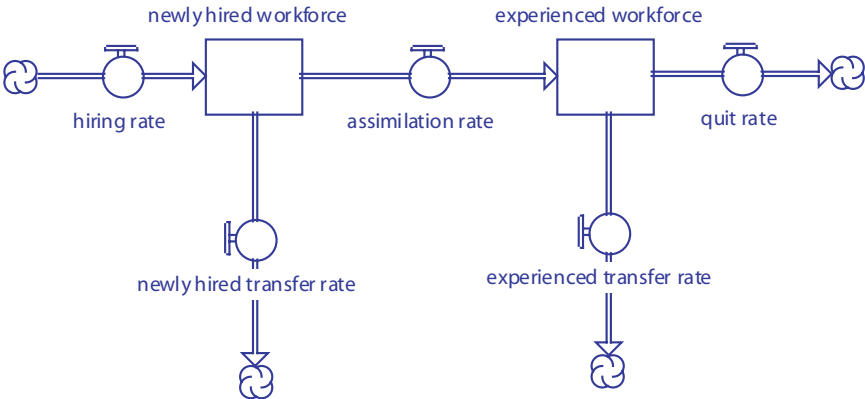


Figure 3.65. Personnel pool from [Abdel-Hamid, Madnick 1991].

personnel pool chain are possible, such as adding chain splits for attrition to occur from any experience level.

3.5.3.2 Example Personnel Chains

The single-level representation in Figure 3.64 is from [Madachy 1994b]. Since the effects of personnel mix were not germane to the study, a single aggregation was sufficient. It can be augmented to handle multiple experience levels, but doing so adds additional overhead for data calibration and model validation.

The two-level chain in Figure 3.65 is from the Abdel-Hamid model. Two levels are used for some considerations such as overhead for rookies and differing error injection rates. This chain has been used as a traditional system dynamics example for many years outside of software engineering.

MAJOR REFERENCES

[Forrester 1968] Forrester J. W., *Principles of Systems*. Cambridge, MA: MIT Press, 1968.
[Hines 2000] Hines J., *Molecules of Structure Version 1.4*, LeapTec and Ventana Systems, Inc., 2000.

[Richmond et al. 1990] Richmond B., and others, *Ithink User's Guide and Technical Documentation*, High Performance Systems Inc., Hanover, NH, 1990.

CHAPTER 3 SUMMARY

Models are composed of building blocks, many of which are generic and can be reused. Model elements can be combined into increasingly complex structures that can be incorporated into specific applications. Generic flow processes, flow chains, and larger infrastructures are examples of recurring structures. Archetypes refer to generic structures that provide “lessons learned” with their characteristic behaviors (and will be addressed in subsequent chapters). There are few structures that have not already been considered for system dynamics models, and modelers can save time by leveraging existing and well-known patterns.

The hierarchy of model structures and software process examples can be likened, respectively, to classes and instantiated objects. Characteristic behaviors are encapsulated in the objects since their structures cause the behaviors. The simple rate and level system can be considered the superclass from which all other structures are derived. Additional rates and levels can be added to form flow chains. Generic flow processes add additional structural detail to model compounding, draining, production, adjustment, coflows, splits, and cyclic loops. Some of their typical behaviors were first described in Chapter 2, such as growth, decline, and goal-seeking behavior. The structures all have multiple instantiations or applications for software processes.

The generic structures can be combined in different ways and detail added to create larger infrastructures and complex models. The production infrastructure is a classic example that combines the generic flow for production with a personnel chain. Other structures and applied examples were shown for growth and S-curves, information smoothing, delays, balancing feedback, oscillation, smoothing, learning, staffing profiles, and others. The structures for attribute averaging, attribute tracking, and effort instrumentation produce no specific behavior and are used for calculations.

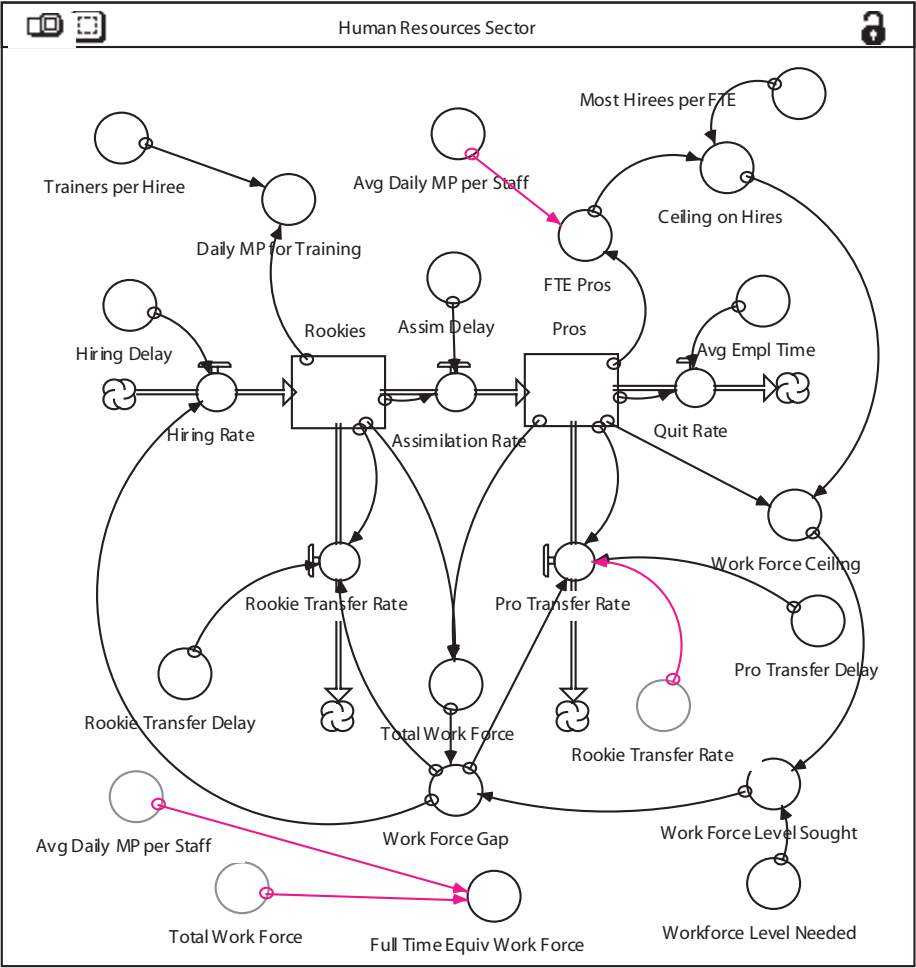
Some decision structures relevant to software projects were shown, including policies to allocate staff, adjust project goals/estimates as a project progresses, and defect rework policies. All of these structures can be reinterpreted to simulate policies for different decision contexts.

Finally, main chain infrastructures for software processes were illustrated. Some common infrastructures include product transformation, defect generation, defect detection and rework, personnel flow chains, and more. Applied examples of these chains from past modeling applications were highlighted to illustrate the concepts. The level of aggregation used in the chains depend on the modeling goals and desired level of process visibility.

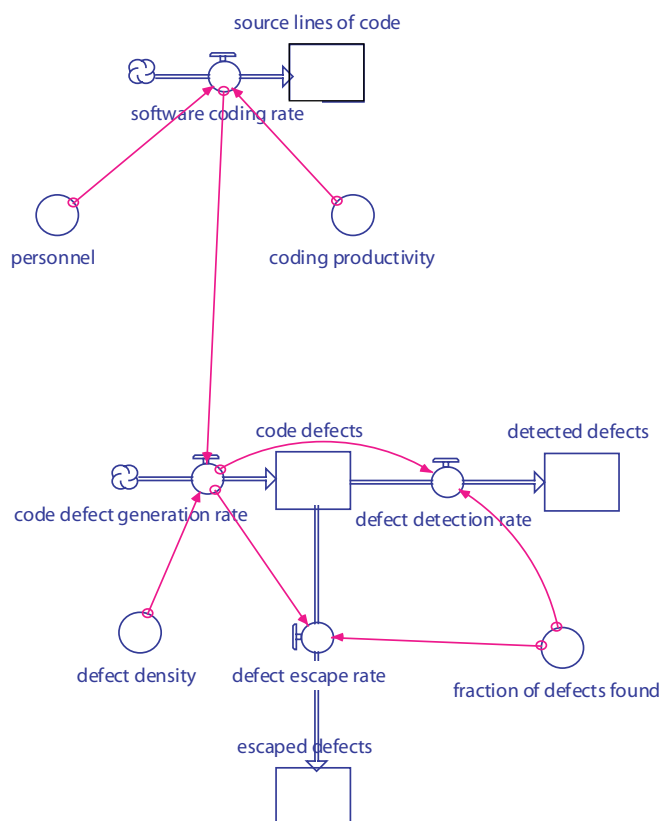
3.8 EXERCISES

Exercises 3.1–3.8 should be done without the use of a computer.

3.1. Below is the personnel structure from the Abdel-Hamid project model. Identify and trace out an example of a feedback loop in it. Each involved entity and connection must be identified.



3.2. Identify the units of measurement for each entity in the system below and sketch the graphic results of a simulation run for the named variables.



- $$\text{code_defects}(t) = \text{code_defects}(t - dt) + (\text{code_defect_generation_rate} - \text{defect_detection_rate} - \text{defect_escape_rate}) * dt$$

INIT code_defects = 0

INFLOWS:

 - $$\text{code_defect_generation_rate} = \text{software_coding_rate} * \text{defect_density}$$

OUTFLOWS:

 - $$\text{defect_detection_rate} = \text{code_defect_generation_rate} * \text{fraction_of_defects_found}$$
 - $$\text{defect_escape_rate} = \text{code_defect_generation_rate} * (1 - \text{fraction_of_defects_found})$$
 - $$\text{detected_defects}(t) = \text{detected_defects}(t - dt) + (\text{defect_detection_rate}) * dt$$

INIT detected_defects = 0

INFLOWS:

 - $$\text{defect_detection_rate} = \text{code_defect_generation_rate} * \text{fraction_of_defects_found}$$
 - $$\text{escaped_defects}(t) = \text{escaped_defects}(t - dt) + (\text{defect_escape_rate}) * dt$$

INIT escaped_defects = 0

INFLOWS:

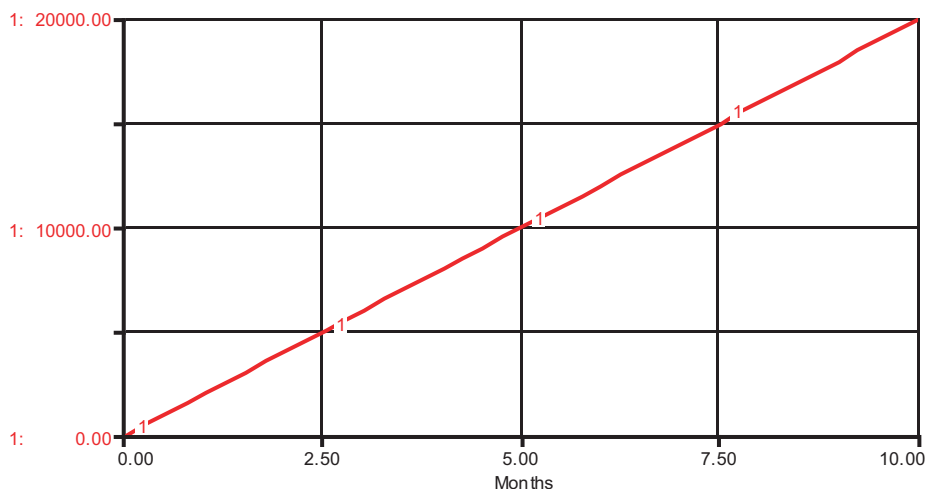
 - $$\text{defect_escape_rate} = \text{code_defect_generation_rate} * (1 - \text{fraction_of_defects_found})$$
 - $$\text{source_lines_of_code}(t) = \text{source_lines_of_code}(t - dt) + (\text{software_coding_rate}) * dt$$

INIT source_lines_of_code = 0

INFLOWS:

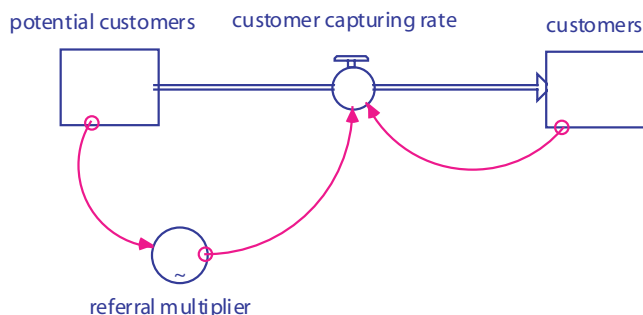
 - $$\text{software_coding_rate} = \text{coding_productivity} * \text{personnel}$$
- coding_productivity = 200
 - defect_density = .05
 - fraction_of_defects_found = .5
 - personnel = 10

1: source lines of code

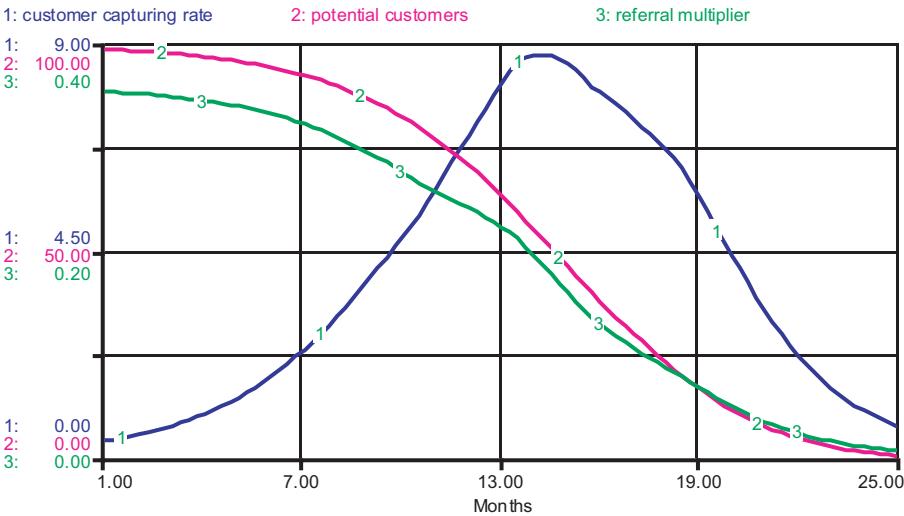
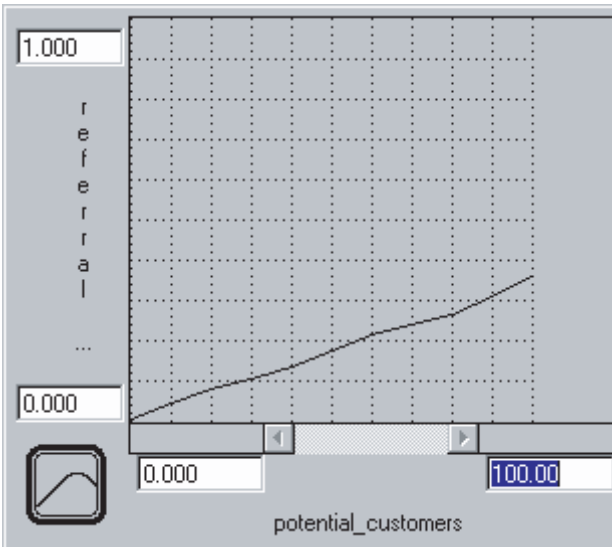


Sketch the three graphs of code defects, defect detection rate, and detected defects.

- 3.3. Below is a model of customer trends, in which the simulation time is measured in months. Identify the units of measurement for each entity in the model and sketch the trend for customers. If an entity has no units, then write “dimensionless.” Be as accurate as possible drawing the shape of the customer graph, and be sure to assign numbers to the y-axis for it.



- ☐ $\text{customers}(t) = \text{customers}(t - dt) + (\text{customer_capturing_rate}) * dt$
INIT customers = 1
INFLOWS:
- ☐ $\text{customer_capturing_rate} = \text{customers} * \text{referral_multiplier}$
- ☐ $\text{potential_customers}(t) = \text{potential_customers}(t - dt) + (- \text{customer_capturing_rate}) * dt$
INIT potential_customers = 99
OUTFLOWS:
- ☐ $\text{customer_capturing_rate} = \text{customers} * \text{referral_multiplier}$
- ☒ $\text{referral_multiplier} = \text{GRAPH}(\text{potential_customers})$
(0.00, 0.005), (10.0, 0.045), (20.0, 0.08), (30.0, 0.105), (40.0, 0.135), (50.0, 0.175), (60.0, 0.215), (70.0, 0.24), (80.0, 0.265), (90.0, 0.31), (100, 0.36)

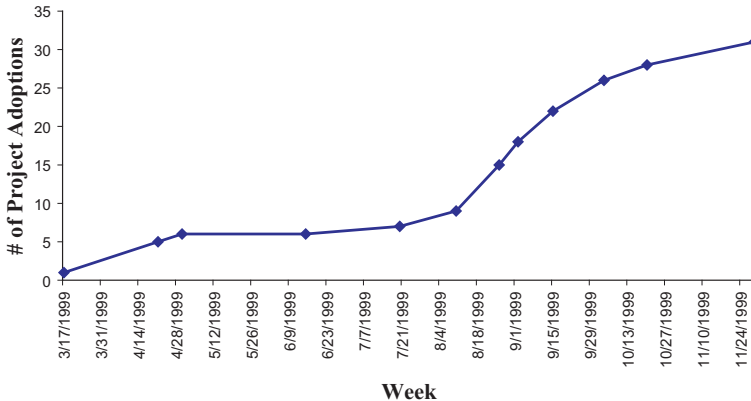


Sketch customers on this graph.

- 3.4. Create a comprehensive list of software process measures that the attribute tracking and attribute averaging structures can be used to calculate. What types of decisions and other submodels could those measures be used in?
- 3.5. What other important cumulative quantities in the software process can be instrumented like effort, and for what purposes? These measures are process indicators but do not necessarily add to the dynamics. They must provide visibility into some aspect of software process performance.

Advanced Exercises

- 3.6. Create some new infrastructures by combining generic flow processes. What do they model and how could they be used?
- 3.7. a. Parameterize the Rayleigh curve model for different staffing shapes. Enable the user to specify the shape through its ramp-up characteristics.
b. Modify and parameterize the Rayleigh curve to be consistent with the appropriate schedule span for a project. It should retain the same effort/schedule ratio as an accepted cost model.
- 3.8. Add the following extensions to the squeaky wheel resource allocation structure:
 - a. Enhance the structure to include additional tasks.
 - b. Vary the productivities.
 - c. Put additional weighting factors into the allocation such as effort multipliers for unique project factors.
 - d. Add a feedback loop so that the resource acquisition rate changes in response to the perceived need for additional resources.
- 3.9. Modify the scheduled completion date structure for effort and quality goals.
- 3.10. Below is a technology adoption curve observed at Litton Systems for a new Web-based problem reporting system.



Create a simple S-curve technology adoption model, and calibrate it as best as possible to the time dynamics of this figure.

- 3.11. Create a model of software entropy that exhibits code decaying over time as the result of ongoing changes to a system. The average cost to fix a defect or implement a new feature should increase over time. The time horizon for this model should be lengthy, on the order of years. Evaluate some of the code decay measures from [Eick et al. 2001] and experiment by testing some of their suggestions (or your own) to reduce code decay effects. Validate the model against actual change data.
- 3.12. Modify a learning curve model so that the learning is a function of completed

tasks rather than the percent of job completion. Optionally, calibrate the learning curve function to actual data, or other models of experience and learning.

- 3.13. Create a model for information smoothing of trends used for decisions and combine it with one of the decision structures. For example, suppose software change requests are received with many fluctuations over time. Smooth the input in order to determine the desired staff. Justify your smoothing interval time.
- 3.14. The example behaviors early in this chapter are shown for unperturbed systems, in order to demonstrate the “pure” response characteristics. Do the following and assess the resulting behaviors:
 - a. Modify the goal for the hiring delay example to vary over time instead of being constant. A graph function can be used to draw the goal over time. Also vary the hiring delay time.
 - b. Take an exponential growth or S-curve growth model and vary the growth factors over time.
 - c. Evaluate how the desired staff model responds to dynamic changes to the scheduled completion date and the tasks to be completed.
 - d. Choose your own structure or model and perturb it in ways that mimic the real world.
- 3.15. Combine the desired staff structure and the scheduled completion date structure. Using the scheduled completion date as the interfacing parameter, assess how the desired staff responds to changes in the completion date.
- 3.16. Enhance the combined model above for the desired staff and scheduled completion date to include hiring delays. Experiment with the resulting model and determine if it can oscillate. Explain the results.
- 3.17. Explain why two levels are necessary for system oscillation and demonstrate your answer with one or more models.
- 3.18. Modify the Rayleigh curve structure to model defect generation and removal patterns.
- 3.19. Create a flow model for different categories of defects using your own taxonomy. Optionally, combine it with a Rayleigh curve function to create the profile(s) of defect generation.
- 3.20. Research the work on global software evolution by Manny Lehman and colleagues (see Appendix B). Modify the cyclic loop structure for a simple model of software evolution.
- 3.21. Integrate the provided Madachy and Tvedt inspection models to create a model of both high-level inspection effects and the detailed inspection steps. Reparameterize the model with your own or some public data. Experiment with the detailed process to optimize the overall return from inspections.
- 3.22. If you already have a modeling research project in mind, identify potential structures that may be suitable for your application. Start modifying and integrating them when your project is well defined enough.

PROJECT AND ORGANIZATION APPLICATIONS

6.1 INTRODUCTION

Applications for projects and organizations generally revolve around management issues such as estimating, planning, tracking, controlling, setting policies on how to do things, long-term strategies, and looking after people. Projects and organizations are where decisions are made, and largely direct how people and processes work to create products. Projects and organizations develop product visions and decide how processes should be set up to achieve the goals. Business cases are analyzed for developing software. Organizations derive product (line) strategies, plan individual projects that contribute to the larger strategies, and then execute the plans.

Projects and organizations provide the context to set goals and identify constraints for people to run processes and make products. These goals and constraints may apply at different levels such as individual projects or collective portfolios of projects. Strategic and tactical policies are analyzed and set with regard to people, processes, and products to achieve the goals within given constraints.

Decision structures are embodied in project and organization modeling applications. They frequently integrate the people/process/product structures to monitor status against plans, track expenditures and business value measures, and enable dynamic decisions to be made regarding people, processes, products, and so on.

Organizations are also responsible for people concerns: having people motivated and available, with the skills to perform their jobs, and ensuring their professional growth over time. Structuring teams in terms of desired skill sets and who should work

together also comes under the purview of projects and organizations. Yet projects and organizations are comprised of people themselves, and it is those very people who collectively plan, track, decide, and so on. Thus, there is substantial overlap with people applications and, again, we conclude that focusing on people will improve organizational performance.

Allocation of people and other resources is a primary management function. Decisions must be made with respect to how many, who, when, and so on should be dedicated to particular tasks. Assessing which tasks are of higher priority or which defects should be fixed first are not easy decisions, but it will be seen that simple models can help make decisions as to where to place scarce resources.

Project and organization considerations are tightly congruent with each other, but organizations must consider initiatives or aspects outside the scope of individual projects or programs. They deal with aggregates of projects. Otherwise, projects and organizations are similar in topological structure and behavior. For example, an organization chart frequently shows major overlap of project and organization structures. They are alike but also affect each other greatly. Dynamic interactions between projects and their larger organizations occur in both directions.

The impacts of project and organizational factors in the COCOMO II model are shown in Figure 6.1. The factor for *Process Maturity* is listed because its rating stems from an organizational and/or project commitment even though it was also listed as a process factor in the last chapter.

An opportunity tree for projects and organizations is shown in Figure 6.2. Since an organization consists of projects and other entities, all project opportunities can also improve the organization at large and are shown on common branches. The management techniques on the top branch are traditionally applied on projects but are equally valid at a higher organizational level. For example, the methods can be applied to a portfolio that is a group of projects with something in common (department, product line, program, etc.).

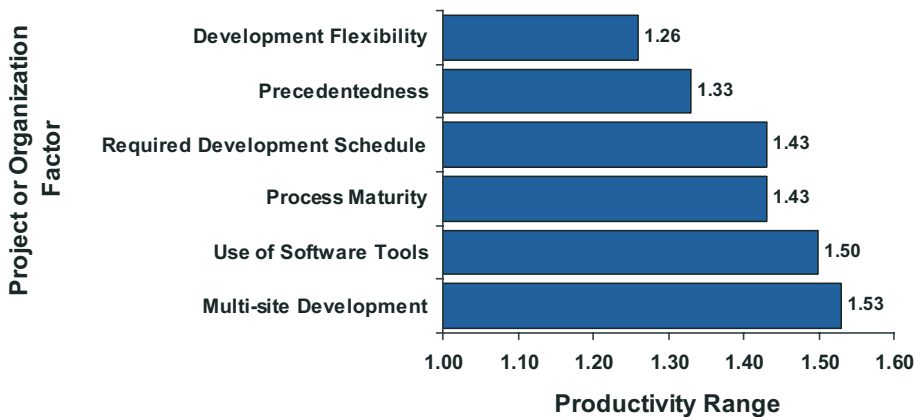


Figure 6.1. Project and organization impacts from COCOMO II—what causes these?

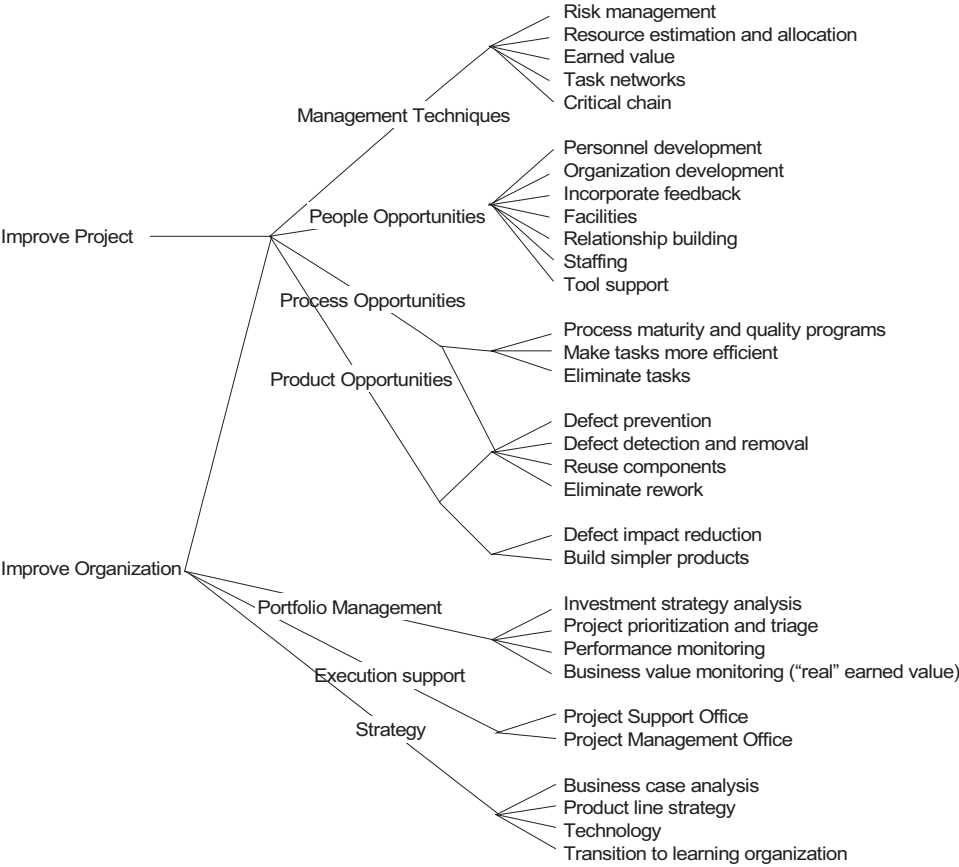


Figure 6.2. Project and organization opportunity tree.

The people, process, and product opportunities clearly impact projects and organizations. Because of the strong taxonomy overlap, the top-level branches of the respective opportunity trees in Chapters 4 and 5 are also shown here.

Simulation itself, as a relevant opportunity for organizations to improve, is perhaps best exemplified in the book, *Serious Play—How the World’s Best Companies Simulate to Innovate* [Schrage 2000]. The book you are reading complements the organizational strategies described by Schrage by providing a simulation methodology and tools for companies that deal with complex systems and software.

6.1.1 Organizational Opportunities for Feedback

Understanding and leveraging system feedback is a primary theme of this book. Many forms of feedback may take place or be made available in order to improve projects and organizations. Table 6.1 is a list of possible feedback channels to be aware of and

utilize for overall process optimization. Virtually all of them can be used to improve processes; if used improperly, they can lead to opposite results.

6.2 OVERVIEW OF APPLICATIONS

Abdel-Hamid’s model of integrated project dynamics was the first major work applying system dynamics to software development phenomena [Abdel-Hamid, Madnick 1991]. It covers many important facets but primarily focused on project management. The first section provides detail of some of its structures for the integrated dynamics. A summary critique of the model is included due to its historical significance and substantial complexity touching on many areas.

Software business case analysis is a crucial responsibility of organizations that simulation is well-suited for. Value-based software engineering is an important thrust that integrates value considerations into software practices, and can be used to analyze business propositions. A model is illustrated that integrates business value with process and product perspectives. Several applied examples are shown for evaluating product strategies, and their effect on quality and business value measures such as market share and return on investment.

Allocating people to tasks and prioritizing their work involves additional detail be-

Table 6.1. Organizational feedback examples

Within a Project	Between Projects	Between Departments or Organizations	Between Stakeholder Communities
Earned value and progress measures	Postmortem and lessons learned	Postmortem and lessons learned	User/customer requests and complaints
Process performance metrics	reports	reports	
Quality indicators and related metrics	Company newsletters and journals	Company newsletters and journals	Investor requests
Status reports	Organizational metric reports	Organizational metric reports	Shareholder meetings
Project meetings	Organizational Process	Organizational Process	Technical articles, journals, conferences,
Customer or peer reviews	Asset Library	Asset Library	workshops,
Problem reports	(normally web-based)	Multidepartmental	summit meetings
People talking and personal e-mail	Departmental meetings	meetings	
Project website or repository	People talking and personal e-mail	People talking and personal e-mail	
Project-wide bulletins and email			
Customer and other stakeholder communication			

yond overall staffing needs. Model structures are provided to simulate different allocation and prioritization policies, and the policies are compared in terms of their trade-offs.

Staffing and personnel allocation are primary responsibilities of project management. Several staffing models are reviewed that can be used for planning human resource needs over time. These include the traditional Rayleigh curve model of staffing, and process concurrence, which provides a more comprehensive way of analyzing staffing needs based on individual project characteristics.

The last section covers earned value, which is a popular and effective method to help track and control software projects. One can become a real-time process controller by using this technique to monitor dynamic trends. Earned value is explained in detail with examples because it is valuable to understand and use. It can also be misused, and some caveats are described. An original model of earned value is used that can be easily adapted. Some experiments are run with the model to evaluate project management policies. For example, it demonstrates that a policy of working hard problems first on a project is advantageous in the long run.

6.3 INTEGRATED PROJECT MODELING

6.3.1 Example: Integrated Project Dynamics Model

Abdel-Hamid's model was a complex integration of feedback structures between planning and control, software production, and human resource dynamics. Many of the structures and behaviors have already been reviewed in previous chapters on people and process applications, but this section provides more detail of its integrated aspects. This section overviews various sectors of the model, and present a few selected relationships and model diagram portions. The entire set of model diagrams and the equations can be found in the model provided. Important results from the model are discussed in several other sections of this book.

In [Abdel-Hamid 1989a] and [Abdel-Hamid 1989b], the dynamics of project staffing was investigated and validation of the system dynamics model against a historical project was performed. He synthesized the previous research into an overall framework in the book *Software Project Dynamics* [Abdel-Hamid, Madnick 1991], and this section describes his model contained in the book.

The model has four main components: human resource management, software production, controlling, and planning. The primary information flows between the model sectors that lead to dynamic behavior are shown in Figure 6.3. The structures of the different sectors are described in the following sections. The actual variable names used in the model show up in the narratives below. The provided model and much of its description in this section comes from [Johnson 1995].

The primary focus of the model is management of software production rather than software production itself. Relatively little structure is dedicated to software artifacts, though there is a detailed model of error propagation and detection. Much of the model is dedicated to planning and control, management perceptions, and personnel management. The simulation of management policies was derived from interviews, and

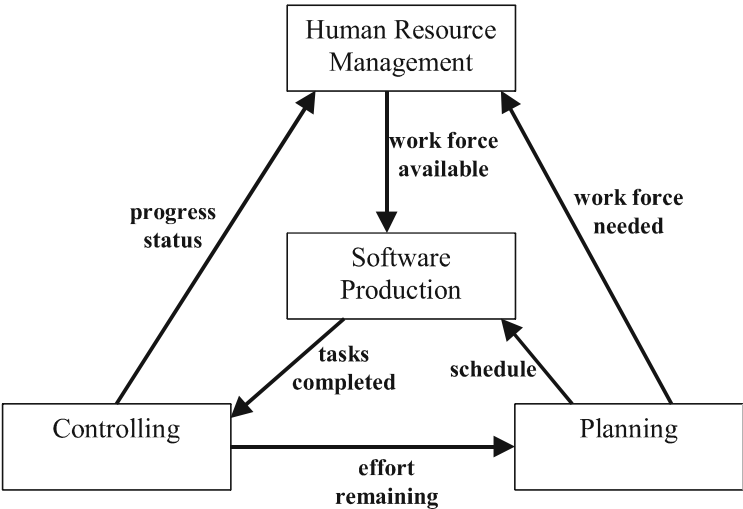


Figure 6.3. Top-level diagram of Abdel-Hamid model.

does not reflect decision-making structures in all organizations. See the last subsection on model critiques.

