3

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.

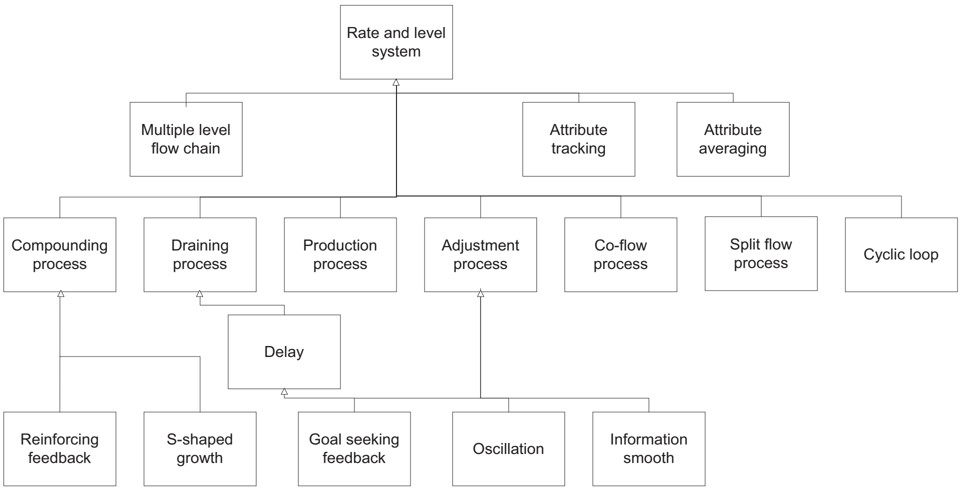


Figure 3.1. Class hierarchy for model structures.

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-

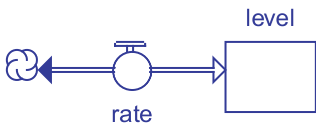


Figure 3.2. Rate and level system.

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:

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

When there are separate inflows and outflows the equation becomes

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

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-

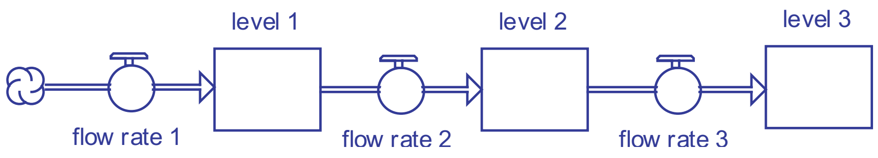


Figure 3.3. Flow chain.

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).

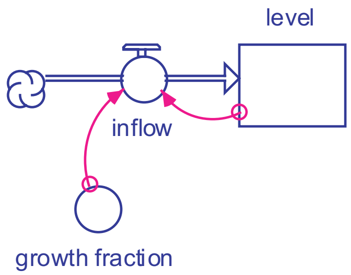


Figure 3.4. Compounding process. Equation:

inflow = level · 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

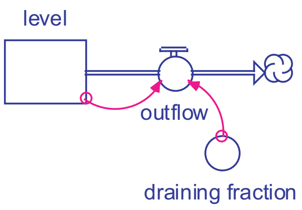


Figure 3.5. Draining process. Equation:

outflow = level · draining fraction

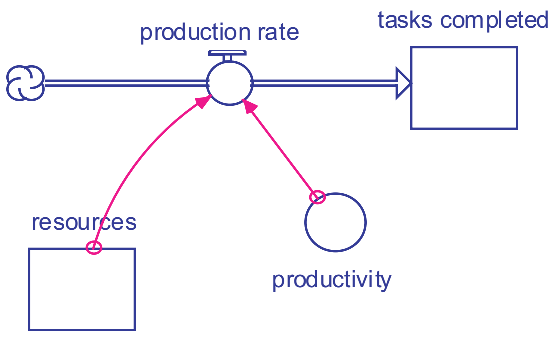


Figure 3.6. Production process. Equation:

production rate = resources · 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

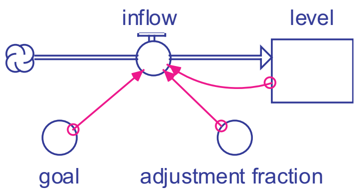


Figure 3.7. Adjustment process. Equation:

inflow = (goal – level) · adjustment fraction

or

inflow = (goal – level)/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 subflows 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.

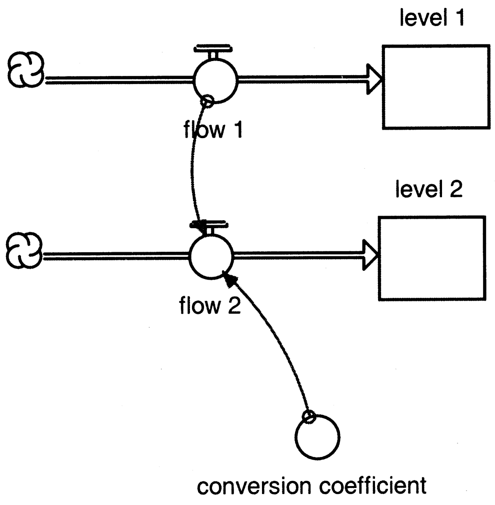


Figure 3.8. Coflow process. Equation:

flow 2 = flow 1 · conversion coefficient

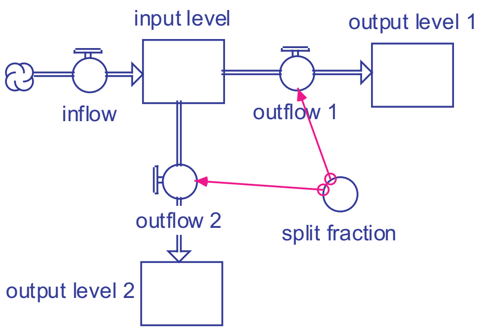


Figure 3.9. Split flow process. Equation:

outflow 1 = *f*(split fraction)

outflow 2 = *f*(1 – split fraction)