The software artifacts flowing through the model are defined as “tasks.” A task is an atomic unit of work that flows through the project, where the units may differ among project phases. This is a simplifying abstraction for the sake of the model. Tasks are assumed to be interchangeable and uniform in size. Abdel-Hamid set a task equivalent to 60 lines of software, or roughly the size of a standard module. The simplified chain of flowing tasks is shown in Figure 6.4. A task changes from “developed” to “QAed” (gone through quality assurance) and finally to “tested.” These are the only three levels associated with software artifacts. They represent the level of process visibility handled by the model; the product elaboration substates within development are not tracked.

Given the abstraction of flowing tasks, which is appropriate for a high-level management perspective, the model is not appropriate to use as a scheduling tool or for tracking milestones. Commensurately, the modeling of personnel with system dynamics does not lend itself to modeling individuals. All tasks and personnel are assumed to have “average” characteristics within any given level.

The Abdel-Hamid model has been mistakenly considered a deterministic “one size fits all” model that requires many inputs. Models should reflect the local organization. Obviously, one shoe does not fit all; each organization has unique aspects and associated problems with their processes. Thus, it is important to not mandate model structure for everyone; instead, allow flexibility and provide skills to adapt archetypes.

6.3.1.1 Human Resources Sector

The human resources sector contains structures for hiring, assimilation, training, and transferring of people off the project. See the Chapter 4 for details of the sector.

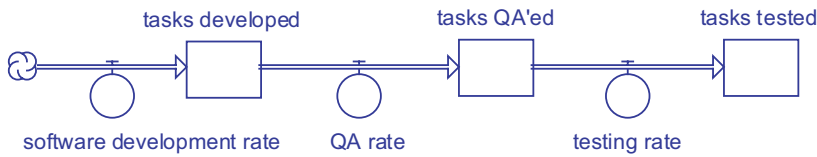


Figure 6.4. Software product chain.

6.3.1.2 Planning Sector

In the planning sector, initial project estimates are made and they are revised as necessary until the project ends. The sector contains elements for the desired number of people, a deadline date, and structures to revise and update estimates as the project progresses, depending on actual work accomplished. This sector works in conjunction with the human resources sector to determine the desired work force levels. Table 6.2 is a high-level summary of the model. The planning sector is shown in Figure 6.5.

The scheduled completion date is an important level used in this sector. It is adjusted by a “rate of adjusting flow,” which depends on the perception of progress to date.

Table 6.2. Planning sector model overview

Purpose: Planning, Process Improvement			
Scope: Development Project			
Inputs and Parameters	Levels	Major Relationships	Outputs
<ul style="list-style-type: none">• Scheduled completion date• Man-days remaining (from control sector)• Assimilation delay (from human resources sector)• Hiring delay (from human resources sector)• Total workforce level (from human resources sector)• Workforce level sought (from human resources sector)• Average manpower per staff (from human resources sector)• Schedule adjustment time• Maximum tolerable completion date	<ul style="list-style-type: none">• Scheduled completion date	<ul style="list-style-type: none">• Willingness to change workforce level	<ul style="list-style-type: none">• Workforce level needed

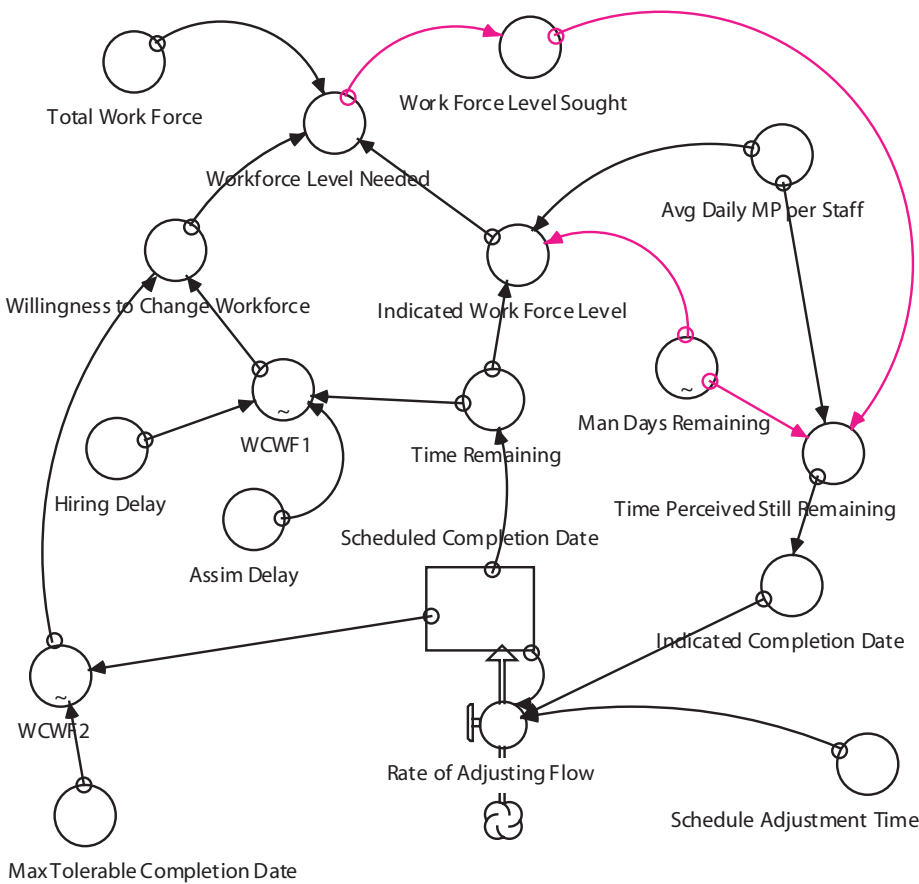


Figure 6.5. Planning sector.

This perception is captured in time perceived to be still remaining, which is added to the elapsed project time to determine the indicated completion date. The scheduled completion date is subtracted from the indicated completion date to determine a difference. The difference is divided by a delay for the schedule adjustment time to represent the change to the scheduled completion date.

The time perceived still remaining is determined with the man-days remaining (from the control sector). The man-days remaining is adjusted with the average daily manpower per staff (from the human resources sector). Workforce level sought (also from the human resources sector) is used to calculate time perceived still remaining by dividing man-days remaining by workforce level sought. Also, the indicated workforce level is determined by dividing time remaining by man-days remaining.

The workforce level needed is calculated from the indicated workforce level sought and a weighting factor for willingness to change workforce level. The weight repre-

sents the desire for a stable workforce, where a stable force does not have many new people on it. The willingness to change workforce level varies from 0 to 1. When it equals 1, the workforce level is set to the actual number perceived needed to complete the job. No weight is applied to the indicated workforce level in this case. But when the willingness is 0, the number of desired employees depends solely on stability concerns and the workforce level needed equals the total workforce. The workforce gap in the human resources sector equals zero, so no hiring or transferring is done.

The willingness to change workforce level (WCWF) has two components: WCWF1 captures the pressure for stability near the end of the project, whereas WCWF2 uses the difference between max tolerable completion date (a drop-dead time constraint) and the scheduled completion date. These are shown in Figure 6.6 and Figure 6.7. WCWF1 is affected by assimilation and hiring delays from human resources. Given these two delays, there is a reluctance to bring on new people as time remaining decreases even though there is a schedule crunch. WCWF2 rises gradually as the scheduled completion date approaches the maximum tolerable completion date. If WCWF2 exceeds WCWF1, then the weighting is dominated by schedule concerns to not go past the max tolerable completion date, and hiring takes place.

The sector thus determines the workforce level needed that is used by the human resources sector, and the two sectors interact with each other. When the total workforce increases and decreases due to hiring or transferring, the rate of change to the scheduled completion date slows down. Thus, two ways to deal with changes in the man-days remaining are to push back the scheduled completion date and to hire or transfer people.

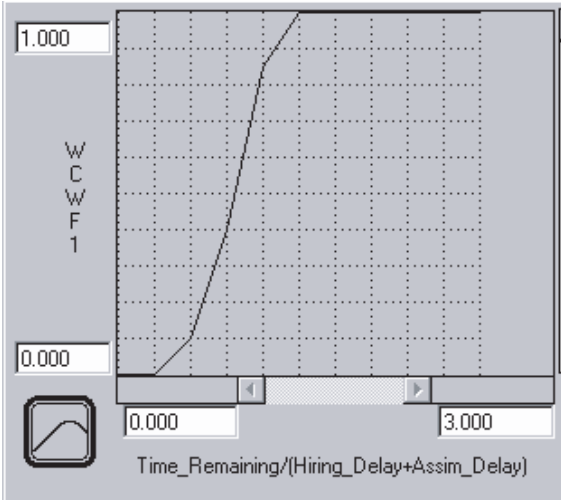


Figure 6.6. Pressure for workforce stability at end of project [willingness to change workforce versus time remaining/(hiring delay + assimilation delay)].

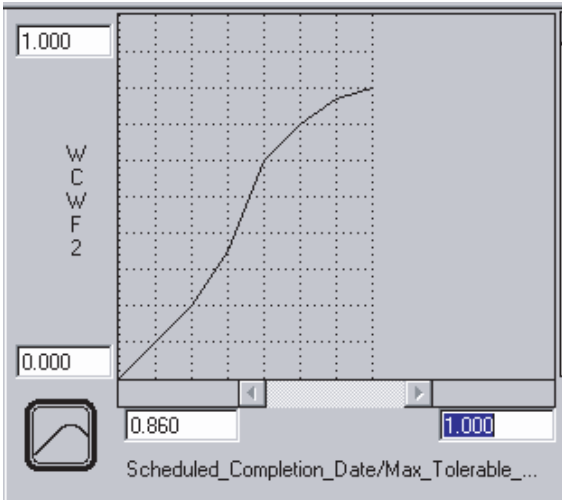


Figure 6.7. Pressure to change workforce during schedule crunch (willingness to change workforce versus scheduled completion date/maximum tolerable completion date).

6.3.1.3 Software Production Sector

The software production component has four sectors associated with it: manpower allocation, software development, quality assurance and rework, and system testing. The software development sector covers both design and coding, and they are aggregated together in a single level. Requirements analysis is not covered in the model. Developed software is reviewed, tested, and reworked in the quality assurance and rework sector, and some errors will go undetected until found in the system testing sector. Quality assurance (QA) in the model covers different techniques including walk-throughs, reviews, inspections, code reading, and integration testing. Unit testing is not included and is considered within coding. Note that this definition of QA differs from some modern practices wherein QA audits for process compliance, rather than getting directly involved with the product. Table 6.3 is a high-level summary of the model.

6.3.1.3.1 MANPOWER ALLOCATION SECTOR. The number of people to work on software production is determined in the human resources sector, and allocations are made for training, quality assurance, rework, system testing, and software development (design and coding). The total daily manpower is adjusted for training overhead first; other tasks are considered after training resources are allocated.

The allocation to QA comes from a converter influenced by a planned fraction for QA and an actual fraction for QA. The actual fraction is affected by schedule pressure from the control sector. If schedule pressure increases, then QA activities are relaxed. The default fraction of manpower for QA is 15% according to Abdel-Hamid’s research, and the fraction is adjusted accordingly based on schedule pressure.

Table 6.3. Software production sector model overview

Purpose: Planning, Process Improvement			
Scope: Development Project			
Inputs and Parameters	Levels	Major Relationships	Outputs
<ul style="list-style-type: none">• Nominal fraction of man-days for project• Nominal productivities• Exhaustion depletion delay• Maximum tolerable exhaustion• Nominal overwork duration threshold• Maximum boost in man-hours• Willingness to overwork• Rookies (from human resources sector)• Pros (from human resources sector)• Man-days remaining (from control sector)• Percent of job worked (from control sector)	<ul style="list-style-type: none">• Actual fraction of man-days for project• Exhaustion	<ul style="list-style-type: none">• Exhaustion flow• Potential productivity• Overwork duration threshold multiplier• Maximum shortage of man-days handled• Slack time growth• Communication overhead• Normal delay	<ul style="list-style-type: none">• Software development productivity

Resources are allocated to error fixing as the errors are found through QA activities. The effort is based on a desired error correction rate and the perceived rework manpower needed per error. The difference between perceived and actual rework manpower needed stems from a delay in management between realizing a need for rework effort and acting on it. So a sudden increase in actual rework manpower needed does not affect allocation decisions until the increase has persisted for a while.

6.3.1.3.2 SOFTWARE DEVELOPMENT SECTOR. This sector is the simplest one in the model, and is shown in Figure 6.8. There is a main level for cumulative tasks developed that drives almost everything else in software production. The software development rate flowing into the level is based on the number of people working and their productivity. The productivity is determined in the software development productivity subsector.

The number of people working on software development is whatever is left over after training, QA, and rework allocations in the human resources sector. The people dedicated to software development gradually transition to system testing as the project progresses (determined by the parameter fraction of effort for system testing). The fraction is dependent on the project manager’s perception, which comes from the perceived job size and tasks perceived to be remaining in the control sector.

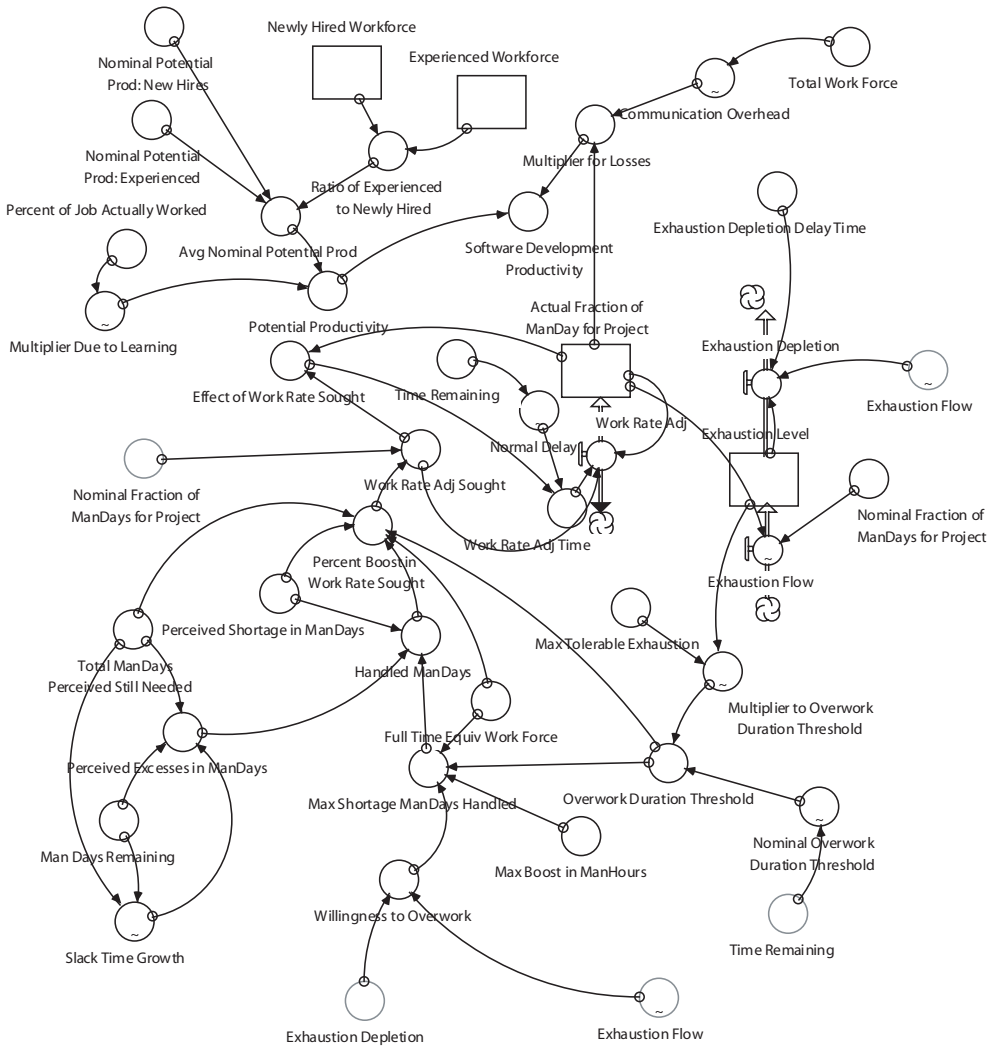


Figure 6.9. Software productivity subsector.

overwork duration threshold, full-time equivalent workforce (from the human resources sector) and the maximum boost in man-hours. The maximum shortage that can be handled varies.

Note that this productivity sector contains the exhaustion model highlighted in Chapter 4. Workers are less willing to work hard if deadline pressures persist for a long time. The overwork duration threshold increases or decreases as people become more or less exhausted. The exhaustion level also increases with overwork. See Chapter 4 for more details of the exhaustion model.

Other effects come into play if the project is perceived to be ahead of schedule. Slack time increases as people take care of personal matters, but only up to a threshold again. This time, management takes care of the threshold by adjusting the schedule.

Communication overhead is the other loss to productivity besides motivation. A graph function is used that relates the overhead to team size. The overhead is proportional to the square of team size. This is the initial formula used to model communication overhead in the Brooks's Law model in Chapter 1 (which only covers the region for 0–30 personnel).

In summary, the actual percentage of a day dedicated to project work is based on the current level of motivation after accounting for deadline pressures and exhaustion. The fraction is also adjusted for communication overhead, and the final result is the multiplier for losses.

6.3.1.3.3 QUALITY ASSURANCE AND REWORK SECTOR. The quality assurance and rework sector in Figure 6.10 models the generation, detection, and correction of errors during development. Potentially detectable errors are fed by an error generation rate. The error generation is the product of the software development rate and nominal errors committed per task. The errors committed per task is defined as a graph function against the percent of job worked. At the beginning of design, the nominal error rate is 25 errors/KDSI (KDSI stands for thousand delivered source instructions) and goes down to 12.5 errors/KDSI at the end of coding.

The workforce mix and schedule pressure also affect the error generation rates. It is assumed that new personnel generate twice as many errors as experienced people, as captured in the multiplier due to workforce mix. There is also a multiplier due to schedule pressure, defined as a graph function. As people work under more pressure, they become more tired and make more errors. The multiplier increases exponentially as schedule pressure increases.

Detection of errors is modeled with an error detection rate flowing out of potentially detectable errors and into a level for detected errors. The QA rate is modeled independent of effort and productivity. The QA function is assumed to be a prescheduled activity that processes all tasks in a fixed time. Thus, there is never a QA backlog, since all the work is done in the allocated timeframe. The QA rate is modeled as a simple delay of 10 days. The software tasks are considered QA'ed after that delay, independent of the QA effort. However, the effectiveness in terms of error detection rates is a function of how much effort is spent on QA.

A potential error detection rate is calculated as the QA effort (daily manpower for QA) divided by the QA manpower needed to detect an error. The nominal QA manpower needed per error is defined as a graph against the percent of job worked. Design errors are set to 1.6 times as costly to fix as a coding error, whereas the average is 2.4 hours to detect an error. The nominal value is also adjusted by the multiplier for losses defined in the software productivity sector. Finally, there is a multiplier due to error density used to adjust the QA manpower needed to detect an error. It is assumed that easy, obvious errors are detected first, and that subsequent errors are more expensive and difficult to find. The multiplier has no effect at large error densities, but it increases exponentially with smaller error densities. The error density (average number of er-

rors per task) is calculated by dividing the potentially detectable errors by the number of tasks worked.

The errors found in QA are routed back to programmers to be fixed as rework. The rework rate is a function of effort (actual manpower needed per error) and manpower (daily manpower for rework). The actual manpower needed per error has two components: the nominal rework manpower needed per error and the multiplier for losses from the software productivity sector. The rework manpower is a function of error type, where a design error requires more effort than a coding error. The multiplier for losses accounts for lost effort on communication and nonproject activities.

6.3.1.3.4 SYSTEM TESTING SECTOR. The system testing sector is assumed to find all errors escaped from the QA process and bad fixes resulting from faulty rework. Any remaining errors could be found in maintenance, which is not included in the model. The sector models both the growth of undetected errors as escaped errors and bad fixes generating more errors, and the detection/correction of those errors. See Table 6.4 for a summary.

There are two types of errors in the model, called passive and active, whereby active errors multiply into more errors. Each type has a level associated with it. All design errors are considered active since they could result in coding errors, erroneous documentation, test plans, and so on. Coding errors may be either active or passive. The undetected active errors level has an inflow from escaped errors and bad fixes from QA, and regeneration of active errors. The outflow comes from finding and correcting the errors, or by making them passive. There is a positive feedback loop between the undetected active errors level and the active error regeneration rate. Undetected passive errors are fed by a flow from retired active errors and the generation of passive errors from QA escaped errors and bad fixes. The outflow is the rate of detection and correction of those errors. A simplified representation of the error flow model is shown in Figure 6.11.

The active error regeneration rate is a function of the software development rate and the active error density. A delay of three months is used for the generation of new errors from active errors. The active error regeneration rate uses a graph function called multiplier to activate error regeneration due to error density as a function of active error density. As the error density increases, the multiplier increases. However, a fraction of the errors will become passive and not multiply indefinitely. The active errors retirement fraction controls the active error retirement rate, and is a graph function of the percent of job actually worked. The fraction starts at 0 in the beginning and increases rapidly until the end of coding, whereby all active errors are retired and become passive.

Another flow into undetected active errors is escaped errors and bad fixes from the QA and rework sector. The flow is the escaped errors plus bad fixes, adjusted by the fraction of escaping errors that will be active. The fraction is another graph function of the percent of the job actually worked and varies from 1 in the beginning of design to 0 at the end of coding. The actual number of bad fixes is a function of the rework rate, and is set to 0.075. The same factors affect the passive error generation rate that flows into the undetected passive errors level.

Table 6.4. System testing sector model overview

Purpose: Planning, Process Improvement			
Scope: Development Project			
Inputs and Parameters	Levels	Major Relationships	Outputs
<ul style="list-style-type: none">• Time to smooth active error density• QA rate (from QA and rework)• Daily manpower for testing (from manpower allocation sector)• Multiplier for losses (from software productivity sector)• Testing effort overhead• Testing manpower per error• Error escape rate (from QA and rework)• Rework rate (from QA and rework)• Percent bad fixes• Percent of job actually worked (from control sector)• Software development rate (from software productivity sector)	<ul style="list-style-type: none">• Errors<ul style="list-style-type: none">Undetected activeUndetected passive• Tasks<ul style="list-style-type: none">QA'edTested• Testing man-days• Errors reworked in testing	<ul style="list-style-type: none">• Active error regeneration rate• Multiplier to regeneration due to error density• Fraction of escaping errors• Active error retiring rate	<ul style="list-style-type: none">• Tasks tested• Undetected active errors• Undetected passive errors• Testing man-days

The two stocks for active and passive errors are drained by a detection and correction rate for testing. It is defined as the daily manpower for testing divided by the testing manpower needed per task. The testing manpower needed per task uses the number of errors in a task and also considers the overhead of developing test plans (set to 2 man-days/KDSI, which can also be adjusted for motivation and communication losses). The error density of active and passive errors is multiplied by the testing manpower needed per error, which is set to 1.2 man-hours per error. Some levels are also present to accumulate the testing man-days, percent of tasks tested, total number of errors, and errors reworked in testing. These accumulations are used in the control sector.

6.3.1.4 Control Sector

The control sector measures project activities, compares estimates with actuals, and communicates the measurement evaluations to other segments of the model for readjusting project parameters. Progress is monitored by measuring the expended resources and is compared to plans. Many of the driving elements of other sectors are calculated

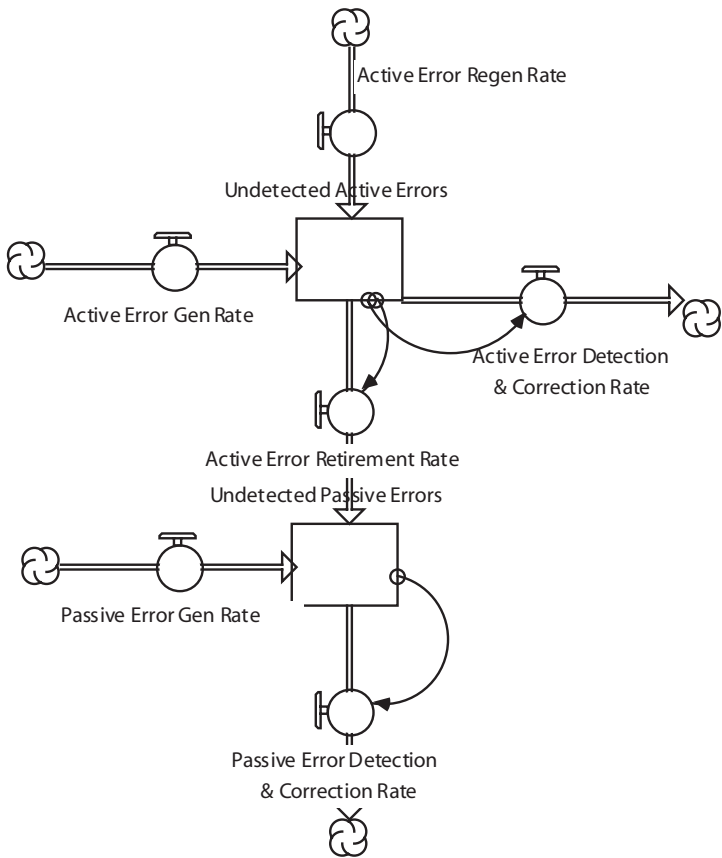


Figure 6.11. Simplified error flow.

in the control sector. See Table 6.5 for a summary. The detailed diagram for the control sector is in Figure 6.12 and Figure 6.13.

The initial value of the total job size in man-days is based on a Basic COCOMO [Boehm 1981] estimate, but it changes over the course of the project based on projected versus perceived progress, and undiscovered tasks being discovered.

One of the primary measures is total man-days perceived still needed, which represents the work for development, QA, rework, and system testing. There is an important distinction between man-days still needed and man-days remaining. Man-days still needed is based on the original plan, whereas man-days remaining is based on perception of actual progress. The two values are equal at the beginning of the project and diverge as the project continues. They essentially represent different modes of measuring progress.

Man-days perceived to be still needed for new tasks is determined by dividing tasks perceived to be remaining by the assumed development productivity. The latter

Table 6.5. Control sector model overview

Purpose: Planning, Process Improvement			
Scope: Development Project			
Inputs and Parameters	Levels	Major Relationships	Outputs
<ul style="list-style-type: none">• Tasks developed (from software development sector)• Cumulative man-days (from manpower allocation sector)• Detected errors (from QA and rework)• Perceived rework manpower needed per error (from manpower allocation sector)• Time to smooth test productivity• Delay in adjusting job size• Delay in incorporating tasks• Fraction of effort for testing (from software development sector)• Reporting delay• Tasks tested (from system testing sector)	<ul style="list-style-type: none">• Tasks<ul style="list-style-type: none">UndiscoveredDiscovered• Perceived job size• Job size• Testing size	<ul style="list-style-type: none">• Assumed development productivity• Perceived testing productivity• Multiplier for development• Multiplier for resources• Fraction of additional tasks• Undiscovered tasks discovered per day• Percent development perceived complete• Percent tasks reported complete• Schedule pressure	<ul style="list-style-type: none">• Man-days remaining• Percent of job worked• Schedule pressure

is a converter that captures the perception change from “still needed” to “remaining” during the project. Assumed development productivity is a weighted average of projected development productivity and perceived development productivity. Projected development productivity is tasks perceived to be remaining divided by man-days perceived to be remaining for new tasks. This corresponds to productivity being a function of future projections early in the project. Whereas perceived development productivity is the cumulative tasks developed divided by the cumulative development man-days, representing productivity later in the project is a function of perceived accomplishments. The weight of projected productivity then varies from 1 to 0 over the project and is a product of the rate of expenditure of resources (multiplier for resources) and the rate of task development (multiplier for development).

The man-days perceived to be needed to rework detected errors is the product of detected errors (from the QA and rework sector) and the perceived rework manpower needed per error (from the manpower allocation sector). The man-days perceived

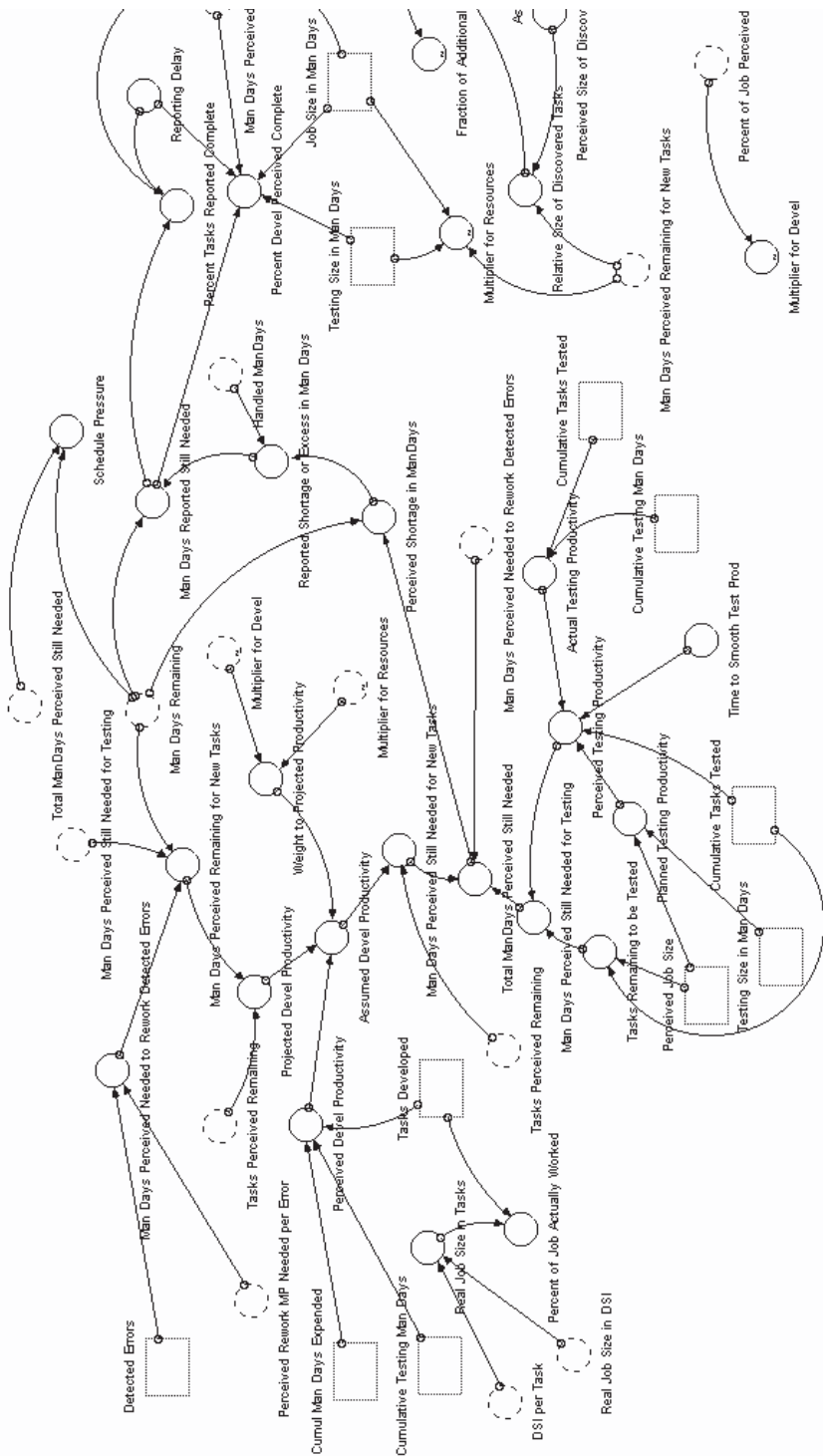


Figure 6.12. Control sector (left side).