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

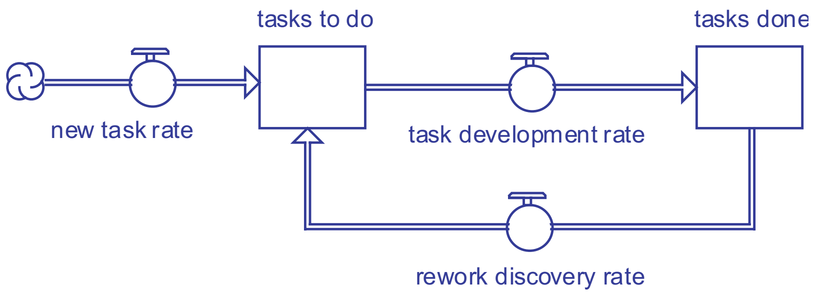


Figure 3.10. Cyclic loop process. The backflow rate formula has no special restrictions.

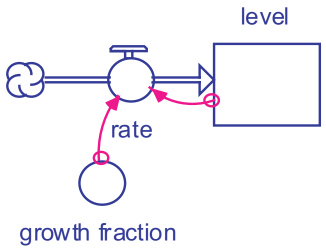


Figure 3.11. Exponential growth infrastructure. Equation:

rate = level · 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

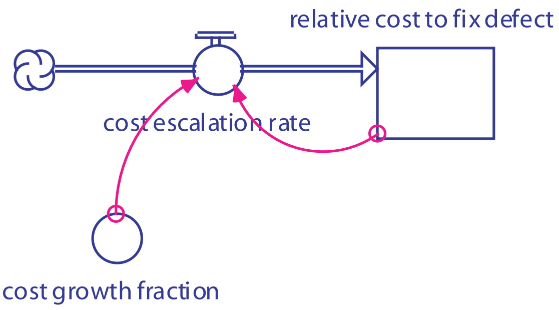


Figure 3.12. Cost-to-fix exponential growth structure.

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

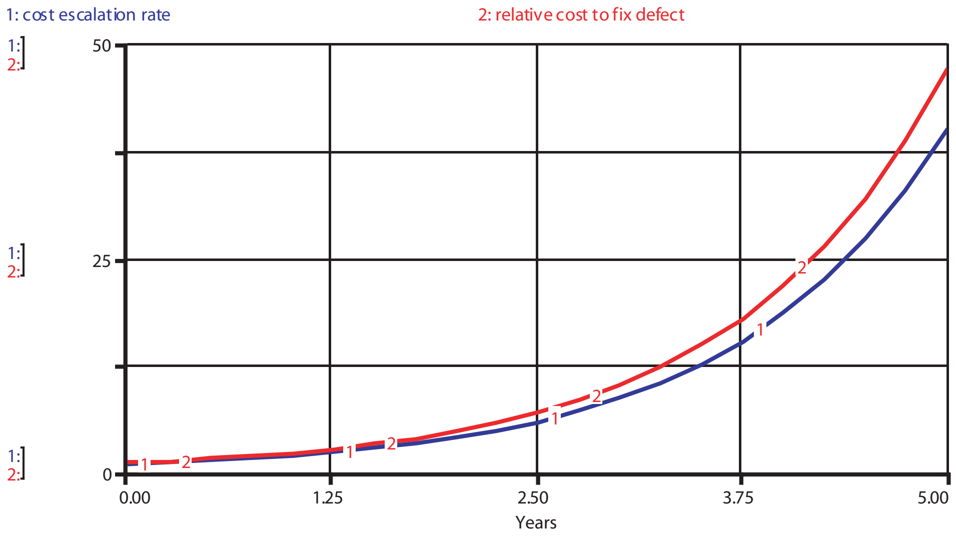


Figure 3.13. Exponential cost-to-fix behavior.

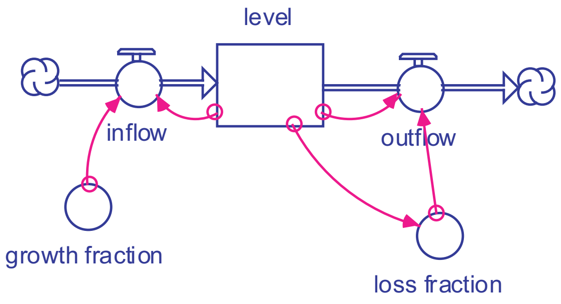


Figure 3.14. S-shaped growth infrastructure. Equations:

inflow = level · growth fraction

outflow = level · loss fraction

loss fraction = graph(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

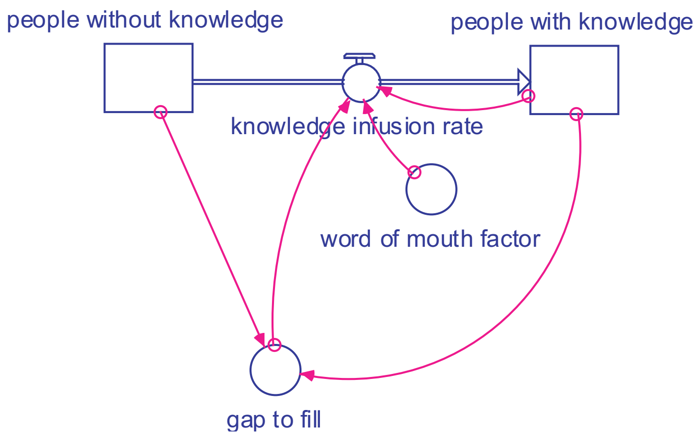


Figure 3.15. Knowledge diffusion structure. Equations:

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

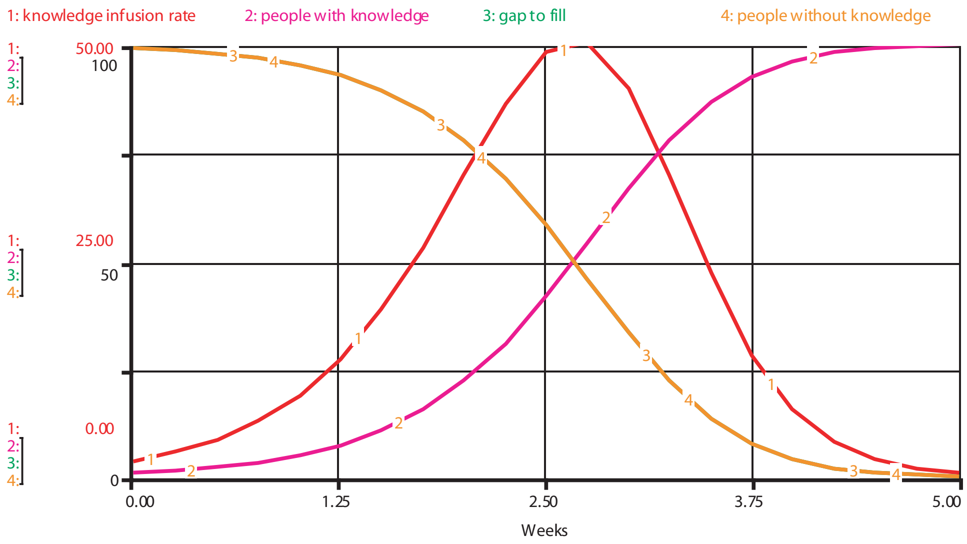


Figure 3.16. Knowledge diffusion S-curve behavior.

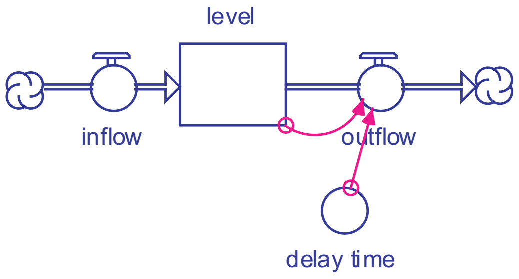


Figure 3.17. Delay structure. Equation:

outflow = level/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.

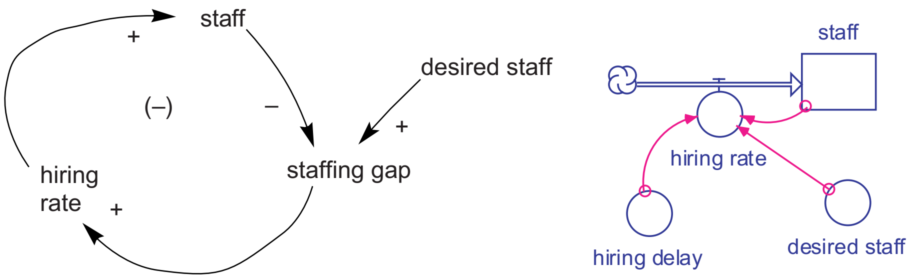


Figure 3.18. Goal-seeking hiring delay structure. Equation:
hiring rate = (desired staff – staff)/hiring delay