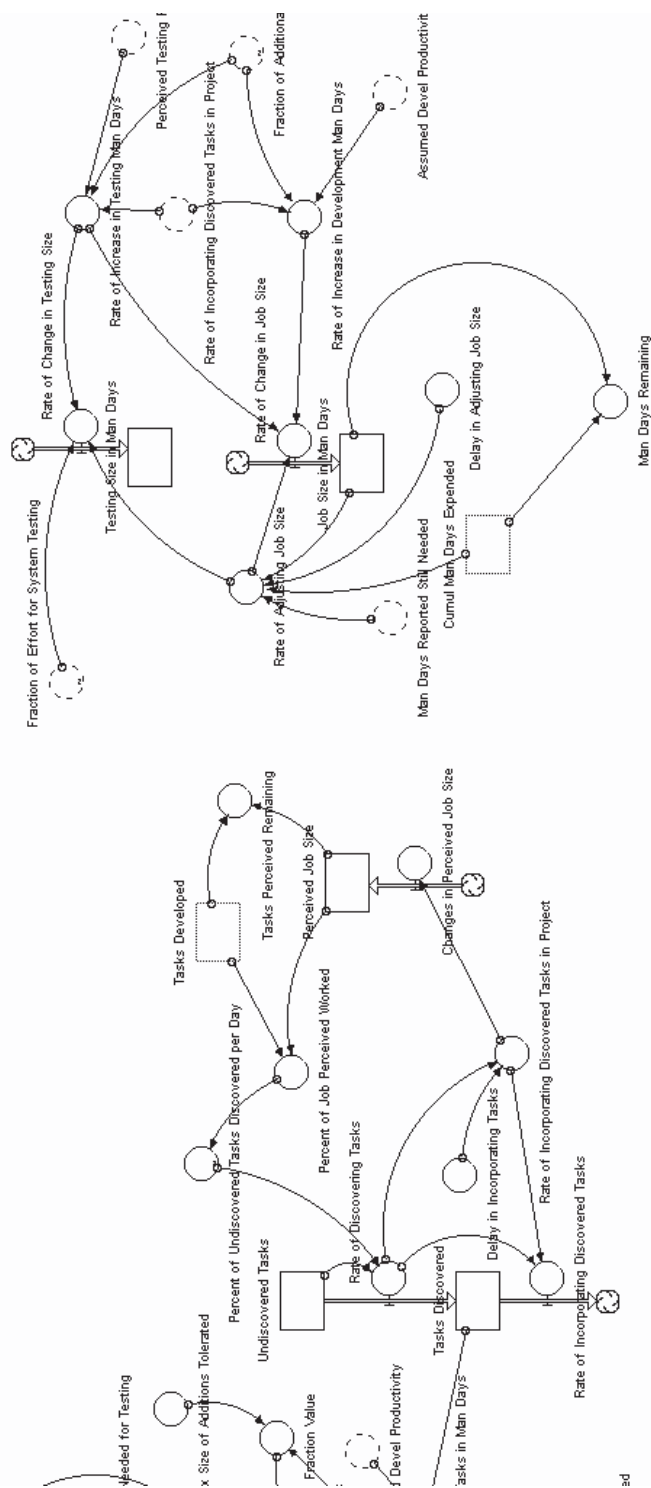


Figure 6.13. Control sector (right side).

to be still needed for testing converter is calculated as the tasks remaining to be tested divided by the perceived testing productivity. Tasks remaining to be tested is the difference between perceived job size in tasks and cumulative tasks tested (from the system testing sector). The perceived testing productivity equals the planned testing productivity at first, but actual testing productivity is used after testing actually begins.

The total man-days perceived to be still needed can be determined with the above. It is then compared with man-days remaining in the project to calculate the perceived shortage in man-days. The size of the project changes when the man-days reported still needed does not equal man-days remaining. The size change perception is translated into a stock called job size in man-days. The adjustment is represented by the rate of adjusting job size calculated as $(\text{goal} - \text{current level}) / \text{adjustment time}$. The adjustment time used is the delay in adjusting size, which is set to 3 days.

Job size can be adjusted for underestimation as well as falling behind in schedule. A parameter called task underestimation fraction is used to simulate the undersizing, as shown in Figure 6.14. Note that the project behavior modeled here is emblematic of some of the critiques in Section 6.3.1.5. The undiscovered job tasks are discovered at the rate of discovering tasks, which is a product of undiscovered job tasks and percent of undiscovered tasks discovered per day. A graph function is used that assumes that the rate of discovering previously unknown tasks increases as a project develops. This rate controls the flow of undiscovered tasks into the discovered tasks level, which is then drained by the delay in incorporating discovered tasks (set at 10 days delay).

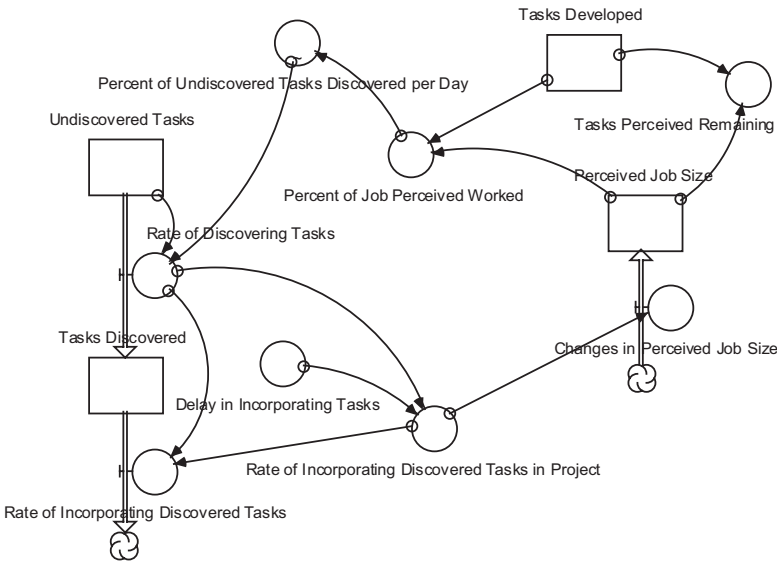


Figure 6.14. Task discovery.

The task underestimation effect is a major factor of the project dynamics in the Abdel-Hamid model. The rise in perceived job size affects many interrelated functions. The summary graph of the model using default settings is shown in Figure 6.15. The task underestimation fraction is set to 0.67, which produces the rise in perceived job size and corresponding delayed rise in the scheduled completion date (both are shown highlighted in Figure 6.15).

The project’s allocation of man-days changes when new tasks are discovered, but only when there is a significant amount. The perceived size of discovered tasks in man-days is divided by man-days perceived to be remaining for new tasks. If the relative size of discovered tasks is less than 1%, then nothing happens; if greater, then a portion of the additional tasks are translated into additional man-days using a graph function called fraction of additional tasks added to man-days. The new task effort is computed as the rate of increase in development man-days, and the testing effort is computed as the rate of increase in testing man-days.

There are also other converters calculated in this sector for use in other places. Man-days remaining is the difference between job size in man-days and cumulative man-days expended, and drives dynamics in the human resources and planning sectors. Schedule pressure is (total man-days perceived to be still needed–man-days remaining)/man-days remaining. This converter is used in the quality assurance and rework sector and manpower allocation sector. Other converters calculated here for other sectors are percent tasks reported completed, percent tasks perceived completed, real job size in tasks, and percent of job actually worked.

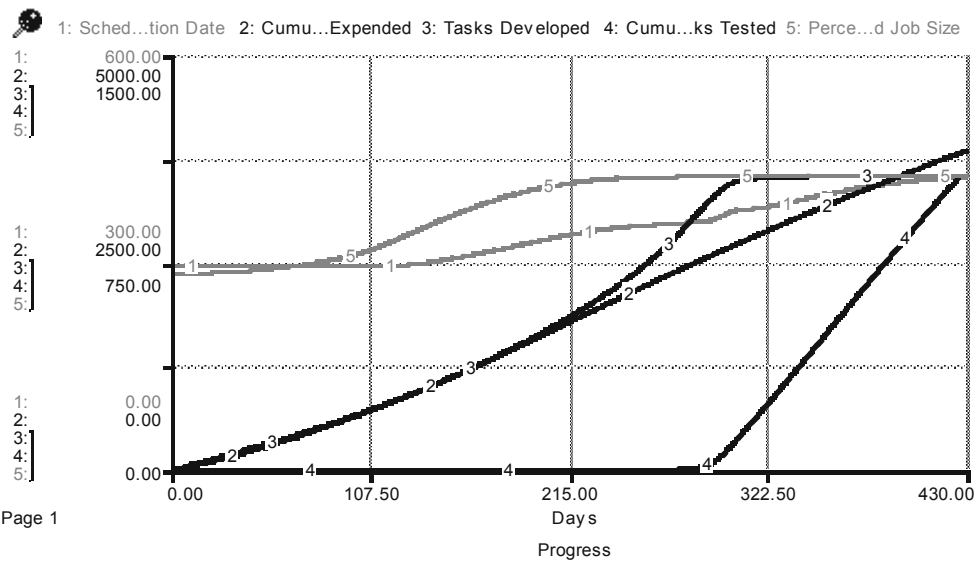


Figure 6.15. Task underestimation behavior (1: scheduled completion date, 2: cumulative man-days expended, 3: tasks developed, 4: cumulative tasks tested, 5: perceived job size).

6.3.1.5 *Insights and Implications of Abdel-Hamid's Integrated Project Model*

Abdel-Hamid's project model was a critical and highly enlightening work. However, all models are created to solve certain problems, come with specific assumptions, and cannot address everyone's needs. Abdel-Hamid's research issues and background were different from those of many readers, so this section summarizes the model's insights and implications to potentially be aware of. It is a very good model, but it models some poor behavior on software projects.

One advantage of the model is that it includes a good deal of important dynamic effects. It does well in illustrating some methods of poor software management (this can also be a downfall if certain policies are emulated, or the model becomes prescriptive instead of descriptive). It uses the very realistic notion that management perceptions, rather than true conditions, dictate actions. Delays in action are also important realistic considerations that the model covers well.

The inclusion of important personnel attributes like motivation, exhaustion, and schedule pressure effects are classic. These are vitally important considerations when managing a project. These structures of the model have been replicated and modified by many others.

The model is also strong in terms of planning and control structures. Many of these structures for project planning and control serve as good examples for other models. Abdel-Hamid showed in several of his publications how the planning and control policies embedded in the model make for interesting experiments with respect to usage of estimates [Abdel-Hamid 1993a] and making project judgments with fallible information [Abdel-Hamid et al. 1993]. However, the model contains too many elements for most managers to understand and use. For example, tracing the effects of Brooks's Law or schedule pressure through the entire model is difficult. The model also requires a good deal of parameters and functions to initialize for a particular environment. The experience has been that many of the inputs are very difficult to obtain without specialized collection efforts, and no known institution tracks the entire set of metrics by default. Some parts of the model also use idiosyncratic terminology. In particular, the error model is difficult for many to follow simply due to the semantics (e.g., active vs. passive errors).

Conversely, the software production model is not overly complex but very simplistic. Design and code are aggregated together. It does not provide the detailed visibility or management levers that would be of concern to many practitioners today.

The model may be wrongly used or overly relied on to perpetuate poor management practices, such as not being able to check status progress early on. Modern software management principles call for increased accounting and attention to earlier life-cycle activities. In the model, for example, the rate of discovering previously unknown tasks increases as a project develops, as opposed to a modern risk management approach with iterative processes that results in reduced volatility and less severe defect fixing as a project unfolds.

The control function uses a misleading indicator of progress to mimic previous projects based on interviews. Instead of measuring the tasks done, it defines progress as

the effort expended against plan. This is a very dangerous policy (earned value can also be abused this way). The “plan” can change during the simulation, but checking the status of actual work accomplished is always preferred on a real project.

The planning and control structures do not describe all environments, especially those with more modern and enlightened management. In a sense, the control structures admit defeat in not being able to check the status of progress early on; a good deal of the structure is dedicated to this phenomenon, and actual accomplishments are used only very late in the project.

The definition of QA is nonstandard and the model represents a nonoptimized, dangerous policy for quality. The defect finding and fixing activities modeled as QA are in practice generally performed by the same people who develop software artifacts. They are implicitly considered part of standard development activities in most installations. The QA model assumes that all QA tasks are batch processed in an allocated fixed amount of time (10 days) as a separate activity tacked onto development. Regardless of error levels or the amount of software per QA batch, the same amount of work per week goes into finding and fixing defects before system testing. Quality can be undermined with this approach or effort can be wasted without considering the variable workloads.

Many of the dynamics arise from the parameter for task underestimation. This parameter is not known exactly prior to a project, but it could be accounted for much earlier by adjusting up the size for expected volatility in the beginning, or modeling size with a distribution. If the uncertainty is planned in from the outset, then the debilitating dynamics would not occur. Management reserves or buffers can be created with contingency plans to have more people ready to go when necessary. This would be prudent planning and management.

The model has been and will continue to be valuable for follow-on research by others. Different parts of the model can be incorporated into other models and/or used for directed experimentation. Thanks to the separation of sectors provided by Margaret Johnson for this book, component sectors of the model can be individually exercised and incorporated into other models; for example, the human resources sector can stand alone by itself as a distinct model.

It was not developed for future calibration. The values for productivity, error rates, and so on are hard-coded into many graph functions, making it difficult to calibrate the model for specific environments. The metrics also come from Abdel-Hamid’s literature search and/or interviews, and often do not reflect currently expected values. These calibration issues should be addressed when adapting the model.

6.3.1.6 Early Follow-ons to Abdel-Hamid’s Work

Abdel-Hamid’s model was the basis for many of the subsequent early software process modeling efforts. Many of them were highlighted in a special issue of *American Programmer* [Yourdon 1993b] and another issue a year later [Yourdon 1994].

One of the first follow-on modeling efforts was performed at NASA Jet Propulsion Laboratories (JPL) by Chi Lin and others [Lin, Levary 1989, Lin et al. 1992]. She worked with Abdel-Hamid to extend his model for their projects to include require-

ments activities and multiple projects. In [Lin et al. 1992], some enhancements were made to the model and validated against project data. Several other organizations experimented with his model in the first few years.

One reason that few people were able to adapt the Abdel-Hamid model was its Dynamo implementation. His work at MIT was completed in the late 1980s when Stella with the Macintosh interface was just becoming popular. After that, few were willing to attempt the steep learning curve and legacy Fortran-like programming environment of Dynamo when much more user-friendly tools were available.

The divide between modern interfaces and the Abdel-Hamid model lasted for many years. It is certainly an arduous task to manually convert from a complex Dynamo model to another system dynamics tool, and several attempts were made to reverse engineer his model. Bellcore spent over a couple person-years to adapt the Abdel-Hamid model to Ithink for internal uses [Glickman 1994]. One adaptation used a more detailed implementation of COCOMO with effort adjustment factors.

Margaret Johnson produced a faithful reproduction of the Abdel-Hamid model in [Johnson 1995] and has provided it for the public domain through this book. Some of the examples in this book are derived from her work.

Johnson also worked with Ed Yourdon and Howard Rubin to further extend the Abdel-Hamid model. They investigated the effects of software process improvement through their adapted models [Rubin et al. 1994] and integrated system dynamics with the Estimacs cost estimation model.

Subsequent work was done by Abdel-Hamid in various experiments to test management policies afforded by the model. In [Abdel-Hamid 1993a], the problem of continuous estimation throughout a project was investigated. Experimental results showed that up to a point, reducing an initial estimate saves cost by decreasing wasteful practices. However, the effect of underestimation is counterproductive beyond a certain point because initial understaffing causes a later staff buildup.

Abdel-Hamid also developed a preliminary model of software reuse [Abdel-Hamid 1993b] with a macroinventory perspective instead of an individual project perspective. It captures the operations of a development organization as multiple software products are developed, placed into operation, and maintained over time. Preliminary results showed that a positive feedback loop exists between development productivity, delivery rate, and reusable component production. The positive growth eventually slows down from the balancing influence of a negative feedback loop, since new code production decreases as reuse rates increase. This leads to a decreasing reuse repository size since older components are retired and less new code is developed for reuse. The reuse inventory modeling effort is a prime example of modeling the process versus a project. The time horizon is such that there is no notion of a project start or end point.

The Abdel-Hamid model has lived on in various other forms over the years. It was a starting point for many of the other applications described in this book. For example, see the section in Chapter 5 on the SPMS model for requirements volatility and how its lineage can be traced back to the Abdel-Hamid model.

Abdel-Hamid's major publications in software-related journals are summarized in Appendix B: Annotated System Dynamics Bibliography. There are a few articles not listed from business journals and other venues, but they contain very little informa-

tion beyond those already listed. The interested reader can look at *Software Project Dynamics* [Abdel-Hamid, Madnick 1991] or [Johnson 1995] for more details of the model.

6.4 SOFTWARE BUSINESS CASE ANALYSIS

It behooves every industrial software project or organizational process to have a business rationale for its existence, and it should be compatible with other plans, goals, or constraints. The business case helps sets the stage for an instantiated software process, and the relationship is a dynamic one.

Thus, the study of software business cases is important to understanding development and evolution processes. Executive decision making should be based on the best information available and linked to stakeholder value. Modeling is valuable if questions like the following need quantitative answers:

- How will customers, suppliers, partners, and competitors act in the future?
- How are changes in the market structure affecting business?
- What are the implications of trade-offs between financial and nonfinancial balanced scorecard measures?
- What effect will management actions have on internal processes and productivity?

[Reifer 2001] is a good reference on how to justify expenditures and prepare successful business cases. It provides useful examples of what to do and what not to do via the case study approach. Dynamic modeling is not explicitly addressed, but the general analysis methods are easily integrated with system dynamics. The examples in this section show how systems thinking and system dynamics can be brought to bear on business case analysis. This usage also falls under the general term of *business simulation*, by which one can quantify the time-based financial return from a software solution, explore alternative business scenarios, and support decision making to optimize business processes.

Business value attainment should be a key consideration when designing software processes. Ideally, they are structured to meet organizational business goals, but it is usually difficult to integrate the process and business perspectives quantitatively. The examples in this section are based on actual client experiences, but have been adapted for simplicity and recalibrated with nonproprietary data.

The first example shows how simulation can be used to assess product strategies from a value-based perspective in order to analyze business case trade-offs. The model relates the dynamics between product specifications, investment costs, schedule, software quality practices, market size, license retention, pricing, and revenue generation. It allows one to experiment with different product strategies, software processes, marketing practices, and pricing schemes while tracking financial measures over time. It can be used to determine the appropriate balance of process activities to meet goals.

Examples are developed for varying scope, reliability, delivery of multiple releases, and determining the quality sweet spot for different time horizons. Results show that optimal policies depend on various stakeholder value functions, opposing market factors, and business constraints. Future model improvements are also identified.

6.4.1 Example: Value-Based Product Modeling

Software-related decisions should not be extricated from business value concerns. Unfortunately, software engineering practice and research frequently lacks a value-oriented perspective. Value-Based Software Engineering (VBSE) integrates value considerations into current and emerging software engineering principles and practices [Boehm, Huang 2003, Biffl et al. 2005]. This model addresses the planning and control aspect of VBSE to manage the value delivered to stakeholders. Techniques to model cost, schedule, and quality are integrated with business case analysis to allow trade-off studies in a commercial software development context. Business value is accounted for in terms of return on investment (ROI) of different product and process strategies. Table 6.6 is a high-level summary of the model.

It is a challenge to trade off different software attributes, particularly between different perspectives such as business and software development. Software process modeling and simulation can be used to reason about software value decisions. It can help find the right balance of activities that contribute to stakeholder value with other constraints such as cost, schedule, or quality goals.

A value-oriented approach provides explicit guidance for making products useful to people by considering different people’s utility functions or value propositions. The value propositions are used to determine relevant measures for given scenarios.

Two major aspects of stakeholder value are addressed here. One is the business value to the development organization stemming from software sales. Another is the value to the end-user stakeholder from varying feature sets and quality. Production functions relating different aspects of value to their costs were developed and are included in the integrated model.

Table 6.6. Value-based product model overview

Purpose: Strategic Management, Planning			
Scope: Long-term Product Evolution			
Inputs and Parameters	Levels	Major Relationships	Outputs
<ul style="list-style-type: none">• Start staff• Function points• Manpower buildup parameter• Reliability setting	<ul style="list-style-type: none">• Defects• Investments• Potential market share• Active licenses• Perceived quality• Cumulative investment• Cumulative revenue	<ul style="list-style-type: none">• Market share production function vs. features• Sales production function vs. reliability• Quality perception• Defect rates• Staffing rate• Market dynamics	<ul style="list-style-type: none">• Return on investment• Market share• Defects• Sales• Staffing

6.4.1.1 Model Overview

The model represents a business case for commercial software development in which the user inputs and model factors can vary over the project duration, as opposed to a static model. Input parameters can also be modified interactively by the user during the course of a run and the model responds to the midstream changes. It can be used dynamically before or during a project. Hence, it is suitable for “flight simulation” training or actual project usage to reflect actuals to date. The model can be used to assess the effects of combined strategies by varying inputs such as scope and required reliability independently or simultaneously.

The sectors of the model and their major interfaces are shown in Figure 6.16. The software process and product sector computes the staffing profile and quality over time based on the software size, reliability setting, and other inputs. The staffing rate becomes one of the investment flows in the finances sector, whereas the actual quality is a primary factor in market and sales. The resulting sales are used in the finance sector to compute various financial measures.

Figure 6.17 shows a diagram of the software process and product sector. It models the internal dynamics between job size, effort, schedule, required reliability, and quality. The staffing rate over time is calculated with a version of Dynamic COCOMO [Boehm et al. 2000] using a variant of a Rayleigh curve calibrated to the COCOMO II cost model at the top level. The project effort is based on the number of function points and the reliability setting. There are also some Rayleigh curve parameters that determine the shape of the staffing curve.

There is a simple defect model to calculate defect levels used in the market and sales sector to modulate sales. Defect generation is modeled as a coflow with the software development rate, and the defect removal rate accounts for their finding and fixing. See Chapter 3 for more background on these standard flow structures for effort and defects.

Figure 6.18 shows the market and sales sector accounting for market share dynamics and software license sales. The perceived quality is a reputation factor that can re-

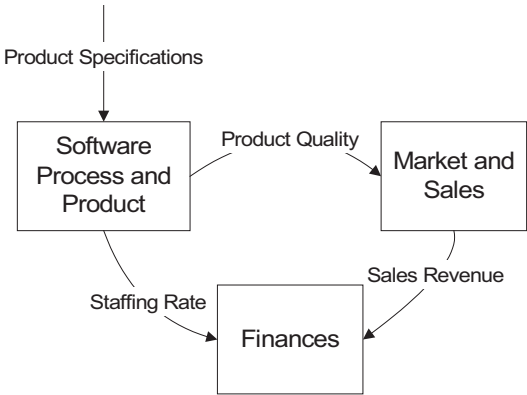


Figure 6.16. Model sectors and major interfaces.

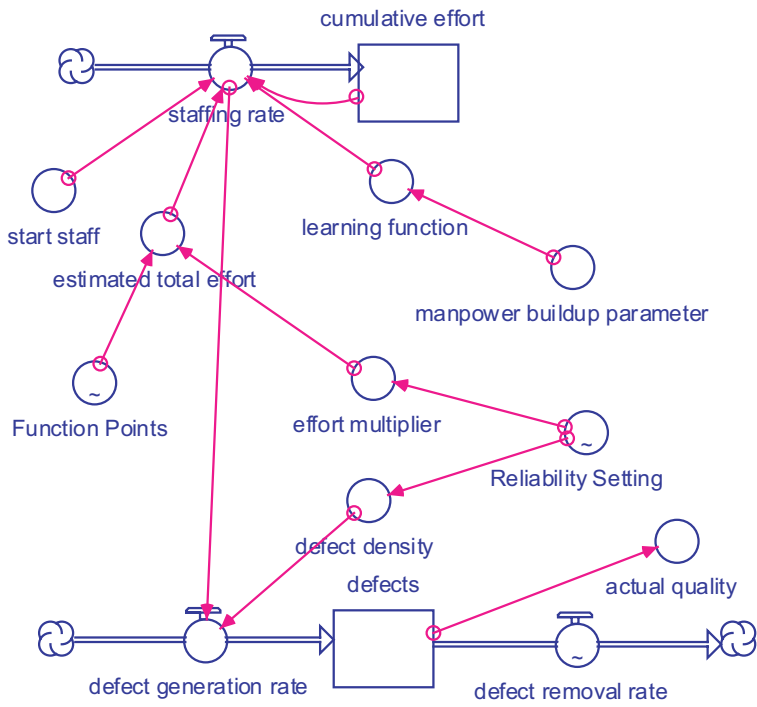


Figure 6.17. Software process and product sector.

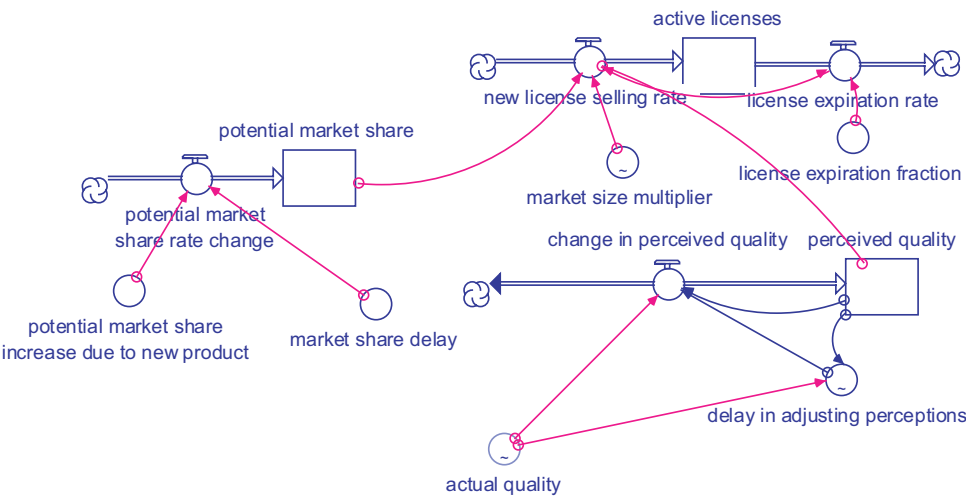


Figure 6.18. Market and sales sector.

duce the number of sales if products have many defects, as described in the next section. The market and sales model presented herein is a simplification of a more extensive model being used in industry that accounts for additional marketing initiatives and software license maintenance sales.

The finance sector is shown in Figure 6.19. It includes cash flows for investment and revenue. Investments include the labor costs for software development, maintenance, and associated activities. Revenue is derived from the number of license sales. Sales are a function of the overall market size and market share percentage for the software product. The market share is computed using a potential market share adjusted by perceived quality. The additional market share derivable from a new product is attained at an average delay time.

6.4.1.1.1 QUALITY MODELING AND VALUE FUNCTIONS. For simplification, software reliability as defined in the COCOMO II model [Boehm et al. 2000] is used as a proxy for all quality practices. It models the trade-off between reliability and development cost. There are four different settings of reliability from low to very high that correspond to four development options. The trade-off is increased cost and longer development time for increased quality. Table 6.7 lists the reliability rating definitions, the approximate mean times between failures, and relative costs from the COCOMO II model. This simplification can be replaced with a more comprehensive quality model to account for specific practices (see future work in Section 6.4.1.3 and the chapter ex-

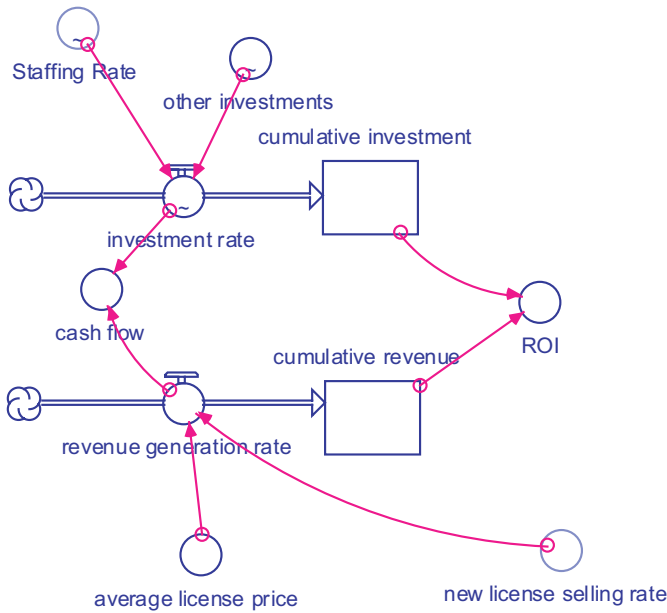


Figure 6.19. Finance sector.

Table 6.7. Reliability definitions.

Reliability Rating	Defect Impact	Approximate Mean Time Between Failure (Hours)	Relative Cost
Low	Small, recoverable losses	10	0.92
Nominal	Moderate, recoverable losses	300	1.00
High	Large, unrecoverable losses	10,000	1.10
Very High	Human life	300,000	1.26

ercises). The resulting quality will modulate the actual sales relative to the highest potential. A lower quality product will be done quicker; it will be available on the market sooner but sales will suffer from poor quality. The mapping between reliability and the relative impact to sales from iterated Delphi surveys is captured as a production function and used in the model.

Collectively, there are two value-based production functions in the model to describe value relationships (they are illustrated in the first applied example). A market share production function addresses the organizational business value of product features. The business value is quantified in terms of added potential market share attainable by the features. The relationship assumes that all features are implemented to the highest quality. Since the required reliability will impact how well the features actually work, the relationship between reliability costs and actual sales is needed to vary the sales due to quality.

The market share production function in Figure 6.20 relates the potential business value to the cost of development for different feature sets. The actual sales production function versus reliability costs is shown in Figure 6.21, and it is applied against the potential market capture. The four discrete points correspond to required reliability levels of low, nominal, high, and very high. Settings for the three cases described in the next section are shown in both production functions.

The value function for actual sale attainment is relevant to two classes of stakeholders. It describes the value of different reliability levels in terms of sales attainment, and is essentially a proxy for user value as well. It relates the percent of potential sales attained in the market against reliability costs. Illustrations of the production functions are shown in the next section.

The market and sales sector also has a provision to modulate sales based on the perceived quality reputation. A bad quality reputation takes hold almost immediately with a buggy product (bad news travels fast), and takes a long time to recover from in the market perception even after defects are fixed. This phenomenon is represented with asymmetrical information smoothing as shown in Figure 6.22 with a variable delay in adjusting perceptions.

The graph in Figure 6.22 shows a poor quality release at time = 3 years with a followup release to the previous quality level. Whereas the perceived quality quickly plummets after the bad release, it rises much more slowly even when the actual quality has improved. There are many historical examples of this phenomena.

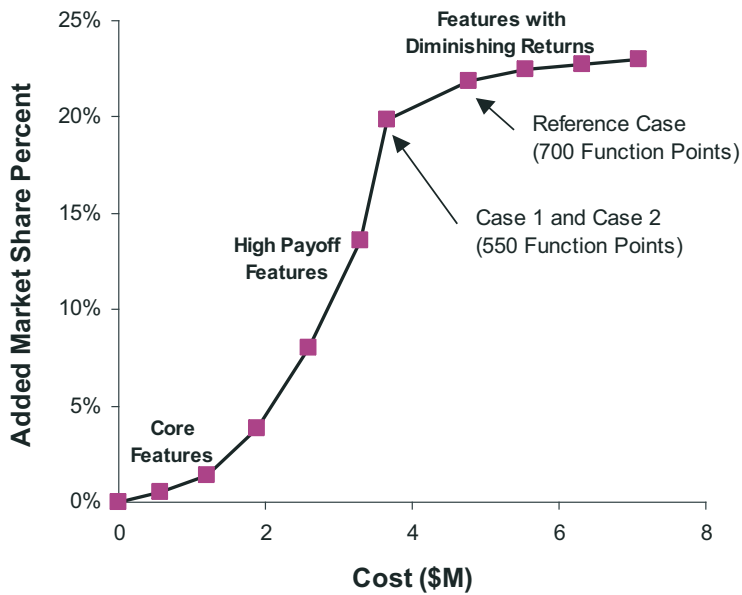


Figure 6.20. Market share production function and feature sets.

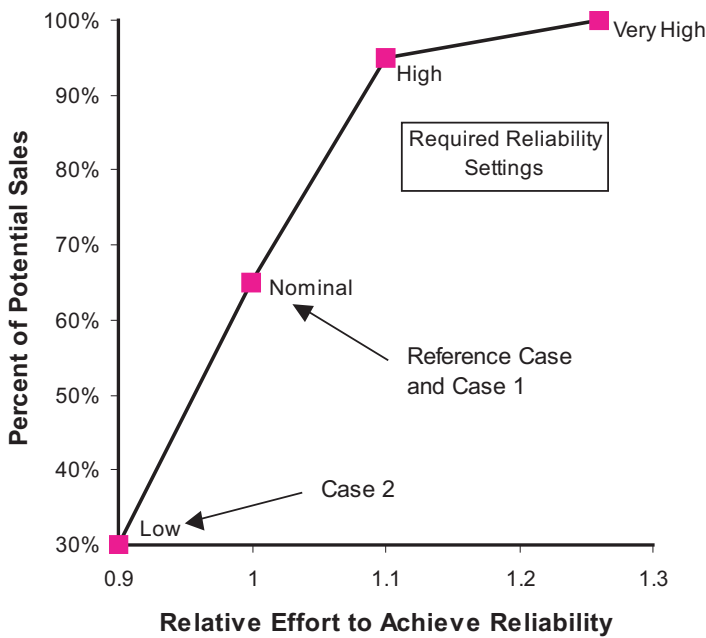


Figure 6.21. Sales production function and reliability.