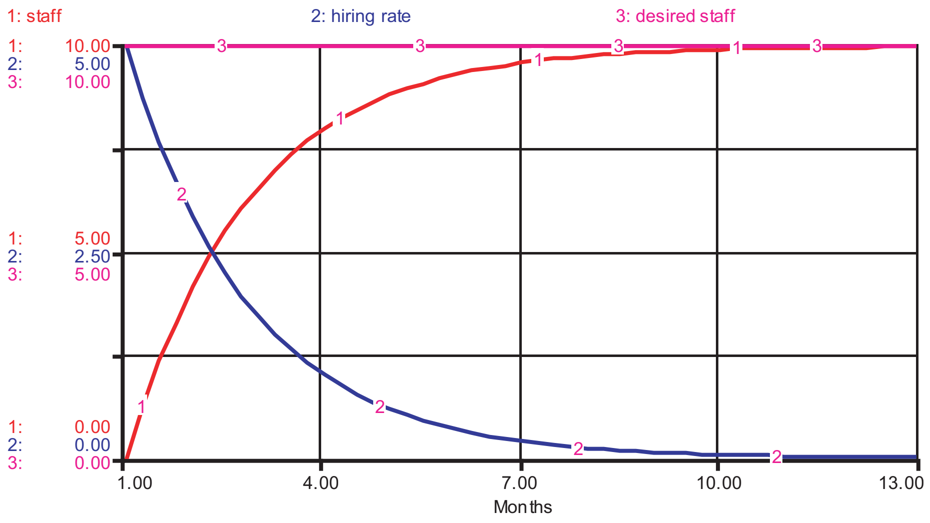


Figure 3.19. First-order hiring delay behavior.

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.

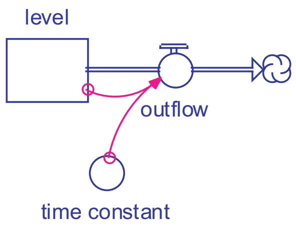


Figure 3.20. Exponential decay. Equation:

rate = level/time constant

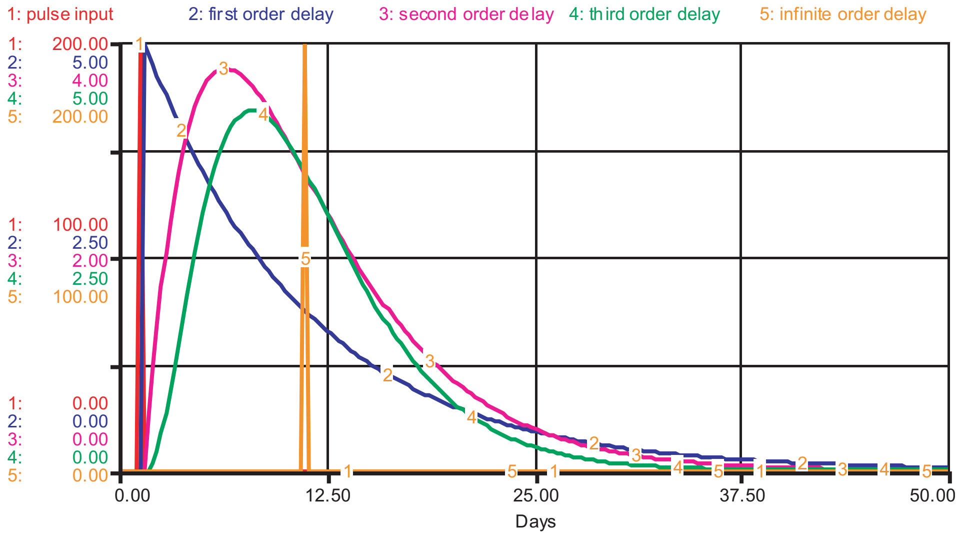


Figure 3.21. Delay responses to impulse.

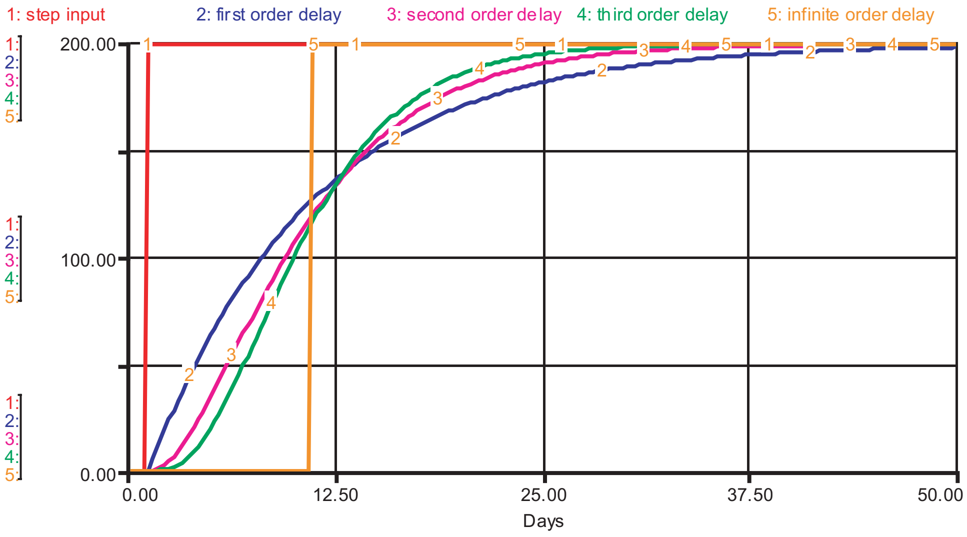


Figure 3.22. Delay responses to step input.