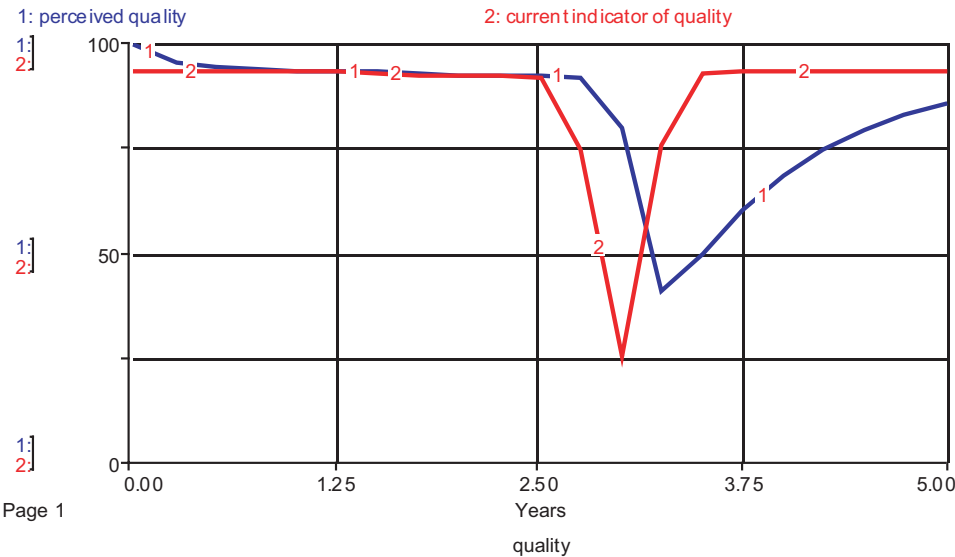


Figure 6.22. Perceived quality trends with high- and low-quality product deliveries.

6.4.1.2 Applications

Several representative business decision scenarios are demonstrated next. The first example demonstrates the ability to dynamically assess combined strategies for scope and reliability. The second example looks at strategies of multiple releases of varying quality. Finally, the model is used to determine a process sweet spot for reliability.

6.4.1.2.1 DYNAMICALLY CHANGING SCOPE AND RELIABILITY. The model can be used to assess the effects of individual and combined strategies for overall scope and reliability. This example will show how it can be used to change product specifications midstream as a replan. Static cost models typically do not lend themselves to replans after the project starts, as all factors remain constant through time. This dynamic capability can be used in at least two ways by a decision-maker:

1. Assessing the impact of changed product specifications during the course of a project
2. Before the project starts, determining if and how late during the project specifications can be changed based on new considerations that might come up

Three cases are simulated: (1) an unperturbed reference case, (2) a midstream descoping of the reference case, and (3) a simultaneous descoping and lowered required reliability. Such descoping is a frequent strategy to meet time constraints by shedding features. See Figure 6.20 for the market share production function and Figure 6.21 for the potential business value function for these three cases.

Figure 6.23 shows a sample control panel interface to the model. The primary inputs for product specifications are the size in function points (also called scope) and required reliability. The number of function points is the size needed to implement given features. The size and associated cost varies with the number of features to incorporate.

The reliability settings on the control panel slider are the relative effort multipliers needed to achieve reliability levels from low to very high. These are input by the user via the slider for “Reliability Setting.” The attainable market share derived from the production function in Figure 6.20 is input by the user on the slider “Potential Market Share Increase.”

Figure 6.23 also shows the simulation results for the initial reference case. The default case of 700 function points is delivered with nominal reliability at 2.1 years with a potential 20% market share increase. This project is unperturbed during its course and the 5 year ROI of the project is 1.3.

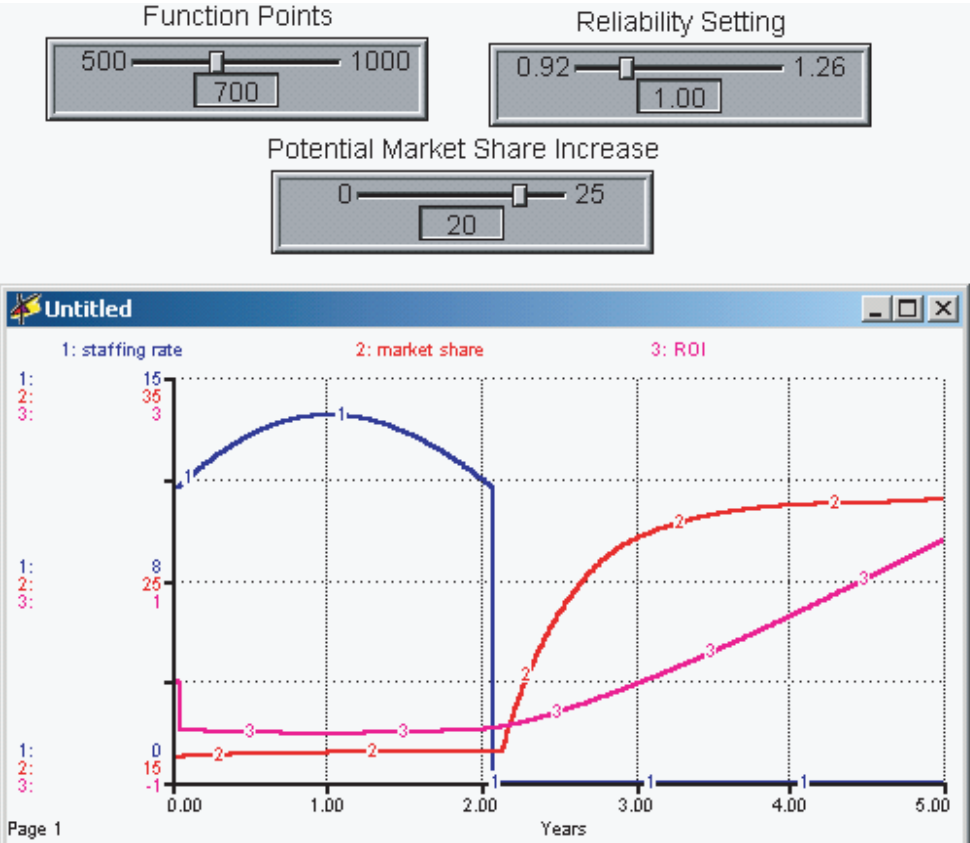


Figure 6.23. Sample control panel and reference case (unperturbed).

Case 1 in Figure 6.24 illustrates the initial case perturbed at 0.5 years to descope low-ROI features (see Figure 6.20 and Figure 6.21 for the points on the production function). The scope goes down to 550 function points and the staffing profile adjusts dynamically for it. The schedule is reduced by a few months. In this case, the potential market share increase is lowered by only two percentage points to 18%. With lower development costs and earlier delivery, the ROI increases substantially to 2.2.

A combined strategy is modeled in Figure 6.25 for Case 2. The scope is decreased the same as before in Case 1 (Figure 6.24), plus the reliability setting is lowered from nominal to low. Though overall development costs decrease due to lowered reliability, the market responds poorly. This case provides the worst return of the three options and market share is lost instead of gained.

In Case 2, there is an early hump in sales due to the initial attention paid to the brand-new product, but the market soon discovers the poor quality and then sales suffer dramatically. These early buyers and others assume that the new product will have the previous quality of the product line and are anxious to use the new, “improved” product. Some may have preordered and some are early adopters that always buy when new products come out. They are the ones that find out about the lowered quality and word starts spreading fast.

A summary of the three cases is shown in Table 6.8. Case 1 is the best business plan to shed undesirable features with diminishing returns. Case 2 severely hurts the enterprise because quality is too poor.

6.4.1.2.2 MULTIPLE RELEASES. This example shows a more realistic scenario for maintenance and operational support. Investments are allocated to ongoing maintenance and the effects of additional releases of varying quality are shown. The refer-

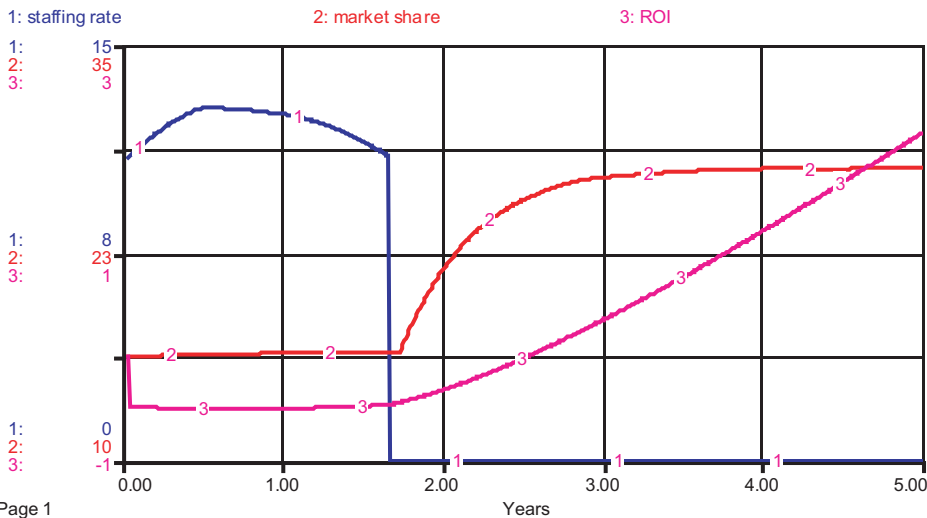


Figure 6.24. Case 1—descope of low ROI features at time = 0.5 years.

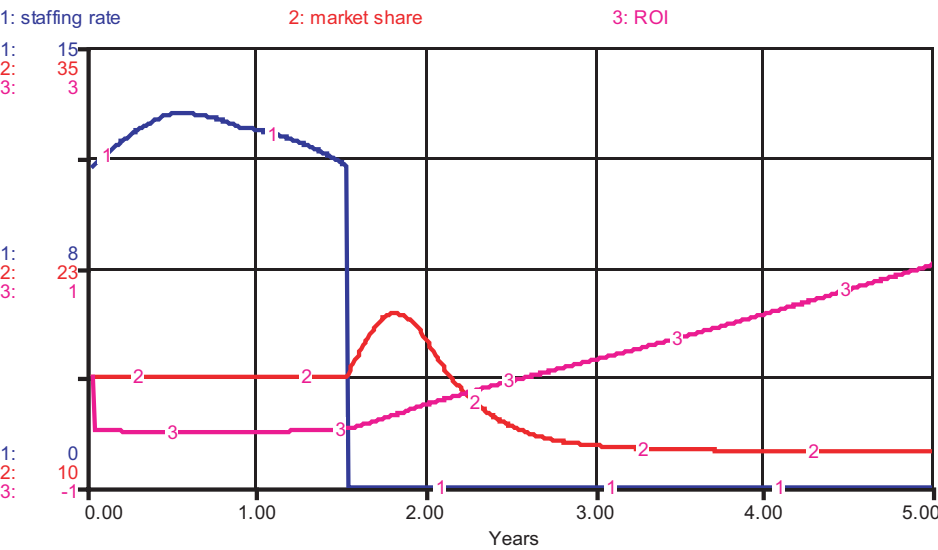


Figure 6.25. Case 2—descope of low ROI features and reliability lowering at time = 0.5 years.

ence case contains two product rollouts at years 1 and 3, each with the potential to capture an additional 10% of the market share. These potentials are attained because both deliveries are of high quality, as seen in Figure 6.26 and Figure 6.27.

A contrasting case in Figure 6.28 and Figure 6.29 illustrates the impact if the second delivery has poor quality yet is fixed quickly (Figure 6.22 shows the quality trends for this case). This results in a change of revenue from \$11.5 M to \$9.6M and ROI from 1.3 to 0.9.

This example is another illustration of the sensitivity of the market to varying quality. Only one poor release in a series of releases may have serious long-term consequences.

6.4.1.2.3 FINDING THE SWEET SPOT. This example shows how the value-based product model can support software business decision making by using risk conse-

Table 6.8. Case summaries.

Case	Delivered Size (Function Points)	Delivered Reliability Setting	Cost (\$M)	Delivery Time (Years)	Final Market Share	ROI
Reference Case: Unperturbed	700	1.0	4.78	2.1	28%	1.3
Case 1: Descope	550	1.0	3.70	1.7	28%	2.2
Case 2: Descope and Lower Reliability	550	.92	3.30	1.5	12%	1.0

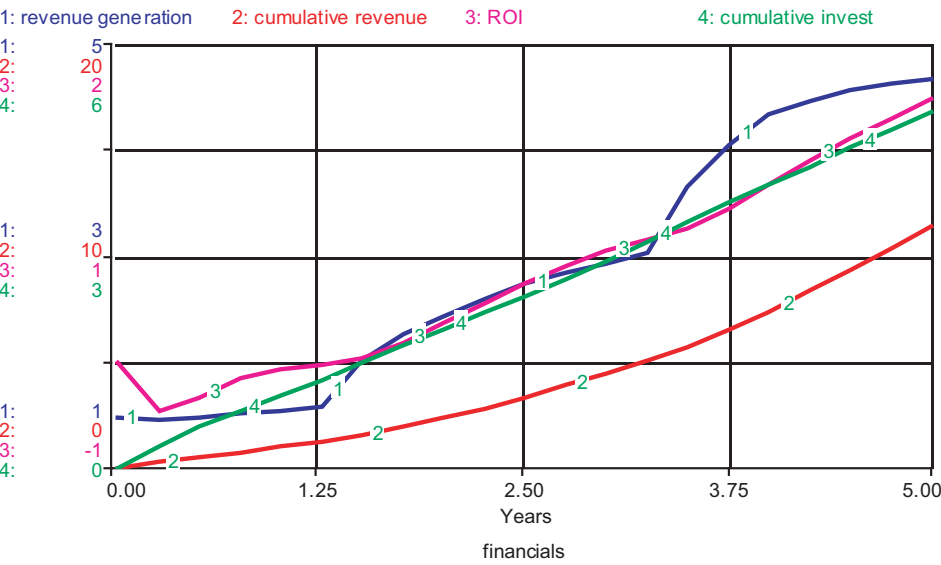


Figure 6.26. Reference case financials for two high-quality product deliveries (1: revenue generation rate, 2: cumulative revenue, 3: ROI, 4: cumulative investment).



Figure 6.27. Reference case sales and market for two high-quality product deliveries (1: active licenses, 2: new license selling rate, 3: license expiration rate, 4: potential market share).

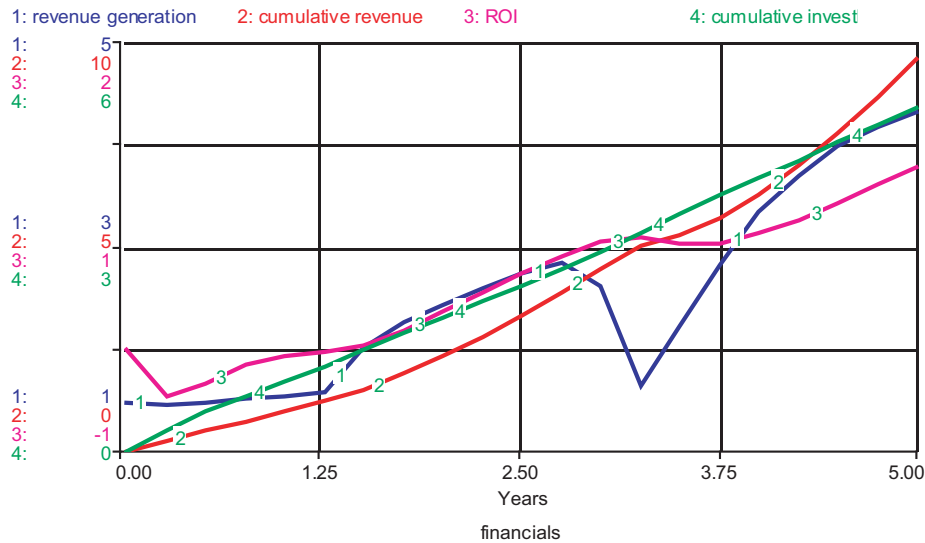


Figure 6.28. Financials for high- and low-quality product deliveries (1: revenue generation rate, 2: cumulative revenue, 3: ROI, 4: cumulative investment).



Figure 6.29. Sales and market for high- and low-quality product deliveries (1: active licenses, 2: new license selling rate, 3: license expiration rate, 4: potential market share).

quence to find the quality sweet spot with respect to ROI. The following analysis steps are performed to find the process sweet spot:

- Vary reliability across runs
- Assess risk consequences of opposing trends: market delays and bad quality losses
- Sum market losses and development costs
- Calculate resulting net revenue to find process optimum

The risk consequences are calculated for the different options. Only point estimates are used for the sake of this example. A more comprehensive risk analysis would consider probability distributions to obtain a range of results. Probability is considered to be constant for each case and is not explicitly used in the calculations. Only the costs (or losses) are determined.

A set of runs is performed that simulate the development and market release of a new 80 KSLOC product. The product can potentially increase market share by 30%, but the actual gains depend on the level of quality. Only the highest quality will attain the full 30%. Other parameterizations are an initial total market size = \$64M annual revenue, the vendor has 15% initial market share, and the overall market doubles in 5 years.

A reference case is needed to determine the losses due to inferior quality. The expected revenues for a subquality delivery must be subtracted from the maximum potential revenues (i.e., revenue for a maximum quality product delivered at a given time). The latter is defined as delivering a maximum quality product at a given time that achieves the full potential market capture. The equation for calculating the loss due to bad quality is

$$\text{bad quality loss} = \text{maximum potential revenue with same timing} - \text{revenue}$$

The loss due to market delay is computed, keeping the quality constant. To neutralize the effect of varying quality, only the time of delay is varied. The loss for a given option is the difference between the revenue for the highest quality product at the first market opportunity and the revenue corresponding to the completion time for the given option (assuming the same highest quality). It is calculated by

$$\text{market delay cost} = \text{maximum potential revenue} - \text{revenue}$$

Figure 6.30 shows the experimental results for an 80 KSLOC product, fully compressed development schedules, and a 3-year revenue timeframe for different reliability options. The resultant sweet spot corresponds to reliability = high. The total cost consisting of delay losses, reliability losses, and development cost is minimum at that setting for a 3 year time horizon.

The sweet spot depends on the applicable time horizon, among other things. The horizon may vary for several reasons such as another planned major upgrade or new

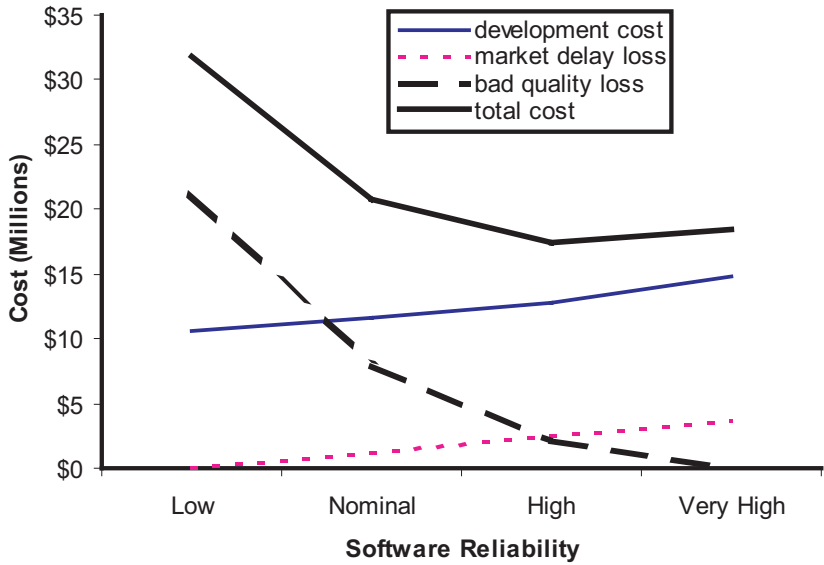


Figure 6.30. Calculating reliability sweet spot (3 year time frame).

release, other upcoming changes in the business model, or because investors mandate a specific time frame in which to make their return.

The experiment was rerun for typical time horizons of 2, 3, and 5 years using a profit view (the cost view is transformed into a profit maximization view by accounting for revenues). The results are shown in Figure 6.31. The figure illustrates that the sweet spot moves from reliability equals low to high to very high. It is evident that the optimal reliability depends on the time window. A short-lived product (a prototype is an extreme example) does not need to be developed to as stringent reliability standards as one that will live in the field longer.

6.4.1.3 Conclusions and Future Work

It is crucial to integrate value-based methods into the software engineering discipline. To achieve real earned value, business value attainment must be a key consideration when designing software products and processes. This work shows several ways that software business decision making can improve with value information gained from simulation models that integrate business and technical perspectives.

The model demonstrates a stakeholder value chain whereby the value of software to end users ultimately translates into value for the software development organization. It also illustrates that commercial process sweet spots with respect to reliability are a balance between market delay losses and quality losses. Quality does impact the bottom line.

The model can be elaborated to account for feedback loops to generate revised product specifications (closed-loop control). This feedback includes:

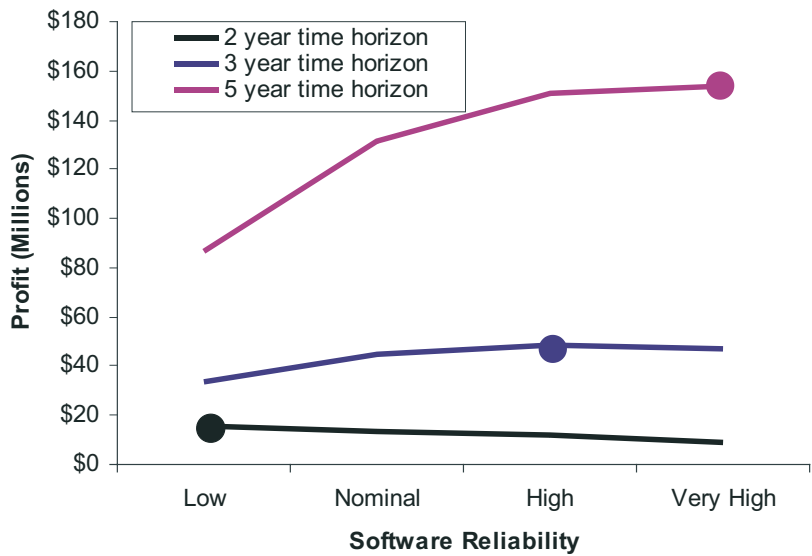


Figure 6.31. Reliability sweet spot as a function of time horizon.

- External feedback from user to incorporate new features
- Internal feedback on product initiatives from an organizational planning and control entity to the software process

A more comprehensive model would consider long term product evolution and periodic upgrades. Another related aspect to include is general maintenance by adding explicit activities for operational support.

The product defect model can be enhanced with a dynamic version of the Constructive Quality Model (COQUALMO) [Chulani, Boehm 1999] to enable more constructive insight into quality practices. This would replace the current construct based on the single factor for required software reliability.

Other considerations for the model are in the market and sales sector. The impact of different pricing schemes and varying market assumptions on initial sales and maintenance can all be explored. Some of these provisions are already accounted for in a proprietary version of the model.

The model application examples were run with idealized inputs for the sake of demonstration, but more sophisticated dynamic scenarios can be easily handled to model real situations. For example, discrete descopings were shown, but in many instances scope will exhibit continuous or fluctuating growth over time.

More empirical data on the relationships in the model will also help identify areas of improvement. Assessment of overall dynamics includes more collection and analysis of field data on business value and quality measures from actual software product rollouts.

6.5 PERSONNEL RESOURCE ALLOCATION

The proper allocation of people to different tasks is one of the most important jobs of software management. There are a variety of policies used for resource allocation, and this section overviews some ways to model them. Some of the referenced models are also provided for usage.

6.5.1 Example: Resource Allocation Policy and Contention Models

Several models of different resource allocation policies are described and analyzed below. The first one is a dynamic allocation policy whereby resources are allocated proportionally to larger tasks. Next are models for situations in which resource staffing can be described with parameterized shape profiles. Lastly is a simple project contention model that was used to examine and modify personnel allocation policies in industry.

6.5.1.1 Squeaky Wheel Gets the Grease

The infrastructure in Figure 6.32 is a modification of the resource allocation example in Chapter 3 that can be used to allocate resources among competing tasks. It is called

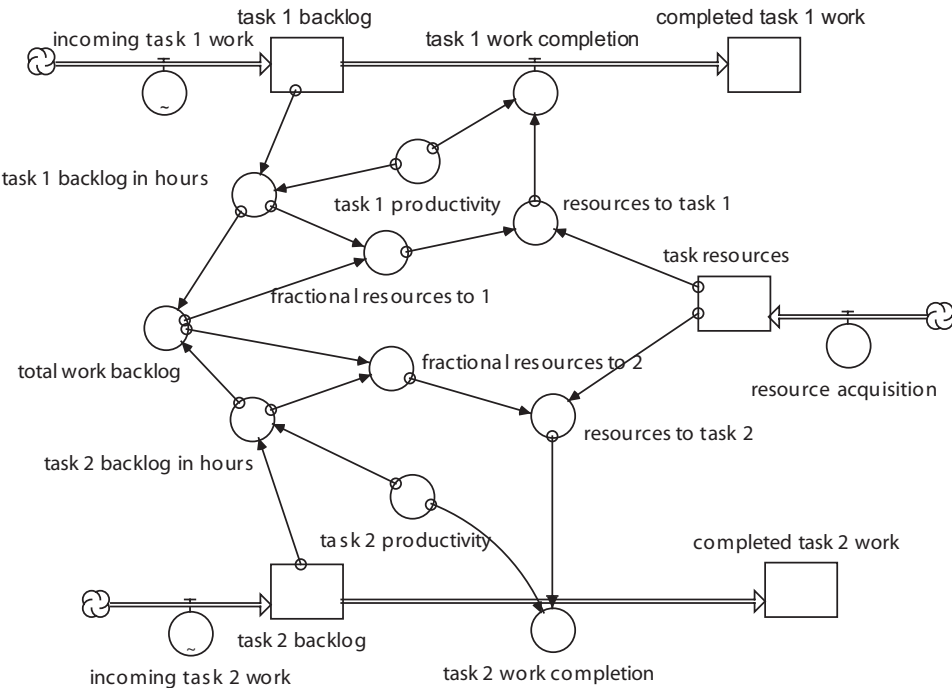


Figure 6.32. Fractional resource allocation structure.

the “squeaky wheel” algorithm because more fractional resources are allocated to those tasks with the highest backlog of work. Productivity serves as a weighting factor to account for differences in the work rates in calculating the backlog hours. The main equation that allocates the fraction of total resources to a given task is:

$$\text{fractional resources to 1} = \frac{\text{task 1 backlog in hours}}{\text{total work backlog}}$$

This type of policy is used frequently when creating project teams. With a fixed staff size, proportionally more people are dedicated to the larger tasks. For example, if one software component is about twice as large as another, then that team will have roughly double the people. The component backlogs are monitored throughout a project, and people are reallocated as the work ratios change based on the respective estimates to complete.

Figure 6.33 shows the results of a simulation run in which both tasks start with equal backlogs and 50% allocations due to equal incoming work rates. At time = 6 more incoming work starts streaming into the backlog for task 2 (see the graph for incoming task 2 work). The fractional allocations change, and more resources are then allocated to work off the excess backlog in task 2. The graph for fractional resources to 2 increases at that point and slowly tapers down again after some of the task 2 work is completed. This example models the allocation between two tasks, but it can be easily extended to cover more competing tasks. The model is contained in the file *proportional resource allocation.itm*.

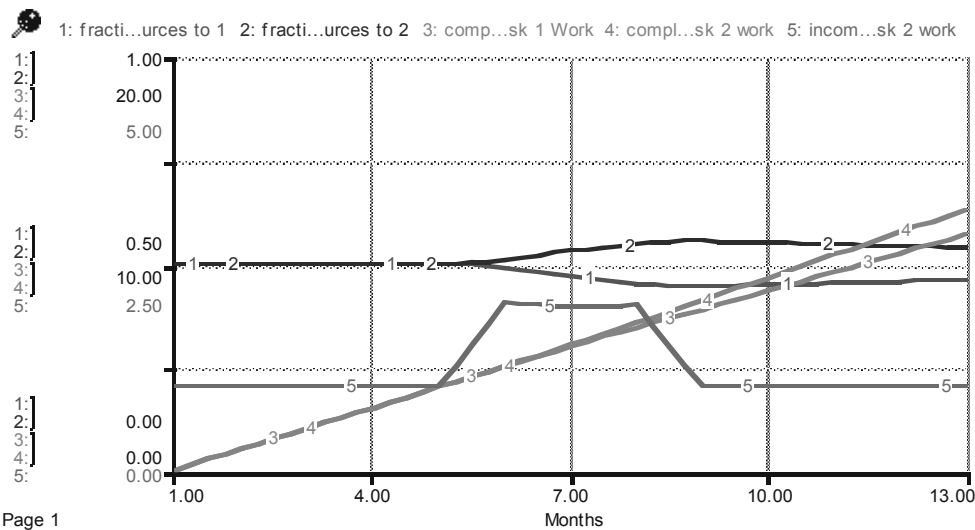


Figure 6.33. Fractional resource allocation simulation output (1: fractional resources to 1, 2: fractional resources to 2, 3: completed task 1 work, 4: completed task 2 work, 5: incoming task 2 work).

6.5.1.2 *Parameterized Allocation Profiles*

Some models use fixed resource levels or relative parameterized shapes that drive the simulation. One example is the portion of the Madachy inspection model in Chapter 5 that allocates staff for the primary development (a variable staffing portion is used for defect fixing based on the number of defects). Dynamic staffing profiles for design and coding activities are created based on the project size and complexity.

Figure 6.34 shows the dimensionless default staffing profile for design work that is proportionally sized for the project at hand based on the overall effort and schedule acceleration. It is a function of a schedule ratio of actual time to the initially estimated schedule time for development (consisting of design and coding activities that cover 85% of the overall lifecycle). The portion of staffing used for design, as defined by the graph in Figure 6.34, is taken from the overall manpower level to calculate the design manpower rate that actually drives the design activities. This structure is shown in Figure 6.35 without the other information links to the various model elements.

The inspection model also has provisions to add variable staff based on the defect levels. This is another type of allocation policy that adds the right amount of people to fix the defects that are detected. The following subsection discusses this and other defect-fixing allocation policies.

6.5.1.3 *Dynamic Resource Allocation with Initially Fixed Levels*

The previous discussion for using fixed resource levels is only applicable when the project scope does not change during the course of a simulation. In the Madachy inspection model, the overall project size stays constant (e.g., the design staffing will not change after the simulation starts) but the varying levels of defects impose different levels of fixing effort. The defect-fixing effort is modeled as a separate activity.

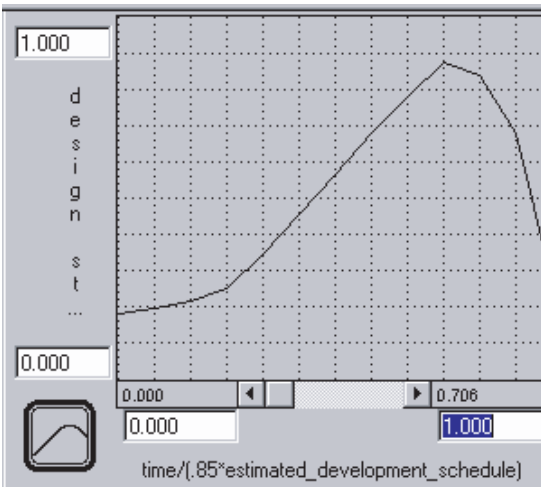


Figure 6.34. Adjustable design staffing curve.

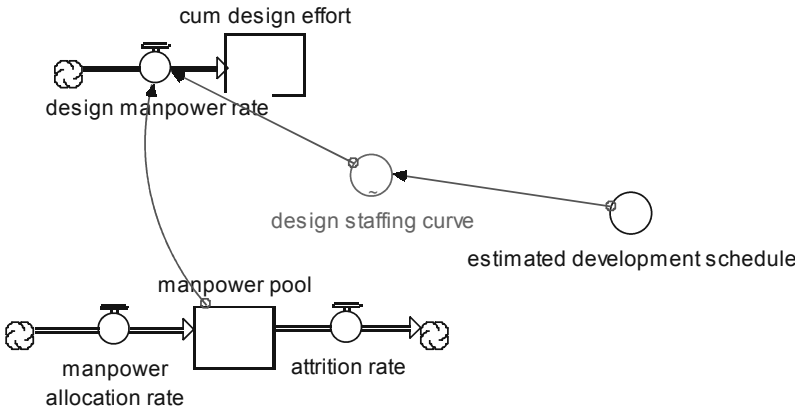


Figure 6.35. Structure to allocate fixed staffing level for design.

An example of staffing levels changing from initial simulation values is in the Abdel-Hamid project model. The initial values for personnel are calculated using the COCOMO model. However, the initial schedule and effort values are augmented by dynamic effects for task underestimation, other unexpected delays, and defect levels. The default run for the Abdel-Hamid model assumes a task underestimation factor of 0.67, so more staff is added as the additional work is needed. Thus, there are policy provisions for reassessing the project effort and allocating more people as necessary, given certain constraints for adding staff (see Section 6.3.1 for a detailed description of these constraints). It is important to note that the model has these factors for management and policy impact that are not included in traditional estimation models.

6.5.1.4 Project Contention

At Litton, creation of a model of resource switching was motivated by reactive behavior on the part of some executive management, whereby senior people were juggled between projects to fix short-term problems per customer whims. The software engineering department wanted to demonstrate that the practice was counterproductive in terms of cost/schedule performance. Both projects suffered learning curve and communication overhead drains that overwhelmed any gains on the initial “troubled” project. Additionally, the juggled individuals themselves experienced losses due to the multiple context switching. Their net output was much less when attempting to work on several tasks at once compared to a single task.

The model is shown in Figure 6.36 and provided in the file *project contention.itm*. Senior people are designated with “sr.” One key to the model is expressing the learning curve as a function of volume (i.e., tasks produced) rather than time per se. See [Raccoon 1996] for a detailed discussion of software learning curves. When there are interruptions and context switching between projects, then an individual going back to a project requires longer than the interruption time to reach previous productivity levels.

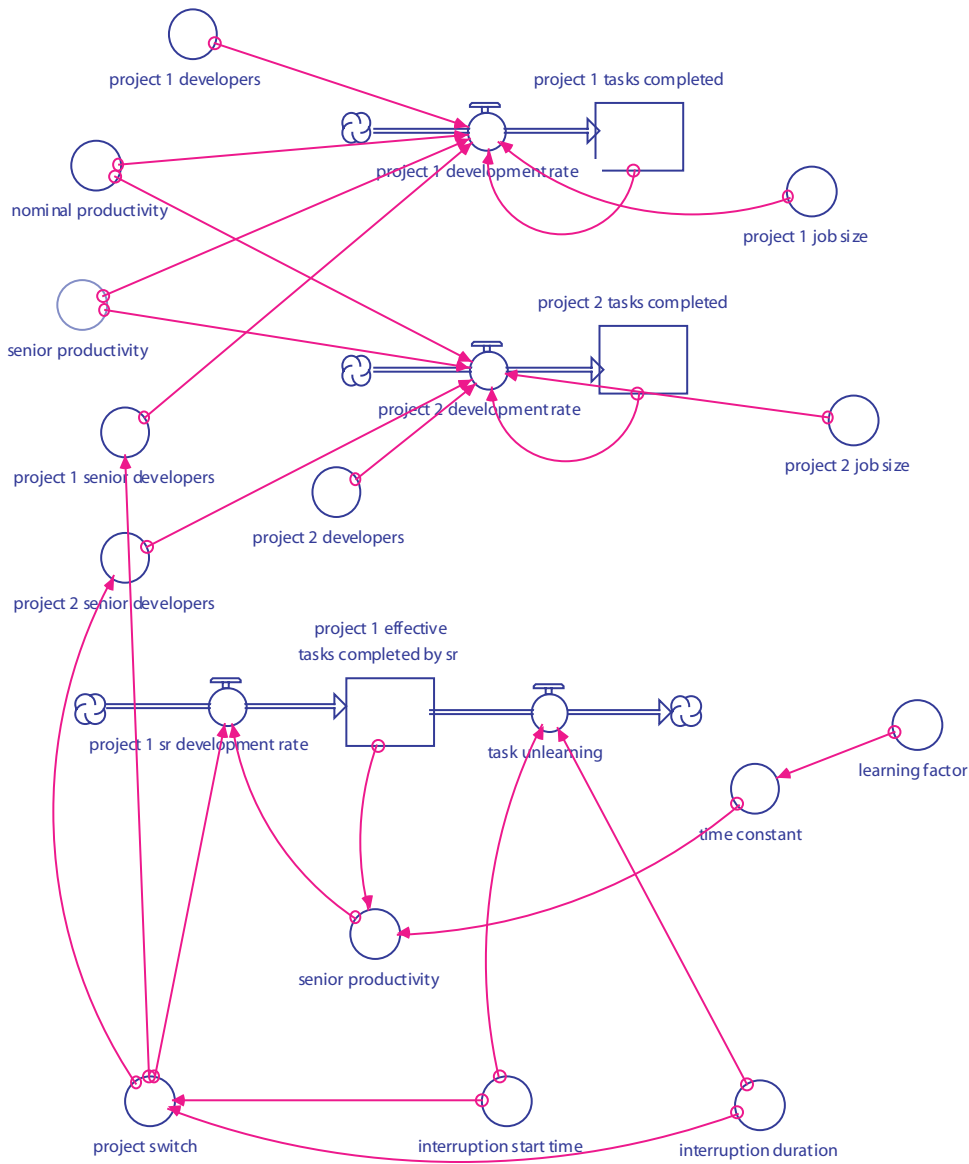


Figure 6.36. Project contention model diagram.

A Delphi poll and other queries were used to gauge the learning curve losses. It was estimated that 2–3 weeks are needed after a one-week break to get back into the productive mode of the original project. Added together, the respective losses and gains for the involved projects produced a net loss compared to not hot-switching personnel. Even more important, the anticipated schedule slips were directly correlated to cus-

tomter (dis)satisfaction. These results were shown to the executives who could not disagree with the conclusions and changed their corresponding policies. Figure 6.37 shows sample output. The provided model can be studied for the detailed formulations.

The effects of working on multiple tasks has also been studied by Weinberg [Weinberg 1992]. He developed a table of time spent on each task versus the number of tasks, reproduced in Table 6.9. The table partially validates the results found in this model. It indicates a good lesson for software organizations to keep in mind—that context switching is costly. As people move from one task to another, they stop current work, try to remember the new project context, and go back and forth similarly among different tasks.

The effect of these multiple interruptions over time seriously reduces people’s performance. To offset this, management should rank the competing tasks in priority and strive to have whole people on projects. The priorities might change over time, but people will generally be most productive for a given project if they can stay on that same project for as long as possible.

There is evidence, however, that team rotation can also have a positive effect on job satisfaction and commitment. Some people like to be continually challenged with new problems. Thus, the goals of innovation versus project stability sometimes need to be balanced.

6.6 STAFFING

Generating staffing plans is an important emphasis for project and organizational management. Cost estimation tools are frequently used to derive cost and schedule esti-

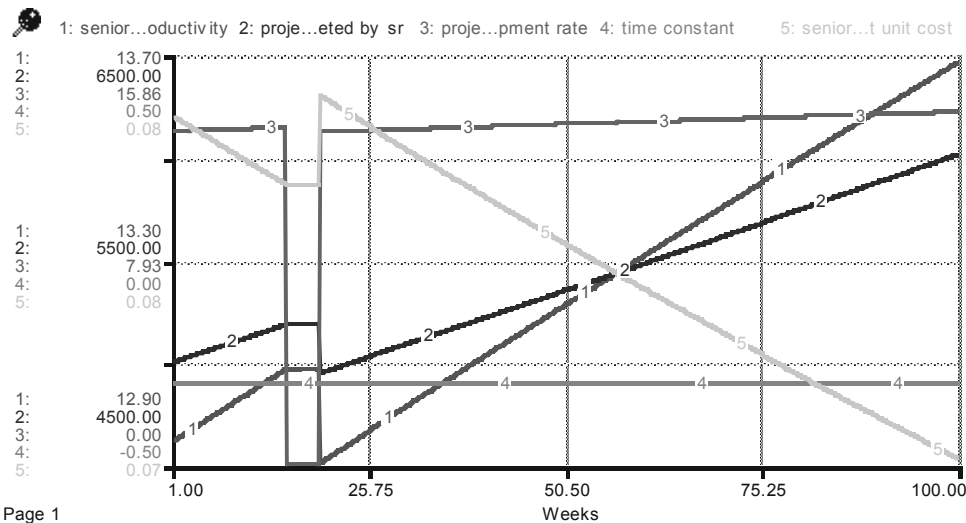


Figure 6.37. Project contention model results (1: senior productivity, 2: project 1 effective tasks completed by sr, 3: project 1 sr development rate, 4: time constant, 5: senior development unit cost).

Table 6.9. Productivity losses due to context switching from [Weinberg 1992]

Number of Tasks	Time on Each Task
1	100%
2	40%
3	20%
4	10%
5	5%
More than 5	random

mates, and various models come into play to derive staffing profiles over time. A simple method to derive step function staffing profiles from a static cost estimate is to calculate average personnel levels per phase by dividing effort by schedule for each time portion.

This section will cover different dynamic methods of estimating staffing needs. They are contained in this chapter instead of in Chapter 5 on people applications because neither method models personnel levels explicitly as state variables. The Rayleigh formulation uses a rate formula, which is interpreted as the staffing per time period (only cumulative effort is a state variable), and process concurrence models product characteristics (which can then be transformed into staffing needs).

Rayleigh curves have been traditionally used for many years to generate continuous staffing profiles over time based on project characteristics. They are smoothed curves that may reflect reasonable ramping up and ramping down, but frequently they prescribe unrealistic staffing situations for organizations.

It is important to have the right number of people on a project at precisely the right time, in order to not waste resources. In situations where a Rayleigh-like staffing curve fits best, like an unprecedented system, a constant level-of-effort profile is inefficient. It will waste people in the beginning and at the end, and will not have enough staff in the project middle, as shown in Figure 6.38.

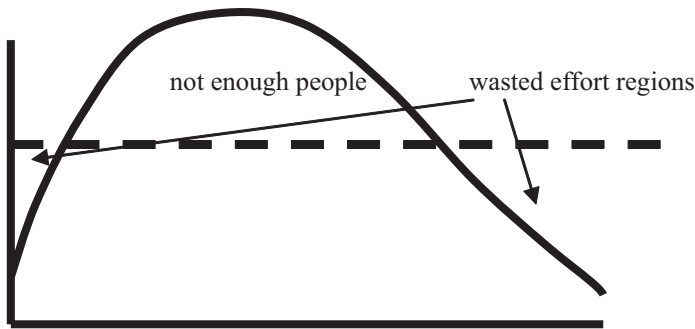


Figure 6.38. Ideal Rayleigh staffing versus level of effort.

Sometimes, a simple fixed staff size is reasonable for a project and/or it could also represent a real hiring constraint. A rectangular staffing pattern may be appropriate for situations including:

- Constrained staffing situations
- Well-known problems
- Mechanical translations
- Incremental development staffed-up portion

Modeling staffing needs is critical because otherwise many resources can be wasted and/or a project slowed down by not having resources at the right time. Modeling a fixed staff size is trivial, so the challenge is to model dynamic situations. Following are dynamic staffing models for Rayleigh curves and process concurrence approaches that can be adapted for many situations.

6.6.1 Example: Rayleigh Manpower Distribution Model

The Rayleigh curve (also called a Norden/Rayleigh curve) is a popular model of personnel loading that naturally lends itself to system dynamics modeling. It serves as a simple generator of time-based staffing curves that is easily parameterized to enable a variety of shapes. The Rayleigh curve is actually a special case of the Weibull distribution, and serves as a model for a number of phenomena in physics.

After analyzing hardware research and development projects, Norden put forth a manpower model based on Rayleigh curves. According to these staffing curves, only a small number of people are needed at the beginning of a project to carry out planning and specification. As the project progresses and more detailed work is required, the number of staff builds up to a peak. After implementation and testing, the number of staff required starts to fall until the product is delivered. Putnam subsequently applied the Rayleigh curve to software development [Putnam 1980], and it is used in the SLIM[®] and SEER[®] cost models among others.

One of the underlying assumptions is that the number of people working on a project is approximately proportional to the number of problems ready for solution at that time. Norden derived a Rayleigh curve that describes the rate of change of manpower effort per the following first-order differential equation:

$$\frac{dC(t)}{dt} = p(t)[K - C(t)]$$

where $C(t)$ is the cumulative effort at time t , K is the total effort, and $p(t)$ is a product learning function. The time derivative of $C(t)$ is the manpower rate of change, which represents the number of people involved in development at any time. Operationally, it is the staff size per time increment (traditionally, the monthly headcount or staffing profile).

The learning function is linear and can be represented by

$$p(t) = 2at$$

where a is a positive number. The a parameter is an important determinant of the peak personnel loading called the manpower buildup parameter. Note that the learning function actually represents increasing product elaboration (“learning” about the product), and is different from the people-skill learning curves discussed in Chapter 4. The distinction is discussed in the next section.

The second term $[K - C(t)]$ represents the current work gap; it is the difference between the final and current effort that closes over time as work is accomplished. The time convolution of the diminishing work gap and monotonic product learning function produces the familiar Rayleigh shape.

6.6.1.1 System Dynamics Implementation

The Rayleigh curve and much of Putnam’s work is naturally suited to dynamic modeling. Quantities are expressed as first- and second-order differential equations, precisely the rate language of system dynamics. Table 6.10 is a high-level summary of the model.

Figure 6.39 shows a very simple model of the Rayleigh curve using an effort flow chain. The formula for effort rate (the manpower rate per time period) represents the project staffing profile. It uses a feedback loop from the cumulative effort, whereby the feedback represents knowledge of completed work.

The equations for the model are

effort rate = learning function · work gap

learning function = manpower buildup parameter · time

work gap = estimated total effort – cumulative effort

Figure 6.40 shows the components of the Rayleigh formula from the simulation model where estimated total effort is 15 person-months and the manpower buildup parameter is set to 0.5. The learning function increases monotonically while the work gap (the difference between estimated and cumulative effort used for effort rate) diminishes over time as problems are worked out. The corresponding effort rate rises and falls in a Rayleigh shape. The two multiplicative terms, the learning function and the work gap, produce the Rayleigh effort curve when multiplied together. They offset each other because the learning function rises and the work gap falls over time as a project progresses.

Table 6.10. Rayleigh curve model overview

Purpose: Planning, Training			
Scope: Development Project			
Inputs and Parameters	Levels	Major Relationships	Outputs
• Start staff	• Effort	• Staffing rate	• Staffing
• Function points			• Effort
• Manpower buildup parameter			• Schedule

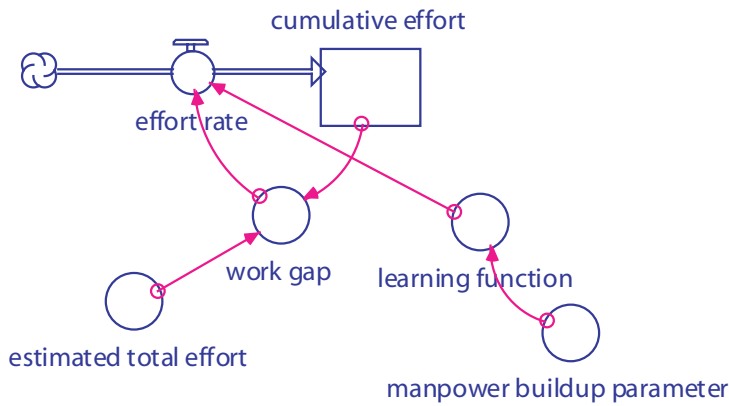


Figure 6.39. Rayleigh manpower model.

The Rayleigh model is an excellent example of a structure producing S-curve behavior for cumulative effort. The learning function and work gap combine to cause the Rayleigh effort curve, which integrated over time as cumulative effort is sigmoidal. The cumulative effort starts with a small slope (the learning function is near zero), increases in slope, and then levels out as the work gap nears zero.

The learning function is linear and really represents the continued elaboration of product detail (e.g., specification to design to code) or increasing understanding about

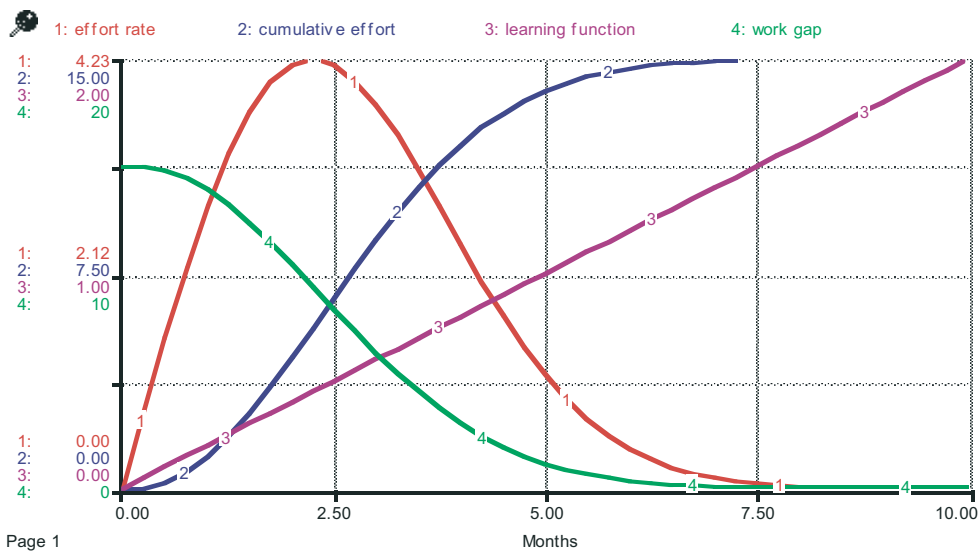


Figure 6.40. Rayleigh curve components.

the product makeup. A true learning curve has a different nonlinear shape (see Chapter 4). We prefer to call this an “elaboration function” to better represent the true phenomena being modeled. This is also consistent with the assumption that the staff size is proportional to the number of problems (or amount of detail) ready to be implemented. This is the difference between what has been specified (the elaboration function) and what is left to do (the work gap). Hence, we will use the term *elaboration function* in place of learning function from now on, except when the context calls for traditional Rayleigh curve terminology. See Section 6.6.3 for further relevance of this terminology.

Figure 6.41 shows the output staffing profile for different values of a . It is seen that the manpower buildup parameter a greatly affects the personnel loading curve. The larger the value of a , the earlier the peak time and the steeper the corresponding profile. For these reasons, a is also called the manpower buildup parameter (MBP). A large MPB denotes a responsive, nimble organization. The qualitative effects of the MBP are shown in Table 6.11.

It has been observed that the staffing buildup rate is largely invariant within a particular organization due to a variety of factors. Some organizations are much more responsive and nimble to changes than others. Design instability is the primary cause for a slow buildup. Top architects and senior designers must be available to produce a stable design. Hiring delays and the inability to release top people from other projects will constrain the buildup to a large extent. If there is concurrent hardware design in a system, instability and volatility of the hardware will limit the software design ready for programming solutions.

The Rayleigh curve is best suited to certain types of development projects like unprecedented systems. The Rayleigh curve can be calibrated to static cost models and

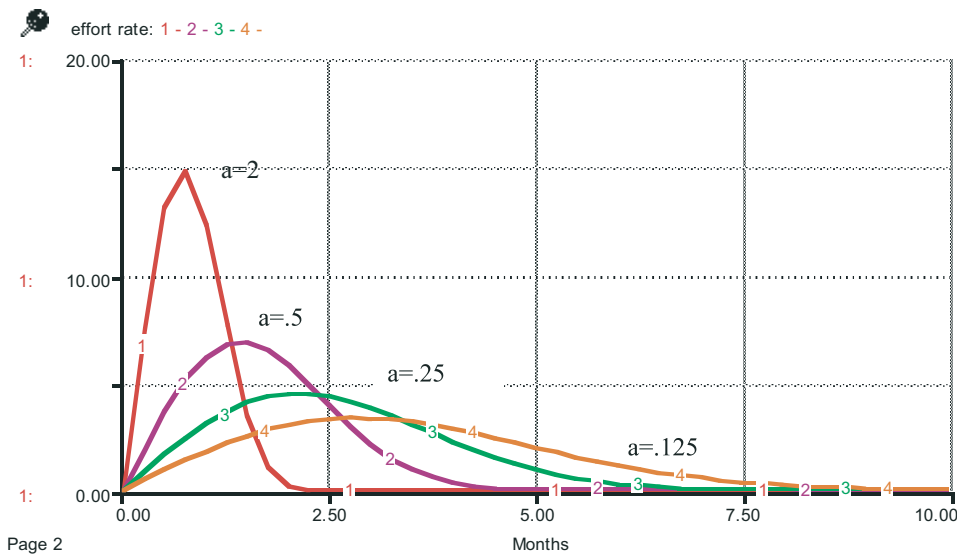


Figure 6.41. Rayleigh manpower model output for varying learning functions.

Table 6.11. Effects of manpower buildup parameter

Manpower Buildup Parameter	Effort Effect	Schedule Effect	Defect Effect
Low (slow staff buildup)	Lower ↓	Higher ↑	Lower ↓
Medium (moderate staff buildup)			
High (aggressive staff buildup)	Higher	Lower	Higher

used to derive dynamic staffing profiles. See Chapter 5 in [Boehm 1981] for examples or the model file *Rayleigh calibrated to COCOMO.itm* for an example.

6.6.1.2 Rayleigh Curve Versus Flat Staffing

In contrast to a Rayleigh curve buildup, a level-of-effort staffing may be possible for well-known and precedented problems that are ready for solution. An example would be a relatively simple porting between platforms of a software package with experienced developers. Since the problem has been solved by the people before, the task can be performed by an initial large staff. The project can be planned with a nearly linear trade-off between schedule and number of personnel (i.e., with a constant staff level, the porting will take about twice as long with one-half of the staff). Another example is an in-house organic project—many people can start the project compared to the slower Rayleigh buildup.

A high-peaked buildup curve and a constant level-of-effort staffing represent different types of software project staffing. Most software development falls in between the two behaviors. These two representative behaviors will be used as prototypical examples later when analyzing staffing curves.

6.6.1.3 Defect Modeling

The Rayleigh curve is also popular for modeling defect introductions and removals, as discussed in Chapter 5. Intuitively, the defect introduction curve should be very similar to the staffing curve, since the number of defects is expected to be proportional to the number of people working or the amount of work being produced.

6.6.1.4 Rayleigh Curve Enhancements

6.6.1.4.1 INCREMENTAL DEVELOPMENT. Some modeling approaches use a superposition of Rayleigh curves to represent parallel activities or phases. Figure 6.42 shows the output of an enhanced Rayleigh curve model for incremental development. Three increments are modeled with specified starting offsets, and the overall staffing shown is a superposition of the three increments. See the model file *rayleigh incremental.itm*.

6.6.1.4.2 DYNAMIC REQUIREMENTS CHANGES. Rather than assuming requirements specifications are fixed in the project beginning, it is more realistic to model changed requirements during the midst of a project. A simple model is provided that allows on-the-fly adjustments to the project size while the simulation is running.

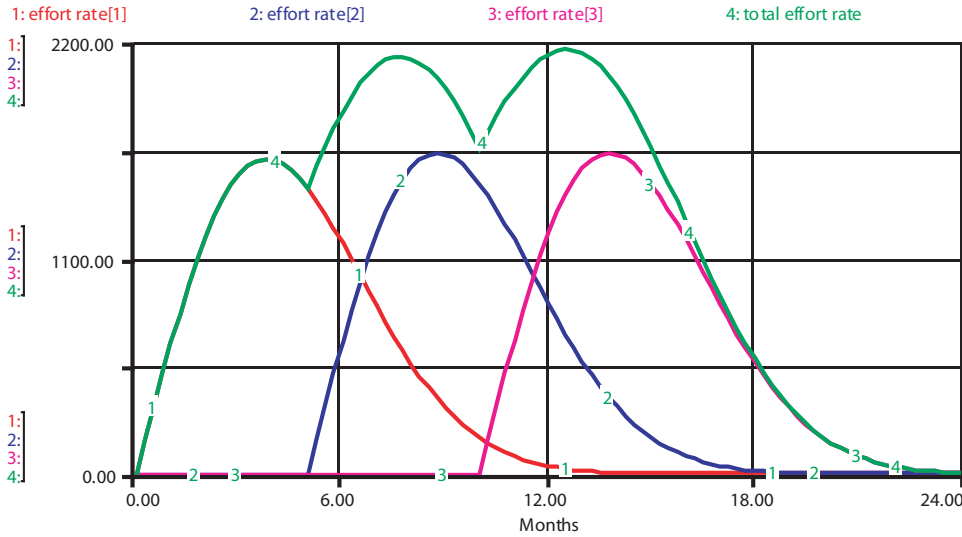


Figure 6.42. Rayleigh curves for incremental development.

The augmentation of the previous model in Figure 6.43 shows how the personnel rate is changed in response to additional requirements reflected in the project size. This mimics the effect of breakage or increased requirements. The slider is an interactive control that the user can adjust during a simulation run (purposely slowed to a human-controllable pace) to see the corresponding change in effort rate. Note that this represents an instantaneous decision regarding perceived additional effort. Only some projects can actually respond this quickly; thus, the modeler must be aware of these real-world constraints and account for them. Models can respond instantaneously, but actual processes may not. Metrics trends are more often analyzed at monthly intervals rather than every dt per the simulation, so there may be delays in getting additional staff.

6.6.2 Example: Process Concurrency Modeling

Process concurrency is the degree to which work becomes available based on work already accomplished, and can be used to derive staffing profiles. It describes interdependency constraints between tasks, both within and between project phases. Concurrency relationships are crucial to understanding process dynamics. Internal process concurrency refers to available work constraints within a phase, whereas external process concurrency is used to describe available work constraints between development phases. A good treatment of process concurrency for general industry can be found in [Ford, Sterman 1997], and this book interprets and extends the concepts for software engineering.

The availability of work described by process concurrency is a very important constraint on progress. Otherwise, a model driven solely by resources and productivity

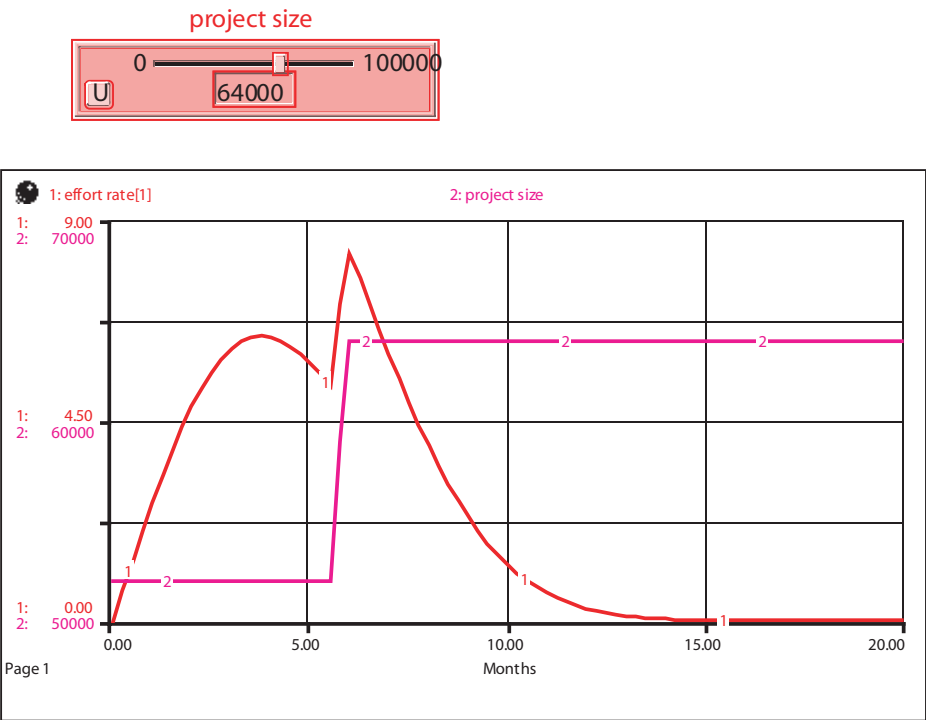


Figure 6.43. Interactive Rayleigh curve for requirements added midstream.

will allow a project to complete in almost zero time with infinite resources. Such is not the case with software processes in which tasks are highly interdependent, since some tasks must be sequential and cannot be done in parallel.

Process concurrence relationships describe how much work becomes available for completion based on previous work accomplished. These realistic bottlenecks on work availability should be considered during project planning and execution. There is a limit to the amount of concurrent development due to interdependencies in software processes. Concurrence relations can be sequential, parallel, partially concurrent, or other dependent relationships. Concurrence relationships can be elicited from process participants. A protocol for the elicitation is described in [Ford, Sterman 1998].

The definition of “task” in this context is an atomic unit of work that flows through a project, where the units may differ among project phases. This is the same treatment of tasks used in the Abdel-Hamid model and many others. Tasks are assumed to be interchangeable and uniform in size (e.g., the Abdel-Hamid task was equivalent to 60 lines of software). A task, therefore, refers to product specification during project definition, and lines of code during code implementation. The assumption becomes more valid as the size of the task decreases.

6.6.2.1 Trying to Accelerate Software Development

It is instructive to understand some of the phenomena that impede software processes. Putnam likens the acceleration of software development to pouring water into a channel-restricted funnel [Putnam 1980]. The funnel does not allow the flow to be sped up very much, no matter how much one pours into the funnel. This is like throwing a lot of software personnel at once into the development chute to accelerate things. They will not be able to work independently in parallel, since certain tasks can only be done in sequence. Figure 6.44 shows the limited parallelism of software tasks using a funnel analogy alongside the corresponding system dynamics structure.

There are always sequential constraints independent of phase. The elemental activities in any phase of software development include:

- Analysis and requirements specification; figuring out what you are supposed to do and specifying how the parts fit together
- Development of some artifact type (architecture, design, code, test plan, etc.) that implements the specifications
- Assessment of what was developed; this may include verification, validation, review, or debugging
- Possible rework or recycle of previous activities

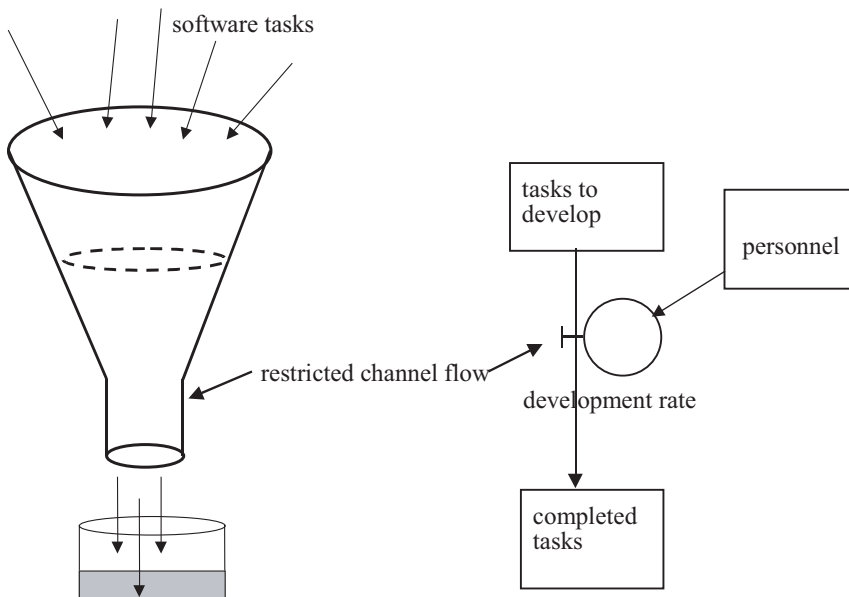


Figure 6.44. Funnel view of limited task parallelism and system dynamics corollary (partially adapted from [Putnam 80]).

These activities cannot be done totally in parallel with more applied people. Different people can perform the different activities with limited parallelism, but downstream activities will always have to follow some of the upstream ones.

The parallel constriction concept is further elaborated in Figure 6.45, which shows the constriction brought about by trying to parallelize sequential (or at least partially sequential) activities for a single thread of software. Tasks can only flow through in proper order.

In *The Mythical Man-Month* [Brooks 1975, 1995], Brooks explains these restrictions from a partitioning perspective in his Brooks’s Law framework. Sequential constraints imply that tasks cannot be partitioned among different personnel resources. Thus, applying more people has no effect on schedule. Men and months are interchangeable only when tasks can be partitioned with no communication among them. Process concurrence is a natural vehicle for modeling these software process constraints.

6.6.2.2 Internal Process Concurrence

An internal process concurrence relationship shows how much work can be done based on the percent of work already done. The relationships represent the degree of sequentiality or concurrence of the tasks aggregated within a phase. They may include changes in the degree of concurrence as work progresses. Figure 6.46 and Figure 6.47 demonstrate linear and nonlinear internal process concurrence. The bottom right half under the diagonal of the internal process concurrence is an infeasible region, since the percent available to complete cannot be less than the amount already completed.

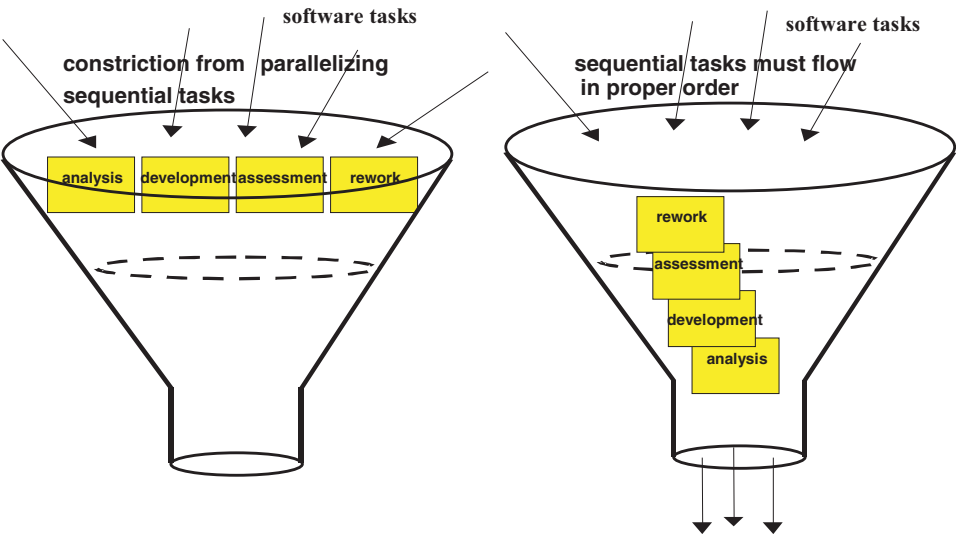


Figure 6.45. Trying to parallelize sequential tasks in the funnel.