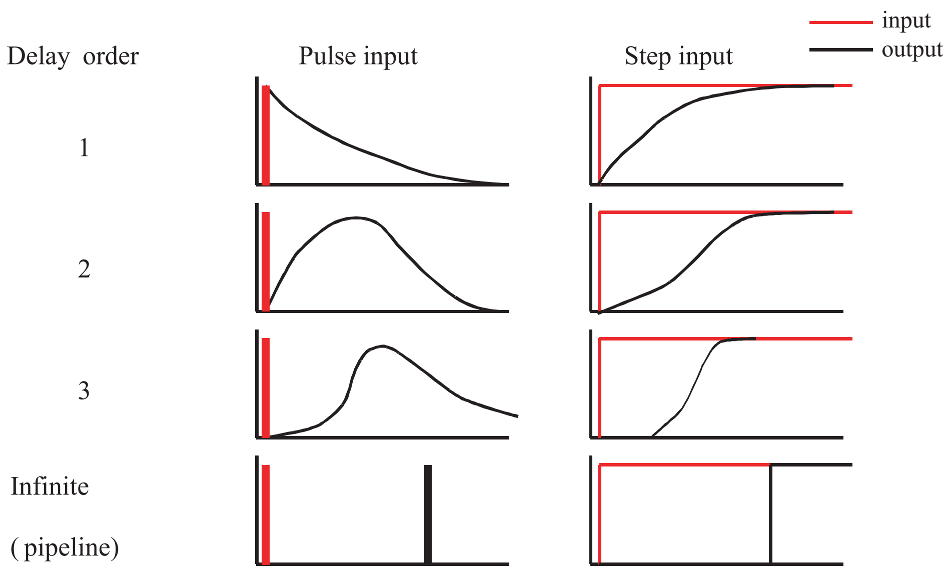


Figure 3.23. Delay summary.

Note that an infinite-order delay (pipeline) replicates the input signal (here a fortieth-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-

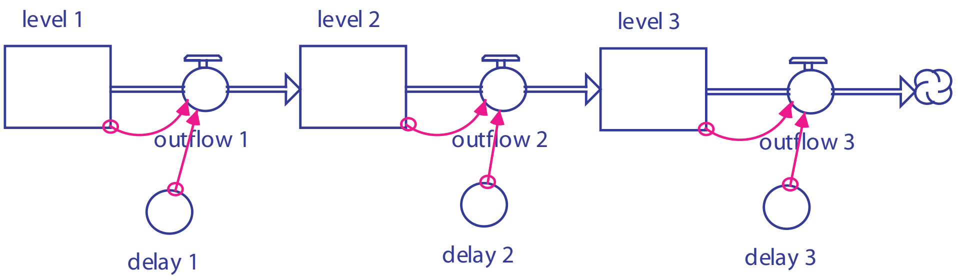


Figure 3.24. Cascaded delay (third order). Equations:

outflow 1 = level 1/delay 1

outflow 2 = level 2/delay 2

outflow 3 = level 3/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.

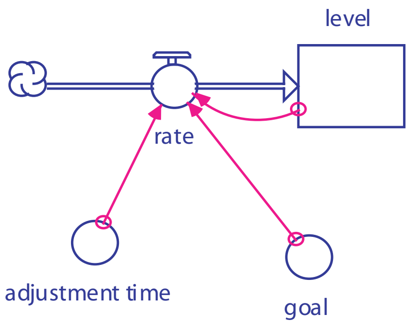


Figure 3.25. Balancing feedback structure. Equation:

rate = (goal – level)/adjustment time

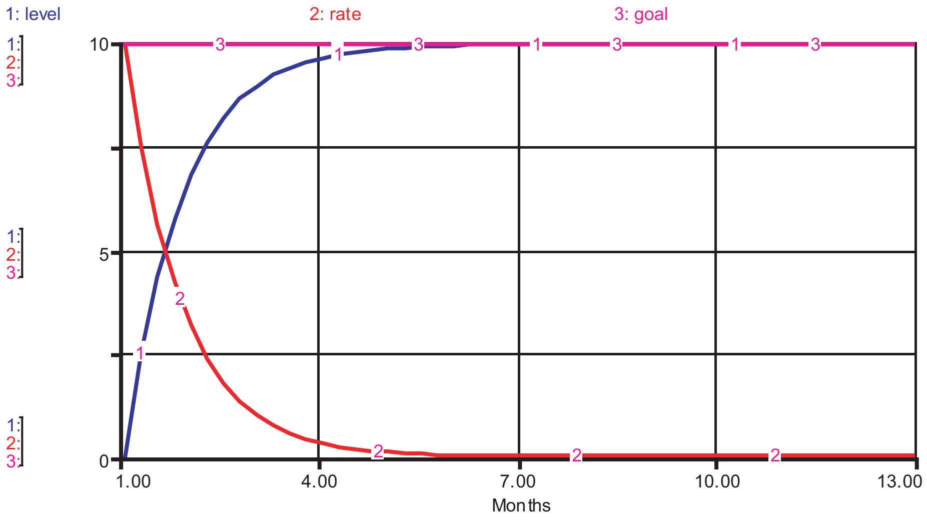


Figure 3.26. Goal-seeking balancing feedback (positive goal).