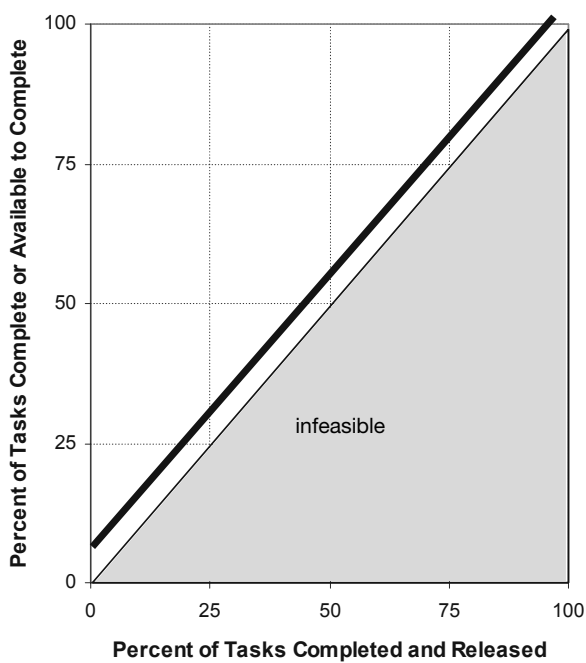


Figure 6.46. Linear internal process concurrence.

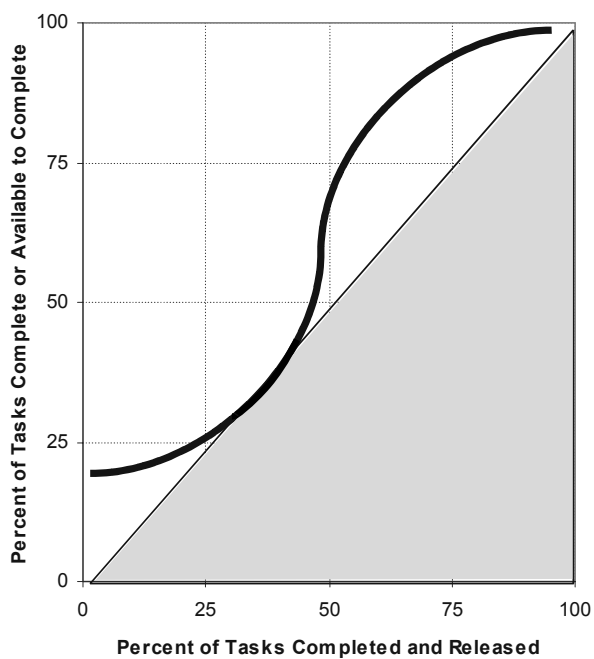


Figure 6.47. Nonlinear internal process concurrence.

The development of a single software task within a phase normally includes the sequences of construction, verification/validation, and, sometimes, rework. These cannot all be simultaneously performed in an individual task. There will always be an average duration of the subactivities, regardless of the resources applied to them. Additionally, the development of some tasks requires the completion of other intraphase tasks beforehand. Thus, the process limits the availability of work to be completed based on the amount of already completed work.

More concurrent processes are described by curves near the left axis, and less concurrent processes lie near the 45° line. The linear relationship in Figure 6.46 could describe the sequential construction of a 10 story building. When the first floor is complete, 10% of the project is done and the second floor is available to be completed, or 20% of the entire project is thus available to finish. This pattern continues until all the floors are done. This is sometimes called a “lockstep” relationship. The linear concurrence line starts above the 45° diagonal, since the y-axis includes work “available to complete.” The relationship has to start greater than zero. Consider again a skyscraper being built. At the very beginning, the first floor is available to be completed.

A more typical nonlinear relationship is shown in Figure 6.47. For example, the overarching segments of software must be completed before other parts can begin. Only the important portions (such as 20% of the whole for an architecture skeleton, interface definitions, common data, etc.) can be worked on in the beginning. The other parts are not available for completion until afterward. This typifies complex software development tasks in which many tasks are dependent on each other. Men and months are not interchangeable in this situation according to Brooks because the tasks cannot be partitioned without communication between them.

Figure 6.48 shows internal concurrence for an extreme case of parallel work. There is very high concurrency because the tasks are independent of each other. Almost everything can be doled out as separate tasks in the beginning, such as a straightforward translation of an existing application from one language to another. Each person simply gets an individual portion of code to convert, and that work can be done in parallel. The last few percent of tasks for integrating all translated components have to wait until the different pieces are there first, so 100% cannot be completed until the translated pieces have been completed and released. Brooks explains that many tasks in this situation can be partitioned with no communication between them, and men and months are largely interchangeable.

6.6.2.3 External Process Concurrence

External process concurrence relationships describe constraints on amount of work that can be done in a downstream phase based on the percent of work released by an upstream phase. Examples of several external process concurrence relationships are shown in Table 6.12. More concurrent processes have curves near the upper-left axes, and less concurrent processes have curves near the lower and right axes.

The partially concurrent interphase relationship is representative of much software development in which complexities impose task dependencies and, thus, intertask communication is necessary. For example, a critical mass of core requirements must

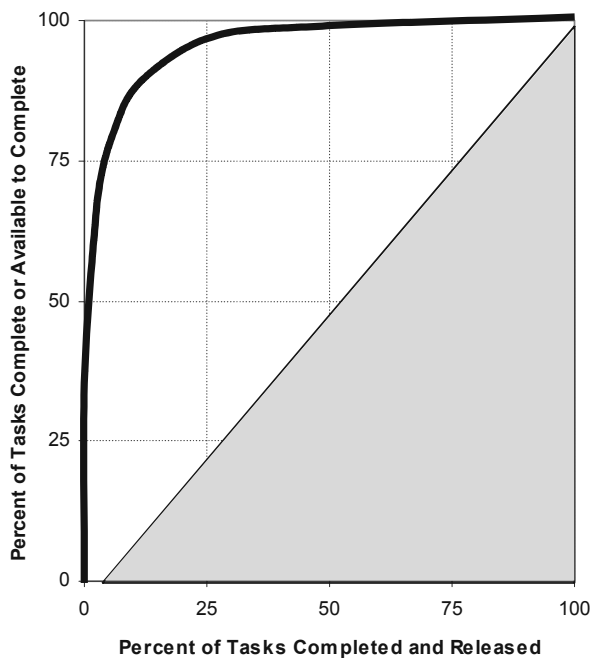
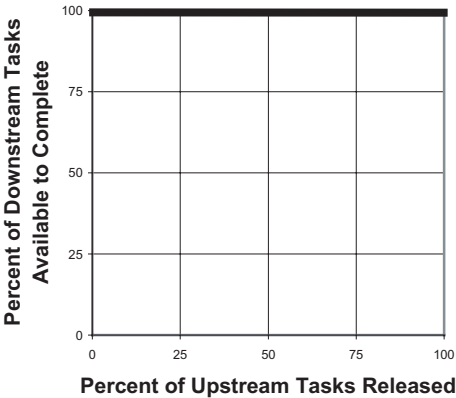


Figure 6.48. Nearly parallel internal process concurrence.

Table 6.12. External process concurrence relationships

Relationship	Characteristics
No Interphase Relationship	<ul style="list-style-type: none">• No dependencies between the phases.• The downstream phase can progress independently of the upstream phase.• The entire downstream work is available to be completed with none of the upstream work released.



(continued)

Table 6.12. *Continued*

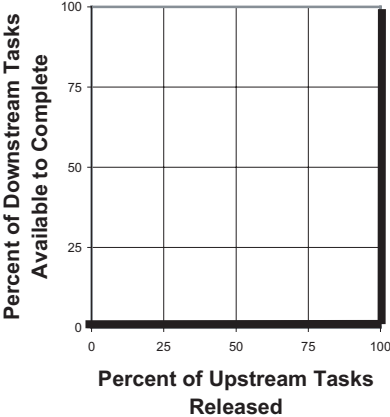
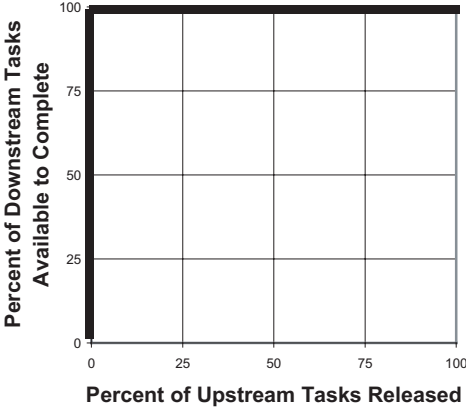
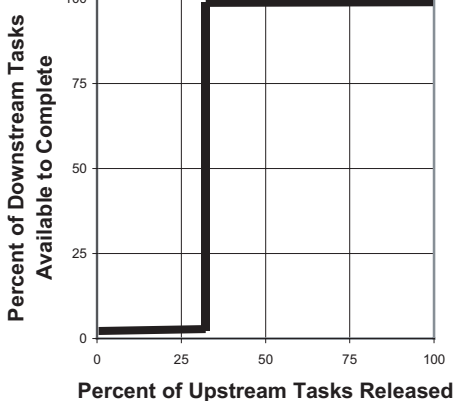
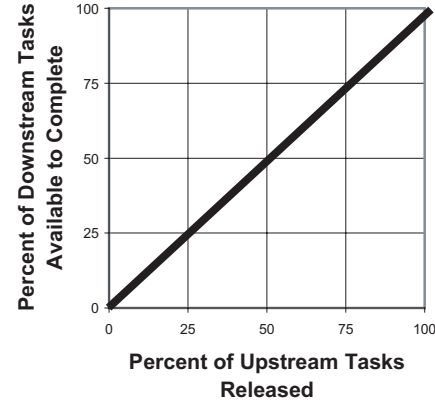
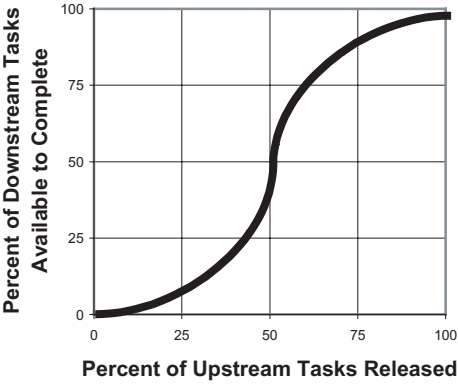
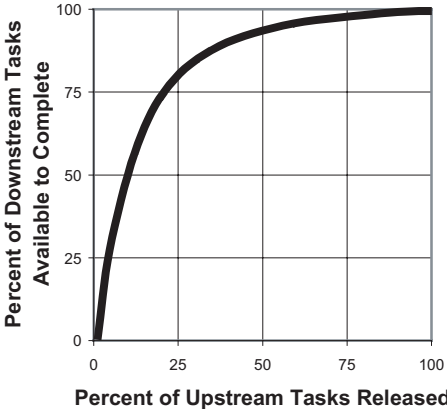
Relationship	Characteristics
<p>Sequential Interphase Relationship</p> 	<ul style="list-style-type: none">• None of the downstream phase can occur until the upstream phase is totally complete.• Like a theoretical waterfall development process in which no phase can start until the previous phase is completed and verified.• Same as a finish–stop relationship in a critical path network.
<p>Parallel Interphase Relationship</p> 	<ul style="list-style-type: none">• The two phases can be implemented completely in parallel.• The downstream phase can be completed as soon as the upstream phase is started.
<p>Delayed Start Interphase Relationship</p> 	<ul style="list-style-type: none">• The downstream phase must wait until a major portion of the upstream phase is completed, then it can be completed in its entirety.• Like a start–start relationship in a critical path network with a delay between the start times.

Table 6.12. *Continued*

Relationship	Characteristics
<p>Lockstep Relationship</p>  <p>The graph shows a direct linear relationship between the percentage of upstream tasks released and the percentage of downstream tasks available to complete. The line starts at the origin (0,0) and extends to the top-right corner (100,100) of the plot area.</p>	<ul style="list-style-type: none"> • The downstream phase can progress at the same speed as the upstream phase; thus, they are in lockstep with each other. • The downstream work availability is correlated linearly 1:1 to how much is released from upstream. For example, after 10% of system design is completed, then 10% of implementation tasks is available to finish. • This relationship is not available in PERT/CPM.
<p>Delay with Partially Concurrent Interphase Relationship</p>  <p>The graph shows an S-shaped curve. It starts at (0,0), remains near zero until about 25% of upstream tasks are released, then rises steeply through the middle of the graph, passing through (50,50), and finally levels off as it approaches 100% of downstream tasks available as upstream tasks reach 100%.</p>	<ul style="list-style-type: none"> • The downstream phase has to wait until a certain percentage of upstream tasks have been released, and then can proceed at varying degrees of concurrence per the graph. • This relationship is representative of complex software development with task interdependencies. • This type of relationship is not available with PERT/CPM methods.
<p>Leveraged Concurrence Relationship</p>  <p>The graph shows a curve that rises very sharply from the origin (0,0). It reaches about 75% of downstream tasks available with only 25% of upstream tasks released, and then continues to rise more gradually, approaching 100% of downstream tasks as upstream tasks reach 100%.</p>	<ul style="list-style-type: none"> • This relationship exhibits a high degree of parallelism and leverage between phases. • Typical of Commercial Off-the-Shelf (COTS) products whereby one specifies the capabilities, and the system is quickly configured and instantiated. Also applicable for fourth-generation language (4GL) approaches.

be released before architecture and design can start. Then the downstream phase availability rate increases in the middle region (e.g., much design work can take place), and then slows down as the final upstream tasks are released.

External process concurrence relationships function like the precedence relationships in critical path and PERT methods to describe dependencies, but contain greater dynamic detail. For example, external concurrence relationships describe the phase dependencies for the entire durations. PERT and critical path methods only use the stop and start dates. They can also be nonlinear to show differences in the degree of concurrence, whereas PERT methods cannot. Lastly, process concurrence relationships are dynamic since the work completed could increase or decrease over time, but only static precedence relationships are used in critical path or PERT methods. The lockstep and delay of partially concurrent interphase relationships are situations that cannot be described with PERT or critical path methods.

6.6.2.4 Phase Leverage Analysis with Process Concurrence

The leverage in software development that we will examine refers to how much can be elaborated based on inputs from the previous phase. For example, a 4GL approach that generates code from requirement statements has much higher leverage than new code development. For about the same amount of effort expended on requirements, a great deal more demonstrative software will be produced with advanced 4GL tools compared to starting from scratch in each phase. Virtually all modern approaches to software development are some attempt to increase leverage, so that machines or humans can efficiently instantiate software artifacts compared to more labor-intensive approaches.

COTS is another good example of high phase leverage, because functionality is easily created after identifying the COTS package. If one specifies “use the existing XYZ package for library operations,” then the functions of online searching, checkout, and so on are already defined and easily implemented after configuring the package locally.

Process concurrence is a very useful means of contrasting the leverage of different approaches, because the degree of concurrence is a function of the software methodology (among other things). When developing or interpreting external process concurrence curves for software development strategies, it is helpful to think of tasks in terms of demonstrable functionality. Thus, tasks released or available to complete can be considered analogous to function points in their phase-native form (e.g., design or code). With this in mind, it is easier to compare different strategies in terms of leverage in bringing functionality to bear.

6.6.2.4.1 RAD EXAMPLE OF EXTERNAL PROCESS CONCURRENCE. Increasing task parallelism is a primary opportunity to decrease cycle time in RAD. Process concurrence is ideally suited for evaluating RAD strategies in terms of work parallelism constraints between and within phases. System dynamics is very attractive for analyzing schedule time in this context compared to other methods because it can model task interdependencies on the critical path. Only those tasks on the critical path have influence on the overall schedule.

One way to achieve RAD is by having base software architectures tuned to application domains available for instantiation, standard database connectors, and reuse. This example demonstrates how the strategy of having predefined and configurable architectures for a problem domain can increase the chance for concurrent development between inception and elaboration.

6.6.2.4.1.1 Developing from Scratch. Suppose the software job is to develop a human resources (HR) self-service portal. It is to be a Web-based system for employees in a large organization to access and update all their personnel and benefits information. It will have to tie into existing legacy databases and commercial software packages for different portions of the human resources records. The final system will consist of the following:

- 30% user interface (UI) front end
- 30% architecture and core processing logic
- 40% database (DB) wrappers and vendor package interface logic (back-end processing)

Table 6.13 describes an example concurrence relationship between inception and elaboration for this system, where inception is defining the system capabilities and elaboration is designing it for eventual construction. The overall percent of tasks ready to elaborate is a weighted average. There is no base architecture from which to start from. Figure 6.49 shows a plot of the resulting external concurrence relationship from this worksheet.

6.6.2.4.1.2 Developing With a Base Architecture. Contrast the previous example with another situation in which there exists a base architecture that has already been tuned for the human resources application domain of processes and employee service workflows. It uses XML technology that ties to existing database formats and is ready for early elaboration by configuring the architecture. It already has standard connectors for different vendor database packages.

Figure 6.50 shows the corresponding process concurrence against the first example in which an architecture had yet to be developed. This relationship enables more parallelism between the inception and elaboration phases, and, thus, the possibility of reduced cycle time. When about 60% of inception tasks are released, this approach allows 50% more elaboration tasks to be completed compared to the development from scratch.

6.6.2.4.2 SYSTEMS ENGINEERING STAFFING CONSIDERATIONS TO MINIMIZE SCHEDULE. Staffing profiles are often dictated by the staff available, but careful consideration of the people dedicated to particular activities can have a major impact on overall schedule when there are no constraints. Consider the problem early in a project when developers might be “spinning their wheels” and wasting time while the requirements are being derived for them. Traditionally, the requirements come from someone with a systems engineering focus (though it is generally best when other disciplines also par-

Table 6.13. Concurrency worksheet for developing HR portal from scratch

Inception (System Definition)	Elaboration (System Design)		
	% of Inception Tasks Released	% of Components Ready to Elaborate	Overall % of Tasks Ready to Elaborate
Requirements Released			
About 25% of the core functionality for the self-service interface supported by prototype. Only general database interface goals defined.	30%	20% UI 10% core 5% DB	11%
About half of the basic functionality for the self-service interface supported by prototype.	55%	40% UI 20% core 20% DB	26%
Interface specifications to commercial package defined for internal personnel information.	60%	40% UI 30% core 40% DB	37%
More functionality for benefits capabilities defined (80% of total front end).	75%	75% UI 60% core 40% DB	57%
Interface specification to commercial systems for life insurance and retirement information.	85%	75% UI 80% core 80% DB	79%
Rest of user interface defined (95% of total), except for final UI refinements after more prototype testing.	95%	95% UI 95% core 80% DB	89%
Time card interface to accounting package defined.	98%	95% UI 95% core 100% DB	97%
Last of UI refinements released.	100%	100% UI 100% core 100% DB	100%

ticipate). If those people are not producing requirements at a rate fast enough for other people to start elaborating on, then effort is wasted. The staffing plan should account for this early lack of elaboration, so that implementers are phased in at just the right times when tasks become available to complete.

Knowledge of process concurrence for a given project can be used to carefully optimize the project this way. The right requirements analysts (normally a small number) have to be in place producing specifications before substantially more programmers come in to implement the specifications. This optimizing choreography requires fine coordination to have the right people available at the right time.

To optimize schedule on a complex project with partial interphase concurrency, the optimal systems engineering staffing is front-loaded as opposed to constant level of ef-

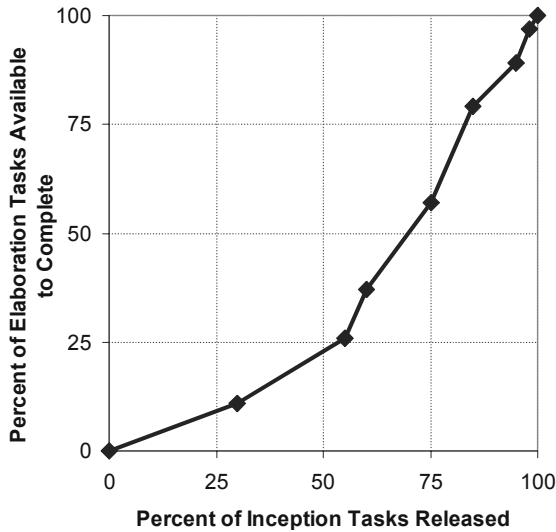


Figure 6.49. External process concurrence for HR system from scratch.

fort. As shown with process concurrence, the downstream development is constrained by the specifications available. Figure 6.51 shows two situations of awareness to achieve rapid application development (RAD) when there is partial interphase concurrency. The cases show what happens with (1) a constant staff size and (2) a peaked staffing profile for requirements generation. In the first case, there is a nonoptimal con-

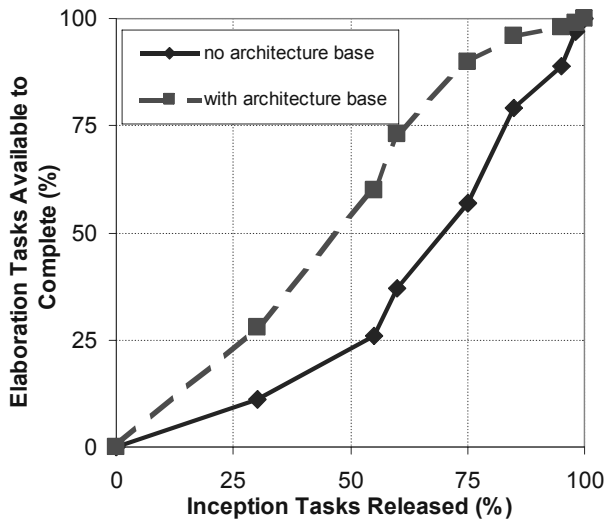


Figure 6.50. External process concurrence comparison with base architecture.

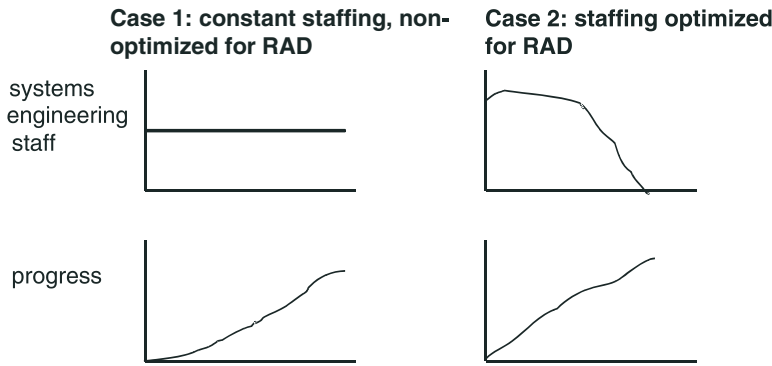


Figure 6.51. Concurrence awareness for systems engineering staffing to achieve RAD.

stant staff level for systems engineering and overall progress is impeded. If a curvilinear shape is used instead to match the software development resources, then cycle-time gains are possible because programming can complete faster.

The figure denotes systems engineering staff, but the staff may be anyone specifying requirements with other titles (they may also be software engineers themselves). See the chapter exercise for testing the hypothesis in this section that to optimize schedule on a complex project with partial interphase concurrency, the optimal systems engineering staffing is front-loaded as opposed to constant level of effort.

6.6.2.4.3 EXTERNAL CONCURRENCE MODEL AND EXPERIMENTATION. A simple model of external process concurrence is shown in Figure 6.52, representing task specification and elaboration. This model is provided in the file *external.concurrence.itm* and is used for the subsequent examples in this section. It models concurrence dynam-

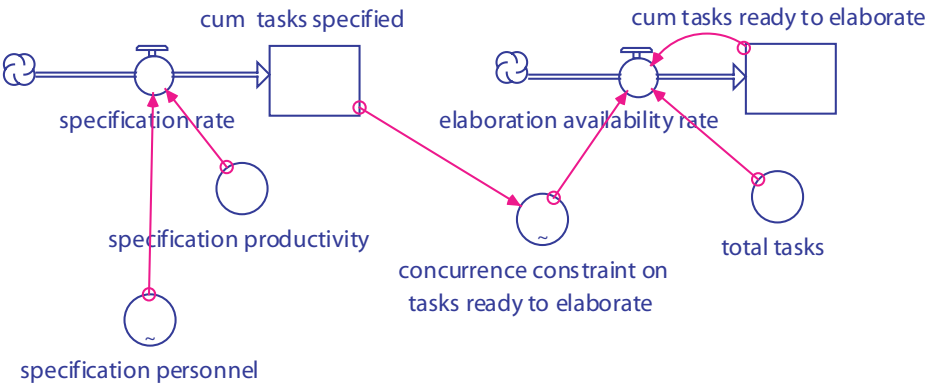


Figure 6.52. External concurrence model.

ics in the elaboration phase of a typical project, and will be used to experimentally derive staffing profiles for different combinations of specification inputs and concurrence types. Table 6.14 is a high-level summary of the model.

The specification personnel parameter is a forcing function to the model, and is a graphical staffing profile that can take on various shapes. The specification input profile mimics how requirements are brought dynamically into a project. The concurrence constraint is also a graphical function drawn to represent different process concurrence relationships to be studied.

The time profile of tasks ready to elaborate will be considered proportional to an “ideal” staffing curve. This follows from an important assumption used in the Rayleigh curve that an optimal staffing is proportional to the number of problems ready for solution. It is very important to note that this model only considers the product view. The real-world process of finding and bringing people on board may not be able to keep up with the hypothetical optimal curve. Thus, the personnel perspective may trump the product one. Much experience in the field points out that a highly peaked Rayleigh curve is often too aggressive to staff to.

The model is used to experimentally derive optimal elaboration staffing profiles (from a product perspective) for different types of projects. We will explore various combinations of specification profiles and concurrence between specification and elaboration to see their effect. The staffing inputs include (1) flat staffing, (2) a peaked Rayleigh-like staffing, and (3) a COTS requirements pulse at the beginning followed by a smaller Rayleigh curve to mimic a combined COTS and new development. In addition, we will vary the concurrence types as such: (1) linear lockstep concurrence, (2) a slow design buildup between phases, (3) leveraged instantiation to model COTS, and (4) S-shaped concurrence that models a wide swath of development. COTS is a pulse-like input because the requirements are nearly instantly defined by its existing capabilities.

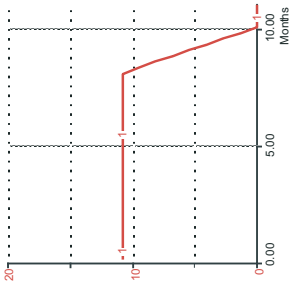
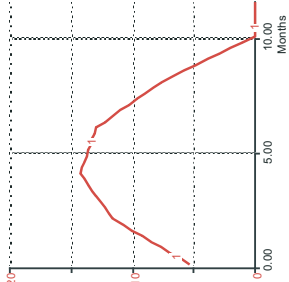
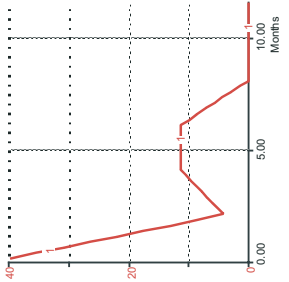
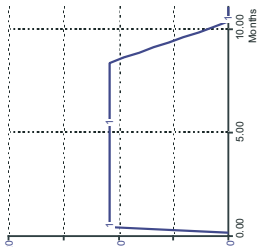
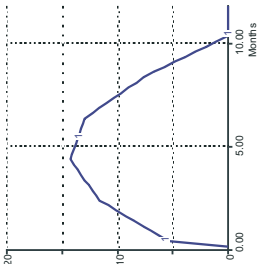
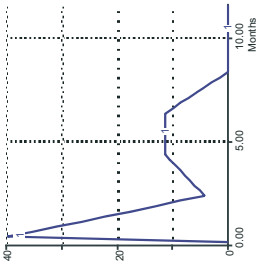
Table 6.15 shows outputs of the external concurrence model for a variety of situations. It is clear that optimal staffing profiles can be far different from a standard Rayleigh curve, though some of the combinations describe projects that can be modeled with a Rayleigh curve.

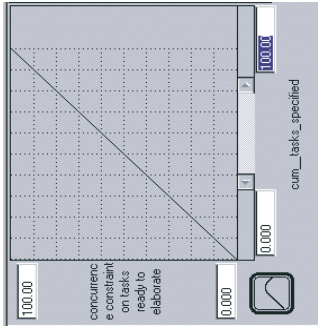
To further use these results for planning a real project, the ideal staffing curves have been modulated by relative effort. That is, implementing COTS or reused components generally require much less effort than new development. If a reused component takes

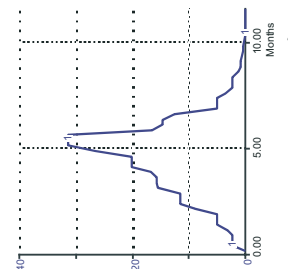
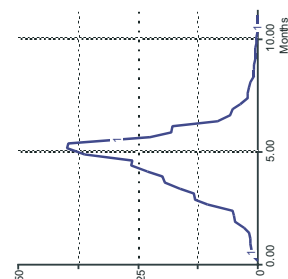
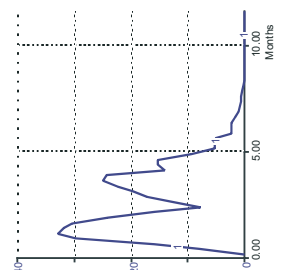
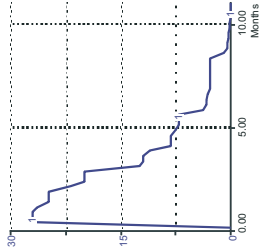
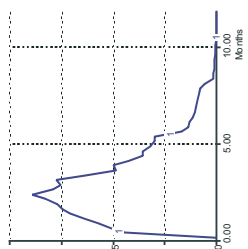
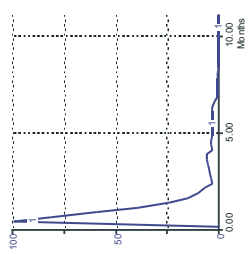
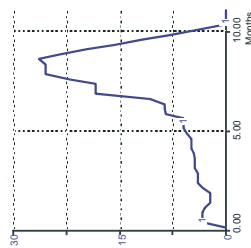
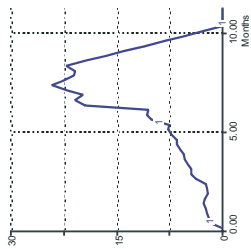
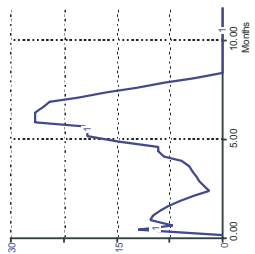
Table 6.14. External process concurrence model overview

Purpose: Planning, Process Improvement			
Scope: Portion of Lifecycle			
Inputs and Parameters	Levels	Major Relationships	Outputs
<ul style="list-style-type: none">• Specification personnel• Specification productivity• Total tasks• Concurrence constraint relationship	<ul style="list-style-type: none">• Cumulative tasks specified• Cumulative tasks ready to elaborate	<ul style="list-style-type: none">• Elaboration availability rate	<ul style="list-style-type: none">• Tasks ready to elaborate

Table 6.15. Elaboration availability simulation results

Concurrency Type	Specifications Input Profile			
	Flat Staffing	Rayleigh	COTS pulse at front	
Linear concurrency				
				





20% of the effort compared to a new one, then the required staffing should be similarly reduced. So the staffing curves need further adjustments for the relative effort of different approaches.

6.6.2.4.4 ADDITIONAL CONSIDERATIONS. Process concurrence curves can be more precisely matched to the software system types. For example, COTS, by definition, should exhibit very high concurrence but an overall system can be a blend of approaches. Mixed strategies produce combined concurrence relationships. Concurrence for COTS first then new development would be similar to that seen in Figure 6.53. The curve starts out with a leveraged instantiation shape, then transitions to a slow design buildup. This type of concurrence would be better matched to the system developed with an initial COTS pulse followed by new software development, corresponding to the last column in Table 6.15. Many permutations of concurrence are possible for various actual situations.

6.6.2.5 *Process Concurrence Summary*

Process concurrence provides a robust modeling framework. It can model more realistic situations than the Rayleigh curve and produce varying dynamic profiles (see the next section on integrating the perspectives). It can be used as a method to characterize different approaches in terms of their ability to parallelize or accelerate activities. It gives a detailed view of project dynamics and is relevant for planning and improvement purposes. It can be used as a means to collaborate between stakeholders to achieve a shared planning vision as well as to derive optimal staffing profiles for different project situations. However, more empirical data is needed on concurrence relationships from the field for a variety of projects.

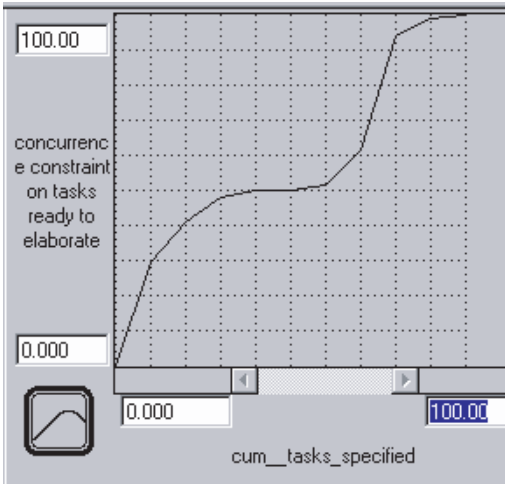


Figure 6.53. Sample process concurrence for mixed strategies.

6.6.3 Integrating Rayleigh Curves, Process Concurrence, and Brooks's Interpretations

There are connections between process concurrence and Rayleigh-curve modeling that are useful for understanding the dynamics and collectively provide a more robust framework for modeling processes. In fact, process concurrence can be used to show when and why the Rayleigh curve does not apply. Process concurrence provides a way to model the constraints on making work available in and between phases. The work available to perform is the same dynamic that drives the Rayleigh curve, since the staff level is proportional to the problems (or specifications) currently available to implement. S-curves result for expended effort over time (the accumulation of the staffing curve) or cumulative progress when a Rayleigh staffing shape applies.

However the Rayleigh curve was based on the initial study of hardware research and development projects that most resemble a traditional waterfall life cycle for unprecedented software systems. Now there are a great variety of situations that it does not match so well. Rayleigh staffing assumptions do not hold well for COTS, reuse, architecture-first design patterns, fourth-generation languages, or staff-constrained situations.

The underlying assumption that an “ideal” staffing curve is proportional to the number of problems ready for solution (from a product perspective only) still seems prudent. With modern methods, the dynamic profile of problems ready for solution can have far different shapes than was observed when the Rayleigh curve was first applied to software. Experimentation with the external concurrence model in Section 6.6.2.4.3 showed examples of this.

Iterative processes with frequent cycles or incremental development projects generally have flatter staffing profiles. Some would argue the flat profiles are the superposition of many sub-Rayleigh curves but, nevertheless, the initial assumptions were based on sequential, one-pass projects.

Other situations in which the Rayleigh curve does not apply too well are highly precededented systems for which early parallel work can take place and the project ends with a relatively flat profile, such as a heavy reuse or simple translation project, or any situation in which a gradual buildup is not necessary. Process concurrence can produce any number of dynamic profiles, and can thus be used to model more situations than the Rayleigh curve.

Schedule-driven projects which implement timeboxing are another example of projects that can have more uniform staffing distributions. These projects are sometimes called Schedule As the Independent Variable (SAIV) projects since cost and quality float relative to the fixed schedule. On such projects, there is no staff tapering at the end, because the schedule goal is attained by keeping everyone busy until the very end. Thus, the staffing level remains nearly constant.

We now have alternative methods of modeling the staffing curve. A standard Rayleigh formula can be used or process concurrence can replace the elaboration function in it. The first term in the Rayleigh equation that we call the elaboration function, $p(t)$, represents the cumulative specifications available to be implemented. The cumulative level of specifications available is the output of a process concurrence relation-

ship that operates over time. Thus, we can substitute a process concurrence relationship in place of the elaboration function, and it will be a more general model for software processes since the Rayleigh curve does not adequately model all development classes. Process concurrence provides a more detailed view of the dynamics and is meaningful for planning and improvement purposes.

Recall the Rayleigh staffing profile results from the multiplication over time of the elaboration function and the remaining work left. The elaboration function increases and the remaining work gap decreases as work is performed. Figure 6.54 shows the idealized Rayleigh staffing components and the process concurrence analogy for the product elaboration function. Internal and external concurrence relationships that can replace the Rayleigh formula for this are in Figure 6.55.

Table 6.16 summarizes the process dynamics of tasks and personnel on prototypical projects per different modeling viewpoints that have been presented. The Rayleigh manpower buildup parameter a depends on the organizational environment and project type, so the relative differences in the table only address the effect of project type on staff buildup limits. A more precise assessment of the buildup parameter for a specific situation would take more into account.

6.7 EARNED VALUE

Earned value is a useful approach for tracking performance against plans and controlling a project. It provides important management indicators but does not have dynamics intrinsic to itself. This section, therefore, focuses on using an earned value simulation model for training and learning. Earned value helps you become a “real-time process controller” by tracking and reacting to the project trends. Monitoring cost and schedule performance against plans helps spot potential schedule slippages and cost overruns early in order to minimize their impact. Earned value refers to the monetary value of work accomplished, where the value of a given task is defined by a

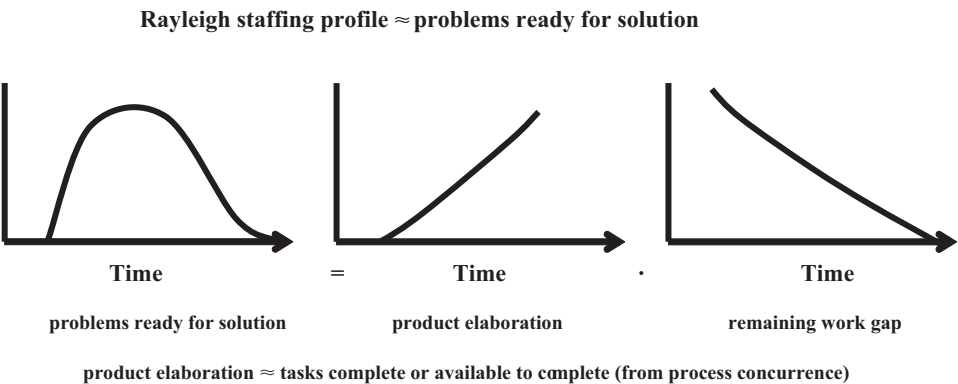


Figure 6.54. Rayleigh curve components and process concurrence analogy.

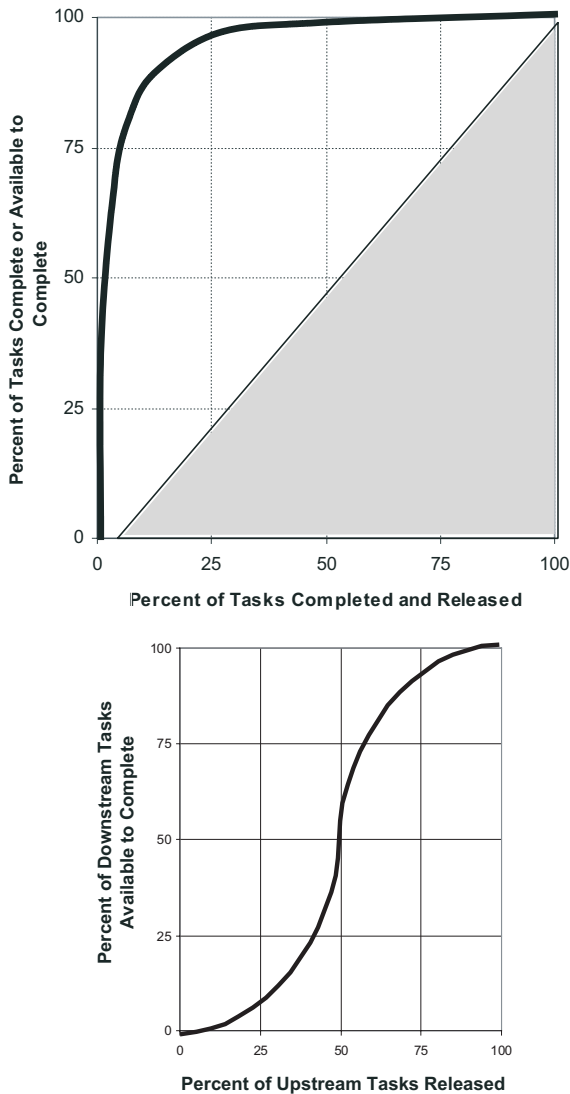
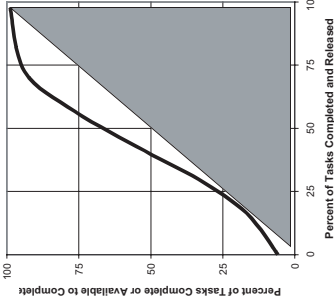
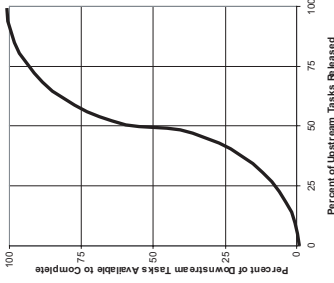
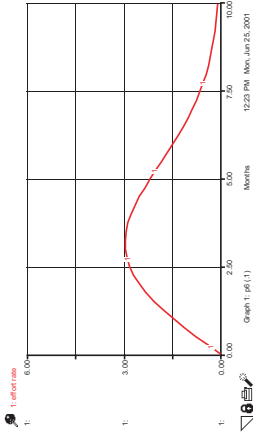


Figure 6.55. Process concurrence replacement for Rayleigh profile.

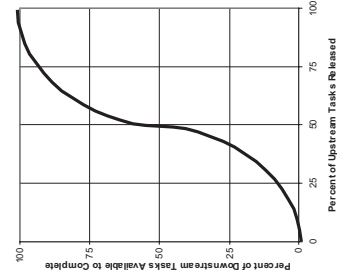
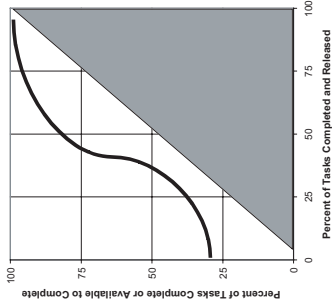
cost/schedule budget baseline. Earned value can also be implemented using straight labor hours rather than costs.

All work is planned, budgeted, and scheduled in time-phased increments constituting a cost and schedule measurement baseline. Value is earned after completion of budgeted milestones as work proceeds. Objective milestones consist of directly observable steps or events in the software process, and earned value is, therefore, a measure of progress against plan.

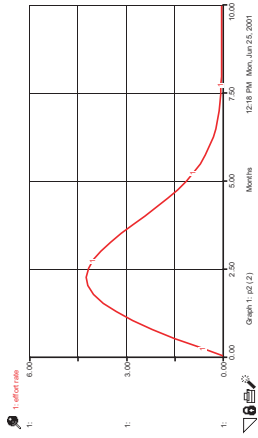
Table 6.16. Task and effort dynamics summary for major project types

Project Type	Process Concurrence		Brooks's Interpretation
	Internal Concurrence	External Concurrence (inception to elaboration only) ²	
New development of unprecedented system			Rayleigh Curve Modeling! (a = manpower buildup parameter)
		a = small.	Interdependent tasks impose sequentiality, so many tasks cannot be partitioned for parallel development. Men and months are not interchangeable.
			
Initial architecture development retards design, more complete definition enables parallel development, then last pieces cause slowdown.			

New development of predated system—internal knowledge of existing domain architectures.



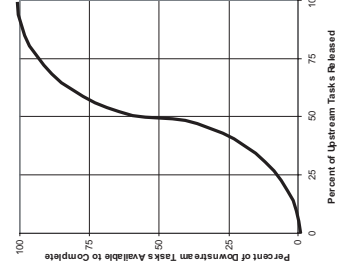
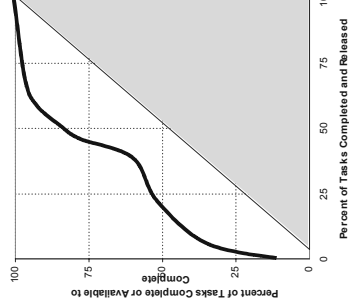
a = small or medium depending on organization.



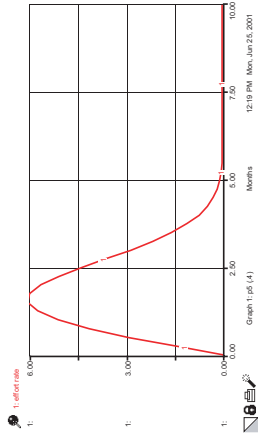
A mix of independent and interdependent tasks. Men and months are partially interchangeable.

Curve will vary with degree of concurrence

Reuse and new development—bottom-up reuse provides some initial components.



a = medium to large.

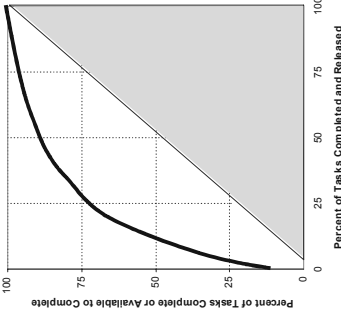
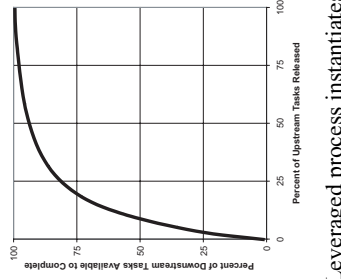


A mix of independent and interdependent tasks. Men and months are partially interchangeable.

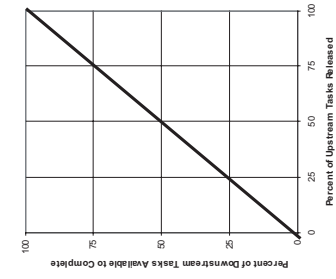
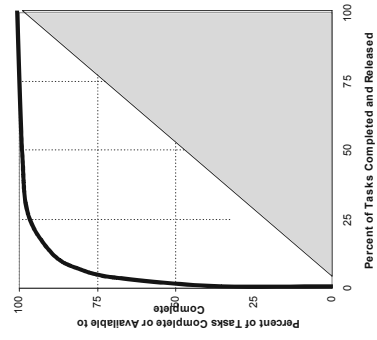
Curve will vary with degree of concurrence.

(continued)

Table 6.16. Task and effort dynamics summary for major project types (*continued*)

Project Type	Process Concurrence			Brooks's Interpretation
	Internal Concurrence	External Concurrence (inception to elaboration only) ²	Rayleigh Curve Modeling! (<i>a</i> = manpower buildup parameter)	
COTS-based—most requirements predefined by COTS capabilities; some glue code development necessary.			<i>a</i> = large—not a good fit for steady-state portions of rectangular staffing (but models extreme ramping up/down portions).	A mix of independent and interdependent tasks. Men and months are partially interchangeable.
Leveraged process instantiates software from defined COTS capabilities.				

Translation of existing application—pieces can proceed in parallel until final integration testing.



Elaboration pieces proceed at same rate as inception.

a = large—not a good fit for steady-state portions of rectangular staffing (but models extreme ramping up/down portions).
 Tasks can mostly be partitioned with no communication between them.
 Men and months are largely interchangeable

¹The generalized buildup patterns shown assume that all else is held constant between these examples except for the project type. Different organizations will exhibit different buildup patterns if they have to staff the same given projects. Some organizations will always be more nimble for quick staff buildup due to their internal characteristics.
²There is often self-similarity observed in the different phases. In many instances, the concurrence between elaboration and construction will be similar.

Cost expenditures and progress are tracked against previously defined work packages. Performance is measured by comparing three earned value quantities:

1. Budgeted cost of work performed (BCWP)
2. Actual cost of work performed (ACWP)
3. Budgeted cost of work scheduled (BCWS)

These quantities and other earned value parameters are displayed in Figure 6.56.

BCWP is the measure of earned value for work accomplished against the overall baseline plan. The accomplishment is stated in terms of the budget that was planned for it. BCWP is compared to ACWP to determine the cost performance (i.e., is the actual cost greater or less than the budgeted cost?). BCWS is the planned cost multiplied by the percentage of completion that should have been achieved according to the project baseline dates. Figure 6.56 shows these earned value quantities for an underperforming project with calculated variances and final projections based on extrapolating the actuals to date.

The cost variance is the difference between the budgeted and actual cost of work performed. Schedule variance quantifies whether tasks are ahead or behind of schedule

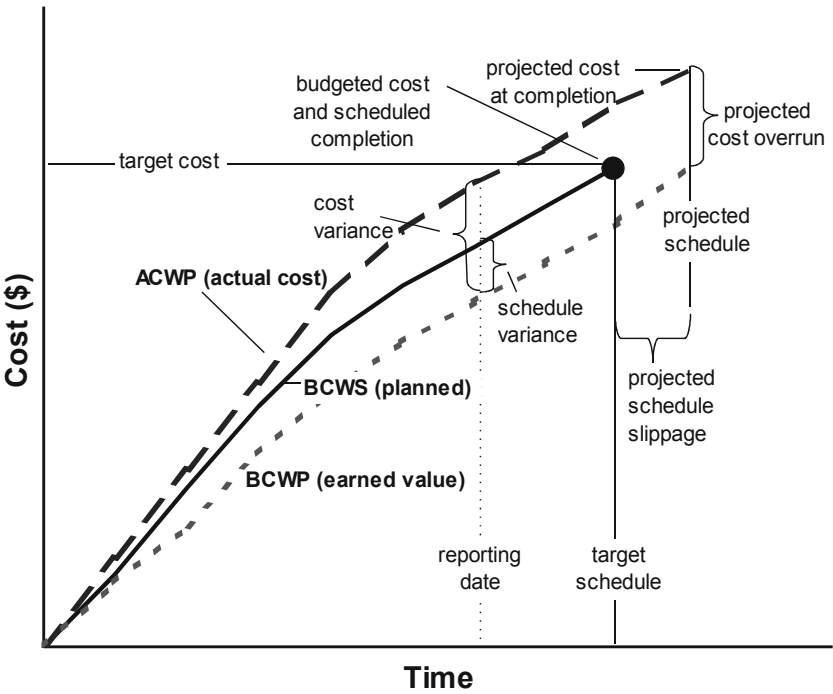


Figure 6.56. Earned value visualization.

at any point by taking the difference between the budgeted cost of work performed and the budgeted cost of work scheduled. These variance equations are:

$$\text{Cost variance} = \text{BCWP} - \text{ACWP}$$

$$\text{Schedule variance} = \text{BCWP} - \text{BCWS}$$

The variances are also visualized in Figure 6.56 with respect to BCWP, ACWP, and BCWS.

The cost performance index (CPI) represents cost efficiency against plan, or the ratio of budgeted cost for the work performed versus the actual cost of work performed for a specified time period. The schedule performance index (SPI) represents schedule efficiency against plan. Values greater than 1.0 represent performance better than planned, and less than 1.0 represent less efficient than planned. The performance indices are calculated as:

$$\text{CPI} = \text{BCWP}/\text{ACWP}$$

$$\text{SPI} = \text{BCWP}/\text{BCWS}$$

Earned value is highly recommended as an integral part of software project management. Trends should be evaluated often and remedial action taken quickly. Even if only 15% of progress has been completed and the budgets are being overrun, recovery is almost impossible. One should then focus on not getting worse, and adjust far-term projections based on current overruns.

An automated earned value system is particularly necessary to provide management visibility on large projects. In any case, a strong benefit to project understanding and organization is derived by thinking through project activities and assigning worth to them. More information on applying earned value concepts can be found in Reifer's software management tutorial [Reifer 2002].

However, there is an important caveat to standard earned value methodology. It may not be tracking the "real value" of a project. See Section 6.4 for more details of value-based software engineering and modeling examples for different stakeholder value functions.

6.7.1.1 General Project Procedures

Implementation of an earned value process can be scaled to fit projects of varied size and complexity. First establish a performance measurement baseline in monetary units such as dollars, or straight labor hours, to determine the budgeted cost of work scheduled for the life of the project. One should select the appropriate level of a work breakdown structure (WBS) for data gathering and appropriate means for determining earned value. This can range from a few top-level categories for which earned value is determined from percent completion estimates to projects for which earned value uses preplanned assignment of dollar values to completion of specific milestones or sub-tasks within tasks. For example, earned value can be determined by such methods as:

- Prorating value based on task percent completion
- Interim milestones with earned value linked to subtask completions, where a pre-determined amount is accrued for each completed subtask
- Level of effort with no definable work product, whereby the budget is laid out as a function of time

BCWS is the planned performance measurement baseline. When the project schedule is expressed as budgeted dollars (or hours of work), BCWS can be determined at any point using work scheduled to have been accomplished under the baseline plan. For large projects, near-term work elements can be budgeted into work packages, whereas future tasks may have budget values allocated as future planning packages or as undistributed budget.

Project status should be evaluated relative to BCWS to assess earned value for completed tasks and those in progress. Determine earned value for work accomplished to date as BCWP. Summing earned value across the WBS determines BCWP. If the work is on schedule, the BCWS will be equal the BCWP (if the earned value accurately reflects performance). If the cost actuals are on target with the budget, the BCWP will be equal to the ACWP.

6.7.2 Example: Earned Value Model

The earned value model described here was developed and used in training venues, and is summarized in Table 6.17. It is a self-contained model that takes planned and actual values from the user. The best use of such a model would be to integrate it with a larger project model, such as Abdel-Hamid’s, plugging in quantities for planned and actual progress (see the corresponding exercise at the end of the chapter).

The diagram for this model is shown in Figure 6.57, which implements the basic formulas for earned value using dynamic indicators. It has flow chains for product

Table 6.17. Earned value model overview

Purpose: Training, Control, and Operational Management			
Scope: Development Project			
Inputs and Parameters	Levels	Major Relationships	Outputs
<ul style="list-style-type: none">• Planned productivity• Planned staffing profile• Actual productivity• Actual personnel• Budgeted cost per milestone• Labor rate	<ul style="list-style-type: none">• Planned cumulative milestones• Actual cumulative milestones• Actual cost of work performed	<ul style="list-style-type: none">• Performance indices	<ul style="list-style-type: none">• Cost performance index• Schedule performance index• Budgeted cost of work performed• Actual cost of work performed• Budgeted cost of work scheduled

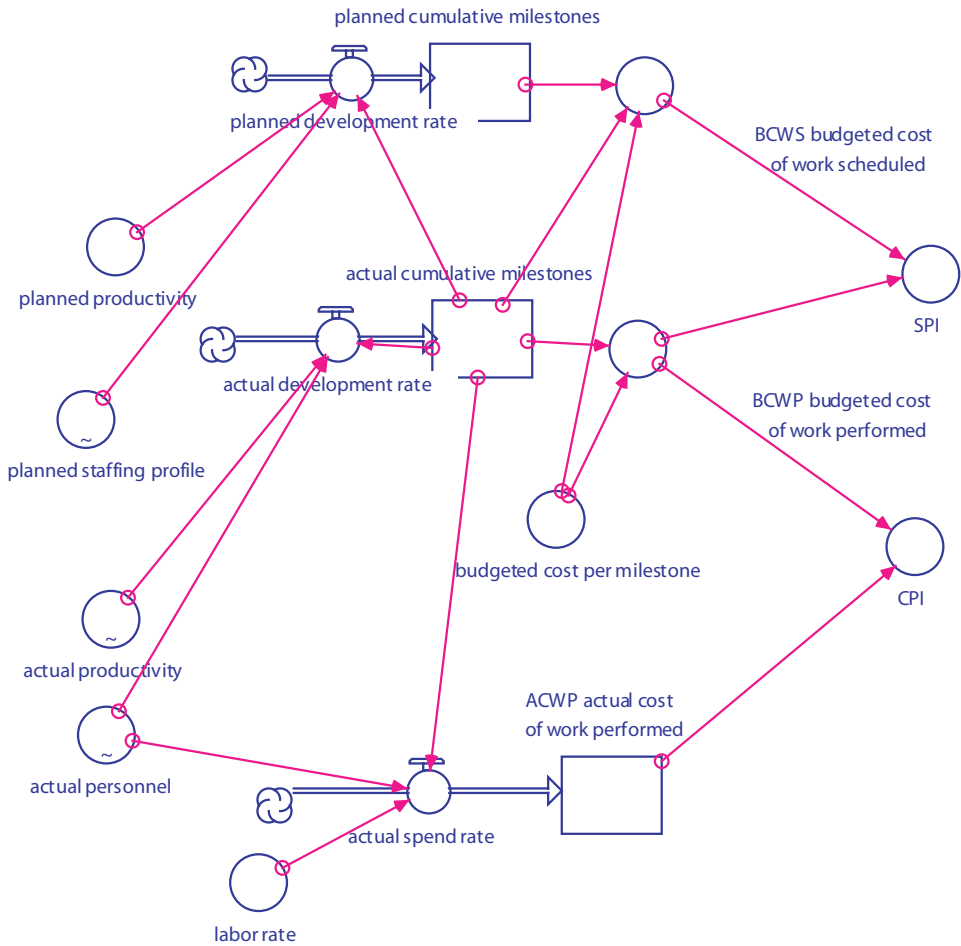


Figure 6.57. Earned value model.

tasks being developed—one for planned and one for actual. The user must input the planned number of people over time for the staffing profile, the planned productivity, and the budgeted cost per milestone.

Completed milestones are accumulated in the levels as a function of the development rates over time. The rates are determined with linear productivity formulas. The model also has a flow chain for actual money spent. For each time point in the simulation, the model compares actual progress to planned and computes the current earned value parameters, including SPI and CPI.

Task units are function points, since they apply across all project stages versus SLOC. Function points are also used as milestones in this case, and each function point is budgeted equally (an actual detailed plan could have unequally weighted milestones). The model is calibrated to a project plan for 100 function points to be devel-

oped at 0.1 function points/person-day. At \$1000 for each person-day of labor, a function point is worth \$10K.

6.7.2.1 Model Testing

We will use the earned value model to demonstrate some model testing procedures. The essential inputs to the model are the planned and actual productivities and staffing levels. Earned value quantities are then computed using the differences between planned and actual milestone completion. CPI and SPI are already normalized to 1.0, which makes the overall testing and model usage convenient.

To further simplify testing, a milestone is set equivalent to a function point in the model. The budgeted cost of each milestone (function point) is set to \$10,000. Most of the tests shown here use a baseline plan for the project using the following nominal values:

- Project size = 100 function points
- Planned productivity = 0.1 function points/person-day
- Planned staffing level = 10 persons

The productivity is a middle-ground value that we used in the Brooks's Law model. The other parameters are also chosen to simplify the validation calculations. The planned schedule and cumulative cost are calculated as:

- Planned schedule (days) = $100 \text{ function points} / (0.1 \text{ function points/person-days}) \cdot 10 \text{ persons} = 100 \text{ days}$
- Planned cost (dollars) = $100 \text{ function points} \cdot \$10,000/\text{function point} = \1 M

Four stages of testing are performed that progressively build confidence in the model by addressing more complex test cases. This is a typical sequence to take whereby the initial tests can be easily validated manually or with a spreadsheet, since rates remain constant. Once those test cases are passed, the model is assessed with increasingly dynamic conditions:

- Stage 1—use constants for all productivity and staffing levels
- Stage 2—use a mix of constants and time-varying inputs for actual productivity and staffing levels
- Stage 3—use time-varying inputs for actual productivity and staffing levels
- Stage 4—allow both planned and actual quantities to vary over time

Within each stage are subdivisions of test cases.

The testing is made convenient since all we have to change for the test cases are the productivity and staffing levels by drawing their time curves and/or typing in the numbers. See Figure 6.58 and Figure 6.59 for examples of specifying project actuals for a simulation run. These examples are intended to be realistic, whereby the productivity

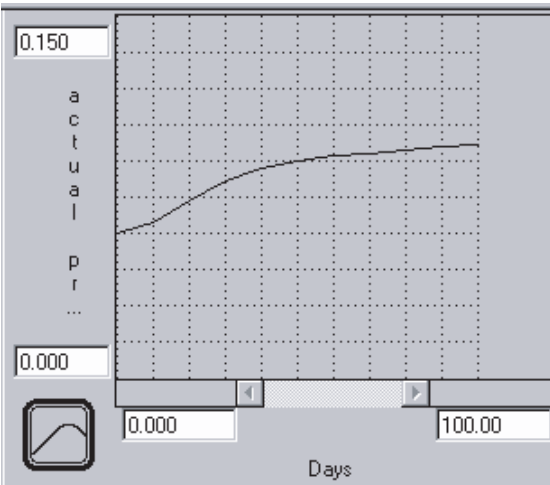


Figure 6.58. Example input for actual productivity (actual productivity vs. days).

shows a learning curve ramp and the staffing profile is tapered at the beginning and end.

The graphical productivity function can be replaced with another simulation model component, such as any of the other models in this book that calculate effective production rates. When using the earned value model in practice on a live project, the actual progress metrics should be substituted.

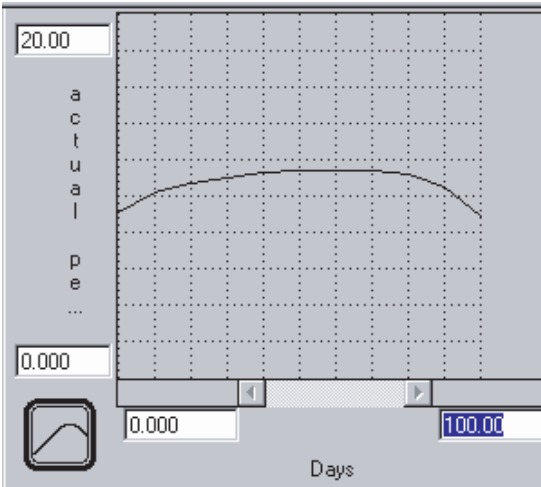


Figure 6.59. Example input for actual personnel (actual personnel vs. days).

The first test is to set the actual quantities equal to the planned ones, using constants for steady-state testing. We expect the CPI and SPI to remain at one during the entire simulation, since the project is exactly meeting its planned milestones on time and with the allotted budget. As expected, the cumulative milestones for planned and actual are equivalent, and both CPI and SPI remain at unity for the entire project because the actual work equals the budgeted work. The respective cost and schedule variances are both zero. The project does complete in 100 days for \$1M.

In the next test, we decrease just the productivity and keep the actual personnel equal to the planned. With lesser productivity, we expect the project to perform worse than planned on both cost and schedule grounds. It should take longer and require more effort than planned. Figure 6.60 shows the input for this case, where the actual productivity never quite reaches the planned. Figure 6.61 shows the corresponding output. As expected, both CPI and SPI stay less than one and the project finishes in 108 days for \$1.07 M.

The dynamic curves for ACWP, BCWP, BCWS, and the variances on the second chart in Figure 6.61 are keys to the CPI and SPI computation. Both variances start at zero and continue to grow negative. Upon inspection, the rate of the growing negative variances decreases as productivity starts to come close to the actual. The varying slopes of the variance curves can be studied to understand the progress dynamics and used to extrapolate where the project will end up.

To support further testing considering all ranges of values, a test matrix can be created that covers the different combinations of actual versus planned inequalities. The input conditions and expected outputs are shown in Table 6.18, where the inequalities are assumed to be for the entire project duration. This type of input/output matrix can help in model experimentation and validation.

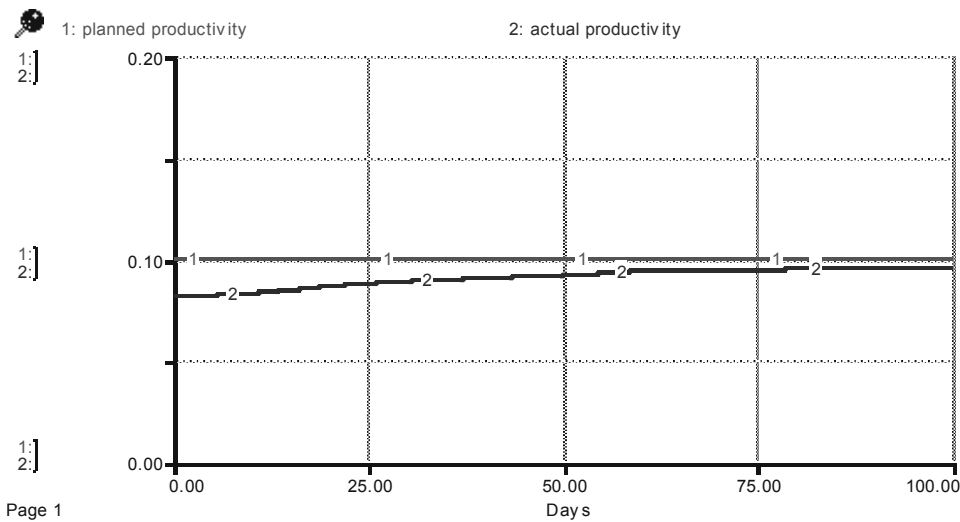


Figure 6.60. Test case inputs—actual productivity < planned productivity.