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,

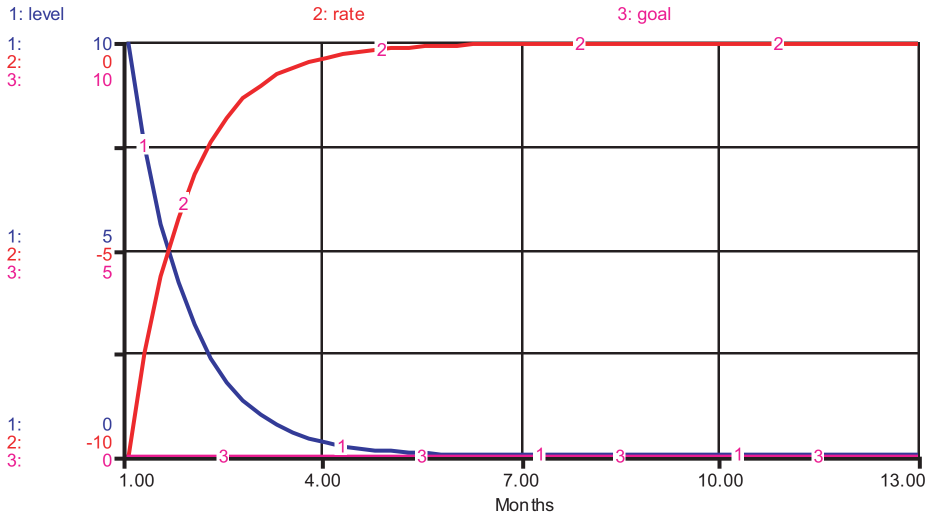


Figure 3.27. Goal-seeking balancing feedback (zero goal).

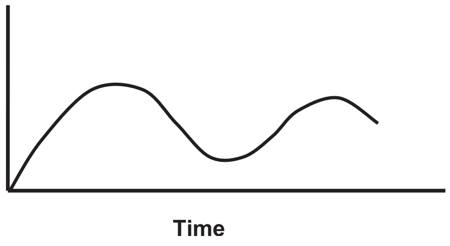


Figure 3.28. Second-order balancing feedback response.

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

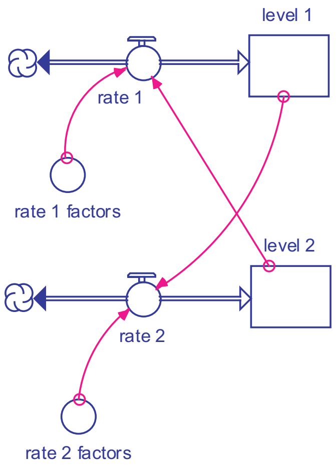


Figure 3.29. Oscillating infrastructure.

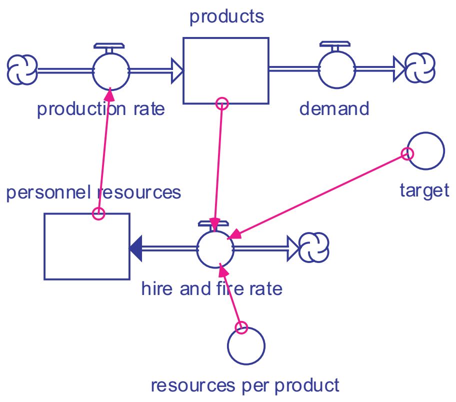


Figure 3.30. Oscillating production and hiring—simplified. Equation:

hire and fire rate = (target – products) · resources per product

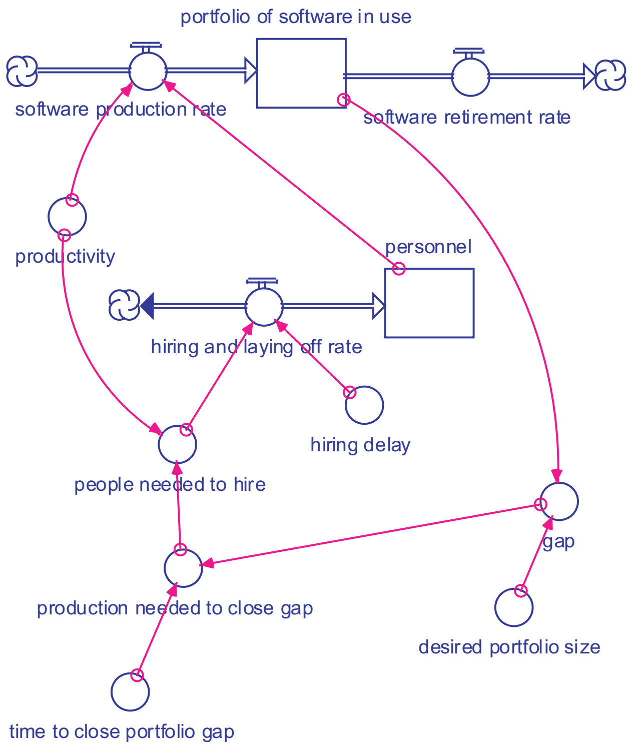


Figure 3.31. Oscillating production and hiring. Equations:

software production rate = personnel · productivity

hiring and laying off rate = people needed to hire/hiring delay

production needed to close gap = gap/time to close portfolio gap

people needed to hire = production needed to close gap/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.