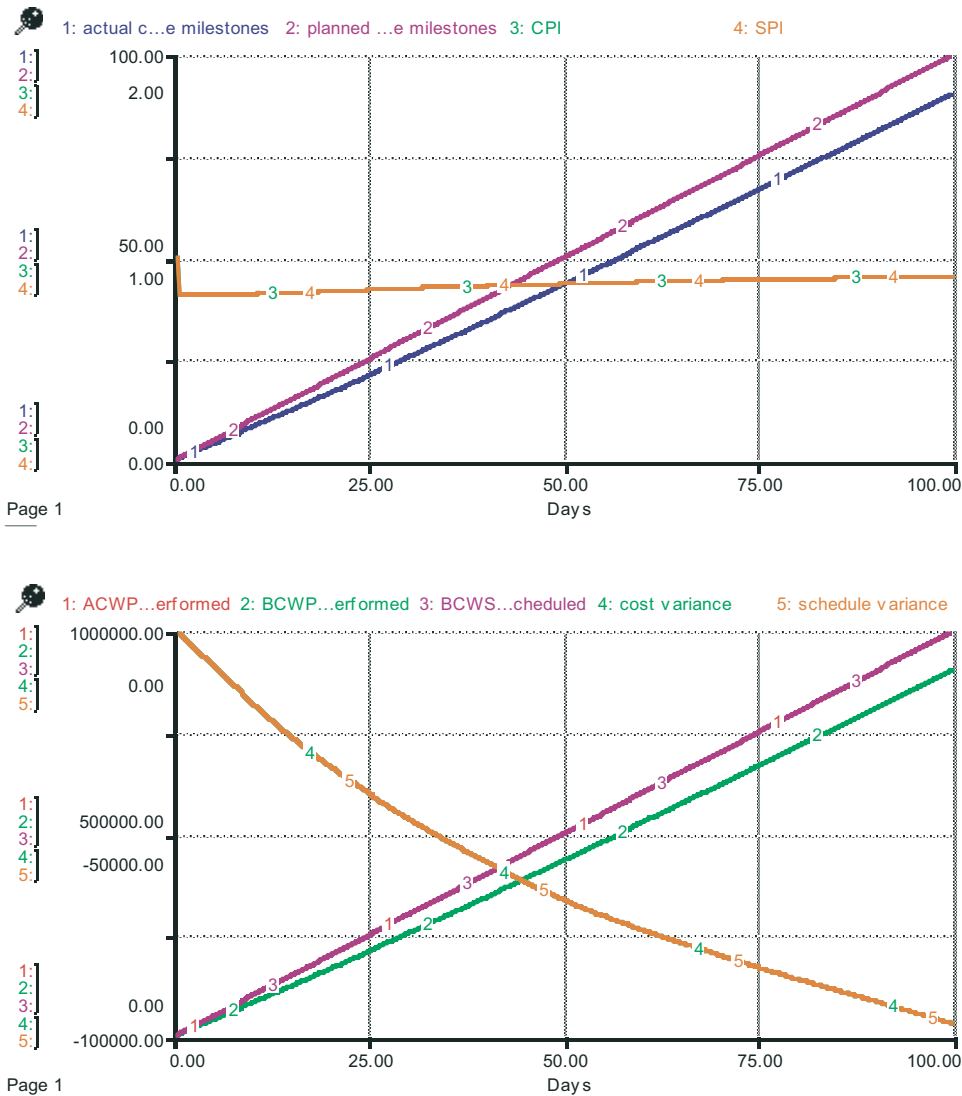


Figure 6.61. Test case results—actual productivity < planned productivity.

Figure 6.62 and Figure 6.63 show the progress and earned value outputs for the case of actual productivity being less than planned and actual personnel less than planned. Based on the results of these first six tests, we have increased confidence that the model correctly calculates earned value dynamically. Subsequent testing with more variations to the planned quantities is left to the student (see the chapter exercises), and more applied examples follow later in this section.

Table 6.18. Earned value test matrix and expected results

Personnel	Productivity		
	actual > planned	actual = planned	actual < planned
actual > planned	CPI indeterminate SPI > 1	CPI > 1 SPI > 1	CPI < 1 SPI indeterminate
actual = planned	CPI > 1 SPI > 1	CPI = 1 SPI = 1	CPI < 1 SPI < 1
actual < planned	CPI > 1 SPI indeterminate	CPI < 1 SPI < 1	CPI indeterminate SPI < 1

More interesting cases arise when the actual values fluctuate relative to the planned. The reader is encouraged to interact with the supplied model *earned value.itm* and experiment with different scenarios. Very minimal effort is required to graphically change the project plans or actuals by drawing them on a grid or, alternatively, keying in precise numbers.

6.7.2.2 Model Usage: Read the Slopes, Do Hard Problems First

To gain more insight into evaluating cost and schedule performance with CPI and SPI, we will play with the model to visualize the dynamic earned value trends. This is an example of using a model to further one’s knowledge and management skills without making costly mistakes on an actual project.

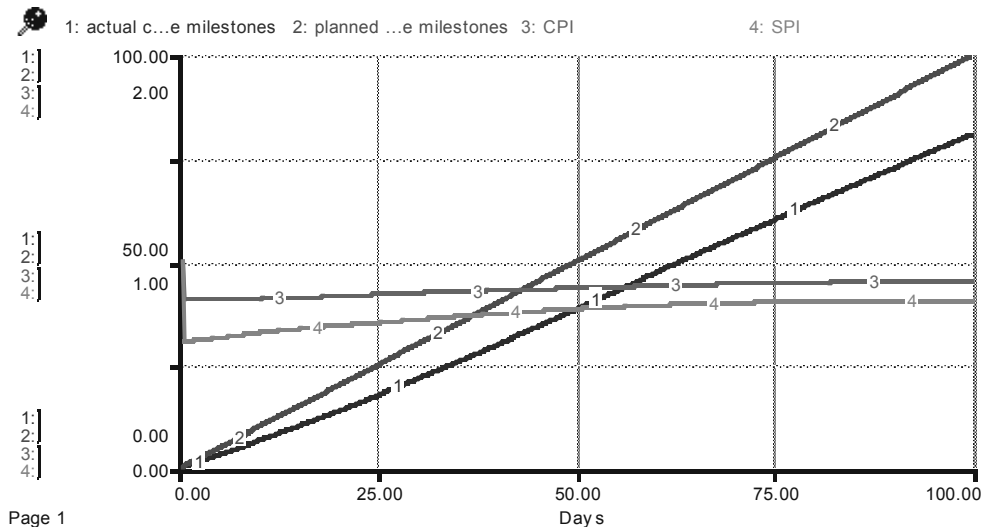


Figure 6.62. Actual productivity < planned productivity and actual personnel < planned personnel (1: actual cumulative milestones, 2: planned cumulative milestones, 3: CPI, 4: SPI).

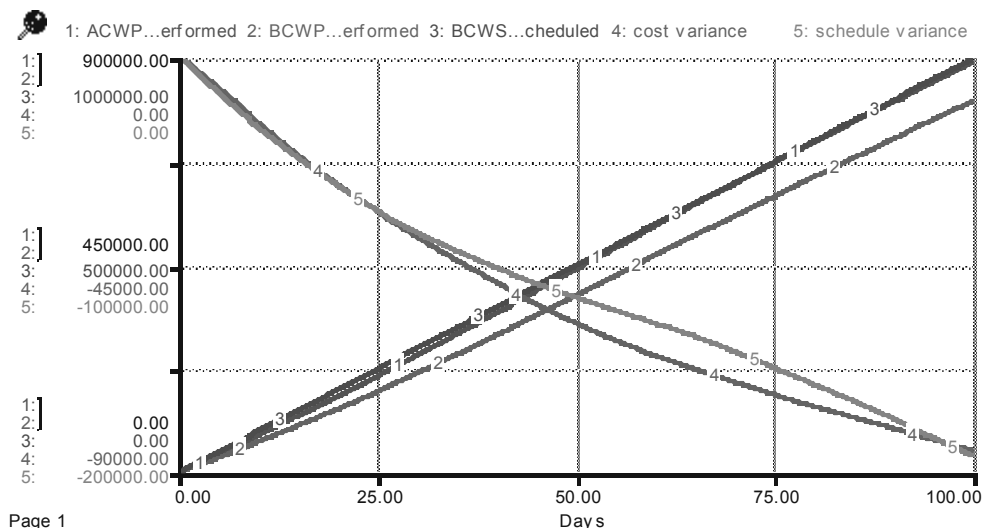


Figure 6.63. Actual productivity < planned productivity and actual personnel < planned personnel (1: actual cost of work performed, 2: budgeted cost of work performed, 3: budgeted cost of work scheduled, 4: cost variance, 5: schedule variance).

We will demonstrate how to spot trends early in order to effect control. This is done by monitoring the *slope* of earned value trends as opposed to current static values only. A quick glance at early trends to assess task completion progress may be deceiving if the slopes (rates of change) are not considered.

For example, a project that starts out solving easy problems at a fast pace creates an illusion of early overall completion. But the slope of the progress line, also borne out in the dynamic CPI/SPI curves, clearly shows a worsening negative trend. The following case demonstrates this.

A partial simulation of two comparative projects stopped after about 25% completion is shown in Figure 6.64. The two projects are being simulated with the same planned progress. Assuming that the same trends continue, the question is, which project will finish first? Many people fail to consider the changing slopes of the initial progress trends, and erroneously choose which project will perform best.

The answer lies in the assessing the slopes of the actual progress lines, which are also reflected in the CPI and SPI curves. Figure 6.65 shows the final comparison between the progress trends, from which it is clear that project 2 finishes substantially sooner. The battery of trends for projects 1 and 2 are respectively shown in Figure 6.66 and Figure 6.67.

Figure 6.66 clearly shows that the rate of accomplishment continues to diminish for project 1; the slope of the actual progress line keeps decreasing. Likewise, the BCWP slope decreases. Accordingly, the CPI and SPI curves fall monotonically. But the slope of the actual progress line or BCWP for project #2 in Figure 6.67 keeps increasing, and

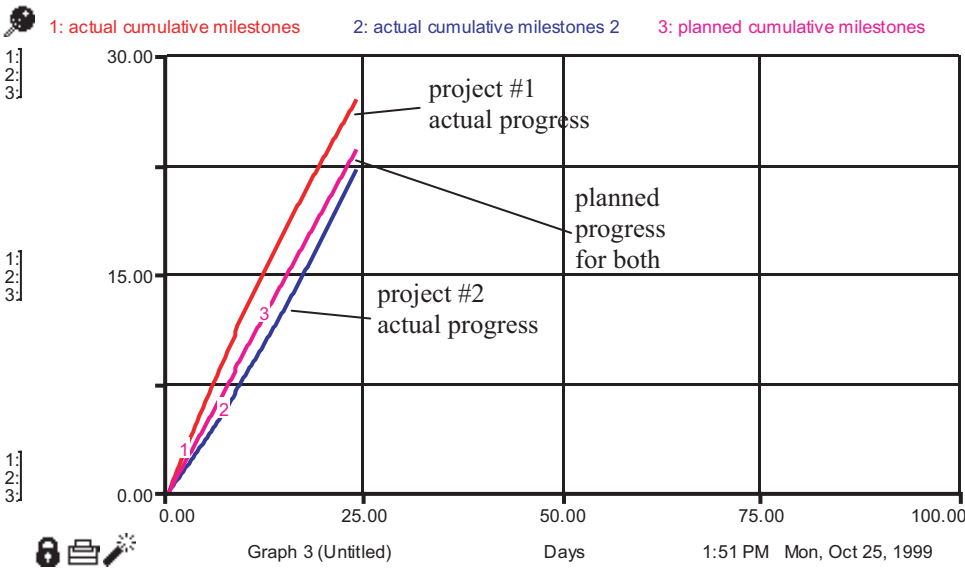


Figure 6.64. Early comparison of two projects.

the CPI and SPI measures continually improve to be greater than one. The lesson is to read, interpret, and extrapolate dynamic trends, rather than look only at static values for cost and schedule performance.

This example also demonstrates one of our primary recommendations: Do the hardest problems first (and mitigate the largest risks first). What drove this simula-

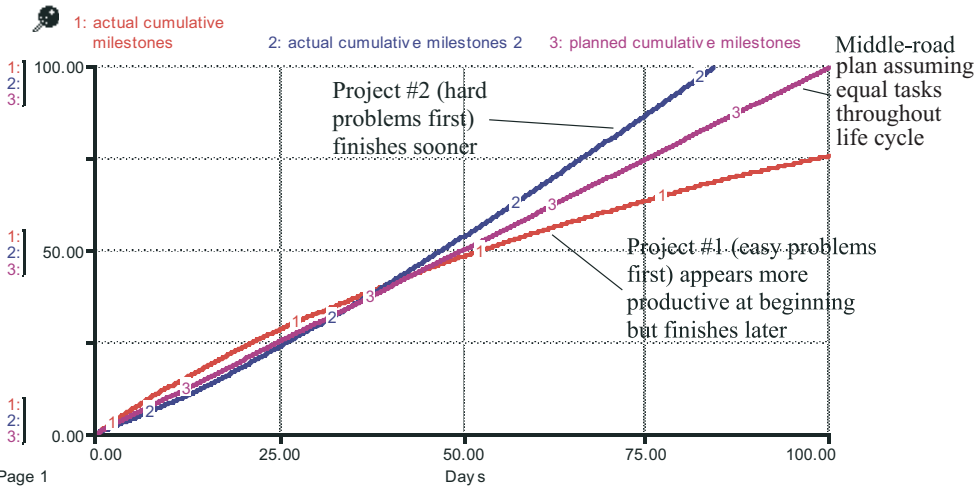


Figure 6.65. Final comparison of two projects.

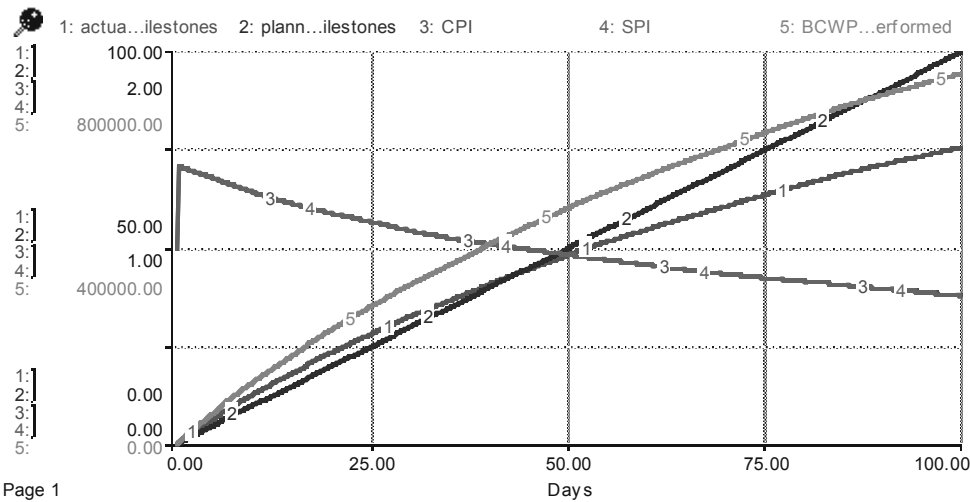


Figure 6.66. Project #1 trends.

tion example is varying the task difficulties. The planned progress assumed that all tasks were equal. Project #1 was modeled to do easy tasks first. This is what gave the illusion of early progress beating the plan. Project #2 was assumed to do hardest tasks first, and its rate of accomplishment continually increased. See the model *earned value project comparison.itm* to see how the varying task difficulty was implemented.

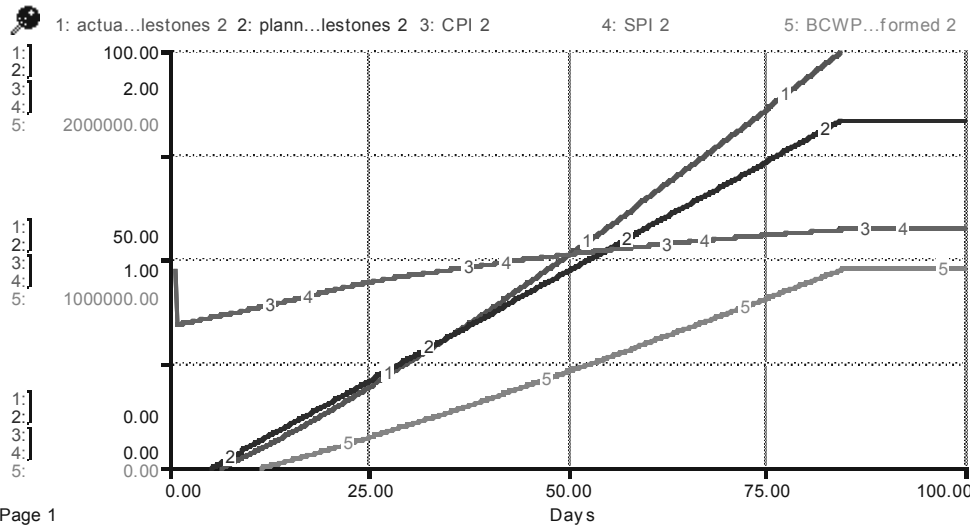


Figure 6.67. Project #2 trends.

6.7.2.3 Litton Applications of Earned Value Model

The earned value model was used at Litton to (1) demonstrate what earned value is, and (2) show how it can be used to manage a project using feedback. Managers, project leads, and metrics analysts were the primary trainees. Examples involved evaluating progress compared to planned completions, such as the example in the section above. The partial simulation of two comparative projects was first shown to students (such as in Figure 6.64) and they were asked to predict which project would finish first. Further simulations were run and explained, and then the students sat down to interact with the model themselves.

The other part of the training involved managers interacting with project simulations as they are running in a “flight simulation” mode. The earned value model is used for hands-on practice in this manner. For example, simulated CPI and SPI trends are monitored by an individual who can control certain parameters during the run. Typically, the simulation is slowed down or paused such that individuals can react in time by varying sliders. In the earned value simulation, managers can control the staffing levels interactively as the simulation progresses. A flight simulation interface to the earned value model was developed, by which the user can adjust staff size during the project with a slider.

6.8 MAJOR REFERENCES

- [Abdel-Hamid, Madnick 1991] Abdel-Hamid T. and Madnick S., *Software Project Dynamics*, Englewood Cliffs, NJ: Prentice-Hall, 1991.
- [Biffi et al. 2005] Biffi S., Aurum A., Boehm B., Erdogmus H., and Grünbacher P. (Eds.), *Value-Based Software Engineering*, New York: Springer, 2005.
- [Reifer 2001] Reifer D., *Making the Software Business Case*, Reading, MA: Addison-Wesley, 2001.
- [Reifer 2002] Reifer D., *Software Management* (6th Edition), Los Alamitos, CA: IEEE Computer Society Press, 2002.
- [Royce 1998] Royce W., *Software Project Management—A Unified Approach*, Reading, MA: Addison-Wesley, 1998.
- [Schrage 2000] Schrage M., *Serious Play*, Boston, MA: Harvard Business School Press, 2000.

6.9 PROVIDED MODELS

The models referenced in this chapter are provided for usage to registered book owners and listed in Appendix C. See the book website for model updates and additions.

6.10 CHAPTER 6 SUMMARY

Project and organization applications include estimating, planning, tracking, controlling, setting policies on how to do things, long-term strategies, and looking after peo-

ple. Projects and organizations provide the context for goals, constraints, and priorities of software development. Decision structures are embodied in project and organization applications to monitor status against plans, track expenditures and business value measures, and enable dynamic decisions regarding people, processes, products, and so on.

Organizations are typically collections of projects but also have broader initiatives (e.g., process improvements or shareholder return) and provide the supporting resources for them. Thus, the techniques for improving projects usually in turn benefit the organization. Organizations are responsible for people concerns and there is substantial overlap with people applications.

Abdel-Hamid's landmark model of integrated project dynamics has a primary focus on management of software production rather than software production itself. The model includes sectors for human resources, planning, controlling, software production, quality assurance and rework, and system testing. Many interesting results have been derived from experiments with the model, particularly in the area of software management policies. Retrospectively, the Abdel-Hamid model also illustrates some methods of poor software management. For example, the control function uses a dangerous policy of defining progress as effort expended against plan instead of measuring the tasks done.

Value-based software engineering considers different components of utility that software provides to stakeholders. It is different from earned value used to track a project and can be termed "real" earned value with respect to the stakeholders. Stakeholder objectives can be far and wide, and simulation can be used to evaluate and trade them off. For example simulation lends itself very well to assessing the business rationale of a software project and quantifying decision options.

A value-based product model incorporated two major aspects of stakeholder value: business value to the development organization stemming from software sales, and the value to the end-user stakeholder from varying feature sets and quality. The model helped identify features with diminishing returns and demonstrated that poor quality can severely hurt an enterprise, even after the problems are fixed. The model was also used to determine the optimal sweet spot of a process.

Personnel resource allocation is a most crucial aspect of management. Various models demonstrated different policies (e.g., needs-based or fixed allocation of resources), and can be used for actual decisions when parameterized for real projects. The adopted policies will depend on project context, constraints, and value priorities.

Some dynamic staffing models include the Rayleigh curve and process concurrence. One of the underlying assumptions of both is that the number of people working on a project is approximately proportional to the number of problems ready for solution at that time.

The Rayleigh curve when integrated is an excellent example of a structure producing an S-curve behavior for cumulative effort. The learning function component representing elaboration growth increases monotonically, whereas the work gap diminishes over time as problems are worked out. The corresponding effort rate rises and falls in a Rayleigh shape. It is been observed that the staffing buildup rate is largely invariant within a particular organization and reflects its characteristics. Despite the pervasive-

ness of Rayleigh-based estimation models, organizations cannot always respond as fast as some Rayleigh models dictate.

Process concurrence modeling can be used to see if and how much the software process can be accelerated. Both internal and external process concurrence relationships were explored. It was shown that process concurrence provides a robust modeling framework and can model more realistic situations than the Rayleigh curve.

Earned value is a valuable method for measuring performance with respect to plans and helping to control a project. It works best when thinking dynamically and can help develop such a mindset. The earned value model can be employed on projects, and was used in industry for earned value training showing how it can be used to manage a project using feedback. The model can be used to train managers to spot trends early in order to administer controlling corrective actions.

It was demonstrated that interpreting the slopes of earned value quantities provides much insight into trends not easily observed otherwise, and enables one to better extrapolate into the future. Static values do not allow this. One important lesson learned from experiments with the earned value model is that a policy of working hard problems first is advantageous in the long run.

6.11 EXERCISES

These application exercises are potentially advanced and extensive projects.

- 6.1. Consider the more modern practices in the Abdel-Hamid model. Define a project to modify some components of it but retain those parts that still apply for your environment. For example, you may choose to replace the software production sector or completely revamp the quality assurance sector for modern peer reviews or the team software process. You may get some ideas for improvement from the critique presented in this chapter. Define the problem(s) to address, include the usage context, conceptualize a high-level model, consider data sources to use, and begin model formulation. Elaborate further and implement the model depending on your goals and constraints.
- 6.2. Organizations often undergo simultaneous and/or sustained changes to processes, organizational structures, personnel, product lines, market shifts, and so on. However, organizations and their people ultimately have limits on the level of changes that can be absorbed at one time. How might one model the limiting factors? For example, it has been hypothesized that many organizations are averse to hiring more people when management overhead nears about 25% of the total workforce (since more managers are needed to supervise a larger workforce). Why might this limit be true? Give examples of other project or organizational situations for which change limits apply, investigate them through modeling, and draw conclusions.
- 6.3. Extend the value-based product model with a more comprehensive quality model. It should allow the user to experiment with details of different quality practices and assess the same process trade-offs already demonstrated.

- 6.4. Add a services sector to the value-based product model to replace revenues from product sales. In this context, an organization derives revenue from their developed software by using it to perform services to other parties. Model the services market and dynamics for your environment.
- 6.5. Consider how the views of different classes of project stakeholders may impact the degree of model aggregation. For example, does an oversight stakeholder need the same amount of detail on individual tasks as those working on implementation? Propose and justify appropriate levels of task aggregation to model production in the following scenarios:

- A three-person team developing an internal prototype in 5 weeks
- A RAD project with 15 people developing an e-commerce solution in 100 days with two team leads
- A software development team varying from 4 to 8 people working for about 18 months under one project manager
- A multicontractor distributed team of 500 people developing a large integrated system over the course of 4 years.

Identify the different stakeholders and their needs in representative situations. Consider anonymity issues within a team and external to a team. When is data on individual entities appropriate and useful and when should model entities be aggregated together? A goal-question-metric approach can be used to define appropriate metric indicators at different hierarchical levels of the project.

Software metrics “etiquette” should be followed. The table below from [Grady, Caswell 1992] provides some organizational guidelines for a metrics program.

Functional Management	<ol style="list-style-type: none"> 1. Do not allow anyone in your organization to use metrics to measure individuals. 2. Set clear goals and get your staff to help define metrics for success. 3. Understand the data that your people take pride in reporting. Never use it against them; do not ever even hint that you might. 4. Do not emphasize one metric to the exclusion of others. 5. Support your people when their reports are backed by data useful to the organization.
Project Management	<ol style="list-style-type: none"> 6. Do not try to measure individuals. 7. Gain agreement with your team on the metrics that you will track, and define them in a project plan. 8. Provide regular feedback to the team about the data they help collect. 9. Know the strategic focus of your organization and emphasize metrics that support the strategy in your reports.

- Project Team
10. Do your best to report accurate, timely data.
 11. Help your managers to focus project data on improving your processes.
 12. Don't use metrics data to brag about how good you are or you will encourage others to use other data to show the opposite.
-

6.6. Adapt the Brooks's Law model from Chapter 1 and implement the enhancements listed below.

- Add a stop to the simulation when all requirements are developed. This will prevent the model from running overtime.
- Add a simple feedback loop that controls the personnel allocation rate by comparing actual production to planned production. The existing model covers actual production; the planned production assumes a constant development rate with all 500 function points completed at 200 days.
- Add logic for a one-time-only correction when the difference between actual and planned is 65 function points. Run the model and show the results for adding 0, 5, 10, and 20 people.
- Make the model scalable for larger team sizes up to 60 people to overcome the current restriction on team size maximum. You might simulate team partitioning for this (see below).

Advanced Exercise

Add the effects of partitioning to the resulting model and try to validate it against empirical data. You can use the data from [Conte et al. 1986] on average productivity versus personnel level to help develop it (if you are using an analytical top-level approach). It would be even better to model the underlying mechanics of how teams are formed and use the data to test your model as described below. [Briand et al. 1999] also provides data analysis on the effect of team size on productivity. Normalize the empirical data on productivity into a relative relationship for evaluation against the Brooks's Law model.

Test the revised Brooks's Law model by putting it into steady state and comparing it to empirical data on software productivity versus team size. Put it into steady state such that personnel do not evolve into experienced people (otherwise the results will be confounded). Make an array of runs to test the model at different project sizes; show the results and discuss them.

Plot the results for individual productivity versus increasing team size. Productivity can be calculated as the overall software development rate divided by the team size for each simulation run for which team size was varied. Compare the plot of individual productivity against the normalized relationship from empirical data. They should exhibit the same qualitative shape.

Background

Brooks stated that adding people to a software project increases the effort in three ways: training new people, added intercommunication, and the work and disruption of partitioning. Repartitioning of teams is a particularly difficult effect to model, as witnessed by Brooks. In his 1995 updated version of *The Mythical Man-Month*, Brooks reviews the Abdel-Hamid Brooks's Law model and one by Stutzke [Stutzke 1994]. He concludes "neither model takes into account the fact that the work must be repartitioned, a process I have often found to be nontrivial." Our experience in assigning this as student homework attests to that fact.

During a real project, as the overall team grows in size, partitioning of teams does take place. More subteams will be needed on a 100-person project than a 10-person project. People naturally subdivide the work per their management instincts as the product design allows. Effects that come into play are two types of communication overhead: (1) the intrateam overhead that we have already modeled and (2) the interteam communication overhead. Either type can overwhelm a project if partitioning is done suboptimally. Any single team that is too large will be swamped with communication, whereas too many small teams will be burdened with trying to communicate between the teams. The team leads designated to coordinate with other teams will have no time for anything else if there are too many other teams to interface with. The balance that is to find is the right number of teams of reasonable size.

First, suppose a team grows from 20 people to 80 without reforming of teams. What does our current model show? The simple model is only scaled for teams up to 30 people, so it is restricted for larger team sizes. If we simply continue with the formula $\text{overhead} = 0.06 \cdot \text{team}^2$, then the equations quickly become greater than 100%. What is wrong with this? It does not account for repartitioning. It assumes that a single team will remain and continue to grow indefinitely in size without breaking into more teams.

Assume that the project acts like a self-organizing system. It will develop additional teams in a near-optimal manner to meet the demands of increased work scope. This is the self-partitioning effect we wish to model.

- 6.7. As a converse variation of the Brook's Law effect, model the impact of taking people off of a continuing project. There is extra coordination work required when people leave a project, associated with sorting out and covering the tasks of those leaving. Since there are fewer workers and the project is slowed down, management may increase pressure. This may then lead to others leaving. In this modeling context, the staff levels are reduced, leaving the same workload to be covered by fewer people.
- 6.8. The generic Rayleigh curve always starts out at zero, but in reality most projects start with a set number of staff. Planners adjust the Rayleigh curve in actual practice to start out with a more reasonable amount of people. Modeling this may involve clipping the curve or revising the equation parameters. Take

the basic Rayleigh curve model and augment it with an initial offset to represent a starting point up on the curve. Parameterize the model so that users can input the desired initial staff size.

- 6.9. Some have theorized that aspects of software processes exhibit self-similar patterns at difference scales. Such phenomena have been described with chaos theory or fractals in other fields. Examine whether such techniques can be applied to superimposed staffing patterns like Rayleigh curves, process concurrence relationships, or any other aspect of software processes. For example, many small Rayleigh curves together can produce a larger Rayleigh curve, and ad infinitum.
- 6.10. Augment the Norden/Rayleigh model for the Putnam time constraint equations. Try to validate the time constraint formula against actual data and document your findings.
- 6.11. Develop a simple staffing model by replacing the $p(t)$ term in the Rayleigh formula with a manually drawn curve provided by the user that represents the accumulation of requirements available to be implemented.
- 6.12. Develop a model that replaces the $p(t)$ term in the Rayleigh formula with a process concurrence formula.
- 6.13. Evaluate the hypothesis that an organization striving to achieve RAD may choose to use a variable staffing profile as opposed to a constant profile for the upfront systems engineering effort. What conditions of the project environment make a difference as to which profile type is more suitable?
- 6.14. Adapt any of the provided process concurrence models or those of Ford and Sterman to be fully iterative. Model an iterative process like RUP or MBASE, including phases and activities. The model should be scalable for any number of iterations per phase. Calibrate it with available data.
- 6.15. Test the hypothesis that for best schedule performance, the optimal systems engineering staffing should be front loaded as opposed to constant level of effort on a complex project with partial interphase concurrency (see the process concurrence section). Assume a fixed project effort and standard percentage of systems engineering for the project (25% would be a reasonable figure). Compare the schedule performance for the level-of-effort systems engineering staffing with a front-loaded staffing profile. Vary the staffing profiles under comparison. Is there an optimal staffing profile for the concurrency relationship you used? Quantify the cost and schedule trade offs and draw conclusions.
- 6.16. Integrate the earned value model with an independent project model (such as Abdel-Hamid's or another), so that planned and actual quantities are computed internally instead of input directly by the user. Develop and run scenarios to assess various management actions using feedback from the earned value indicators.
- 6.17. Model the impact of requirements volatility in the context of earned value tracking. Suppose management wants to understand the consequences of tak-

- ing on more requirements changes and they ask, “What will be the impact to our CPI and SPI with a given percentage of requirements volatility?” Start with the earned value model and analyze the impact of varying levels of requirements changes. You can report your results in two ways: (1) adjusting the earned value baseline assuming that the changes are out of scope and (2) taking on the changes without adjusting the baseline.
- 6.18. (This exercise is for the mathematically inclined.) Examine varying the choice of the dt interval in the Rayleigh curve generator and tie this to feedback delays. Can equivalent behavior be created between a Rayleigh curve and a substituted policy of hiring delays? Show your results and explain why or why not.
 - 6.19. Model the “rubber-band” schedule heuristic identified and described in [Rechtin 1991]. He states that “The time to completion is proportional to the ratio of the time spent to the time planned to date. The greater the ratio, the longer the time to go.” Use the schedule stretch-out experienced to date on a project to recalibrate and extrapolate the future remaining schedule. Alternatively, analyze this effect on a completed project using midpoint status references. Does it make sense to include this schedule effect in a cost estimation model? Describe what type(s) of projects and conditions it might hold true for.
 - 6.20. Weinberg describes the dynamics of various management issues with causal-loop-type diagrams in [Weinberg 1992]. Choose one or more of them and elaborate them into simulation models. Experiment and derive policy conclusions from the results.
 - 6.21. Compare and evaluate different product line strategies with a dynamic product line model. The provided COPLIMO static model spreadsheet or another referenced model can be used as a starting point. If a static model is adapted, then reproduce the static calculations first as a steady-state test before introducing test cases with dynamics.
 - 6.22. Discuss and model the variability in metrics analysis intervals for different software processes. For example, effort reporting may only be available at monthly intervals but late testing activities may be statused on a daily basis. Run experiments to determine trade offs and to optimize the metrics feedback process for visibility and decision making.