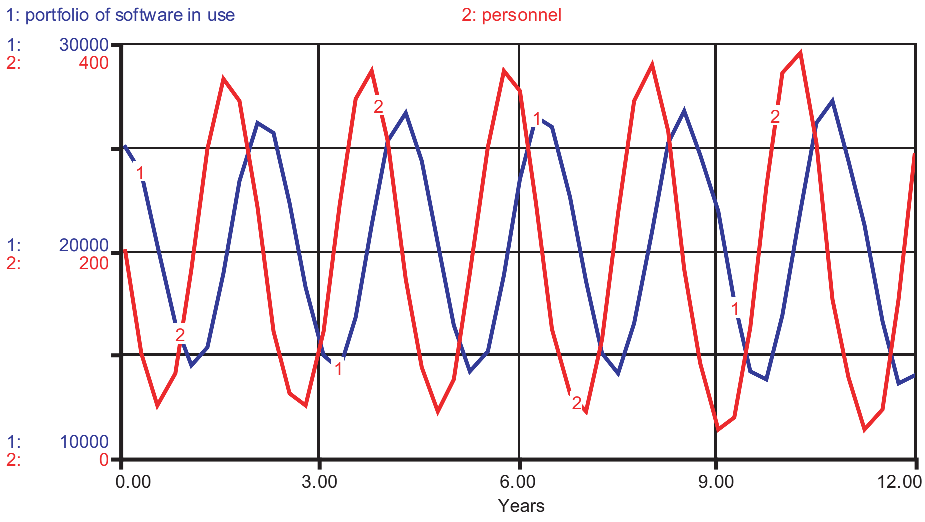


Figure 3.32. Oscillating portfolio and personnel dynamics.

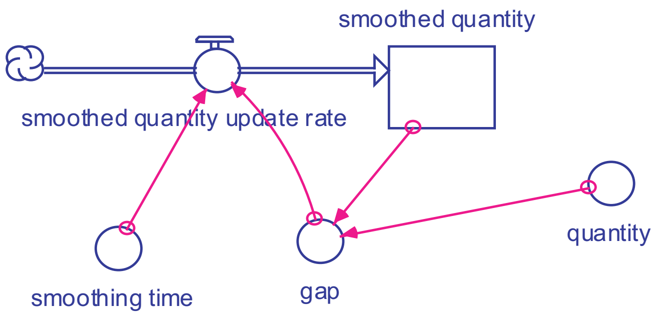


Figure 3.33. Information smoothing structure. Equations:

smoothed quantity update rate = gap/smoothing time

gap = quantity – 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-

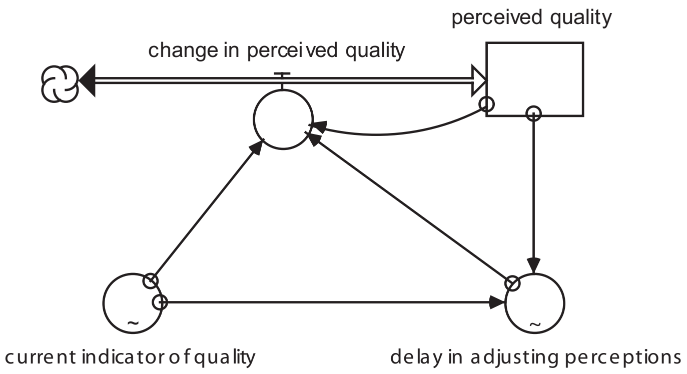

perceived quality
change in perceived quality
current indicator of quality
delay in adjusting perceptions

Figure 3.34. Perceived quality information smoothing. Equation:

change in perceived quality = (current indicator of quality-perceived quality)/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

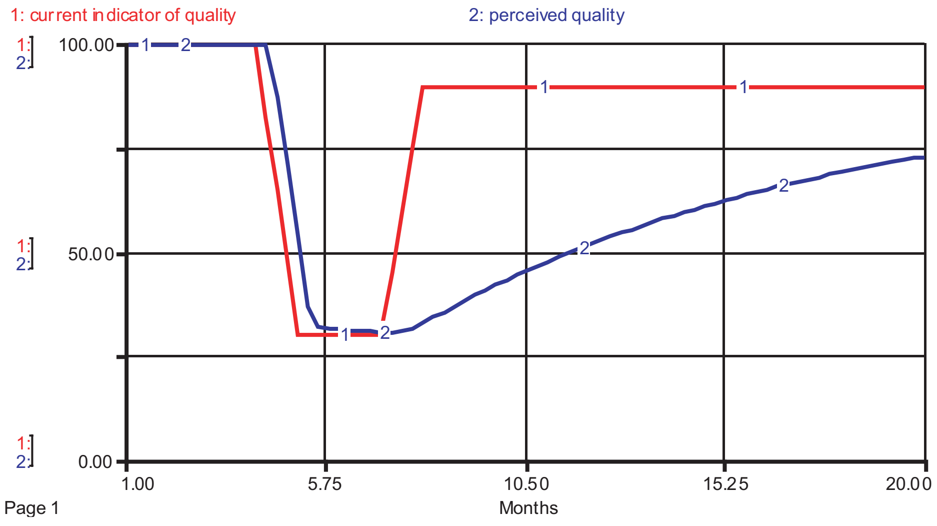


Figure 3.35. Output of perceived quality information smoothing.

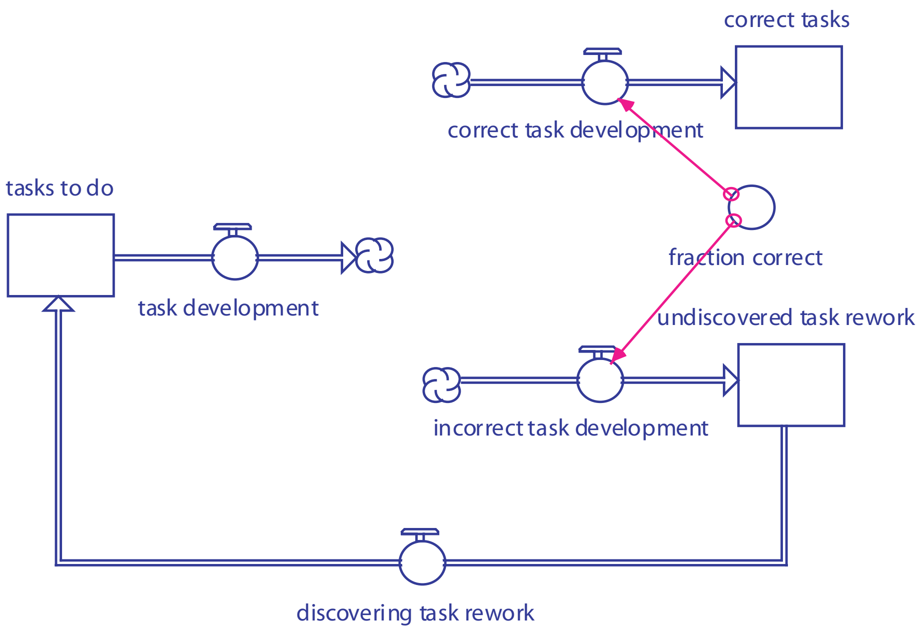


Figure 3.36. Production and rework structure. Equations:

correct task development = task development · fraction correct

incorrect task development = task development · (1 – 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.

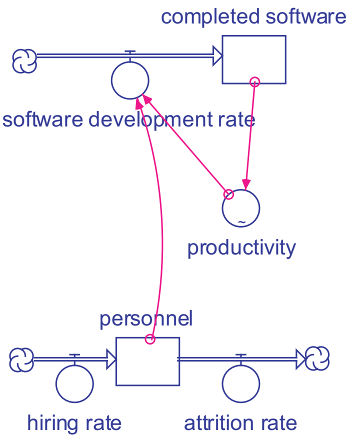


Figure 3.37. Production infrastructure with task and personnel chains. Equation:

software development rate = personnel · 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

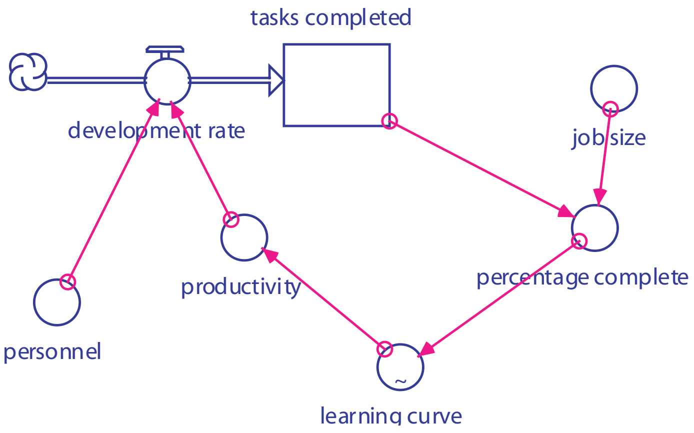


Figure 3.38. Learning curve. Equations:

learning curve = f(percentage complete)

productivity = f(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

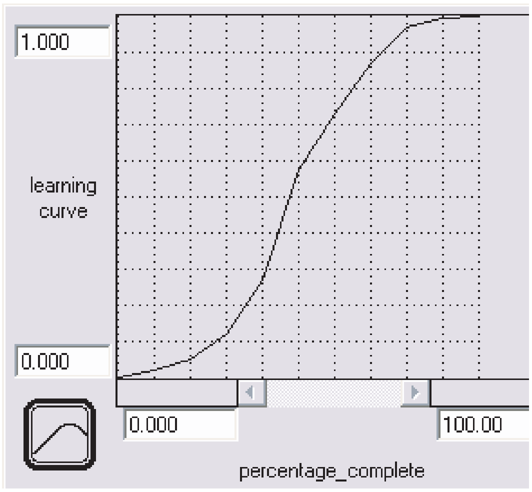


Figure 3.39. Learning curve function.

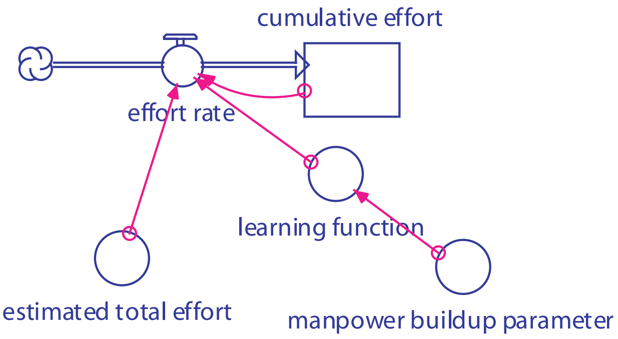


Figure 3.40. Rayleigh curve generator. Equations:

effort rate = learning function · (estimated total effort – cumulative effort)

learning function = manpower buildup parameter · 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

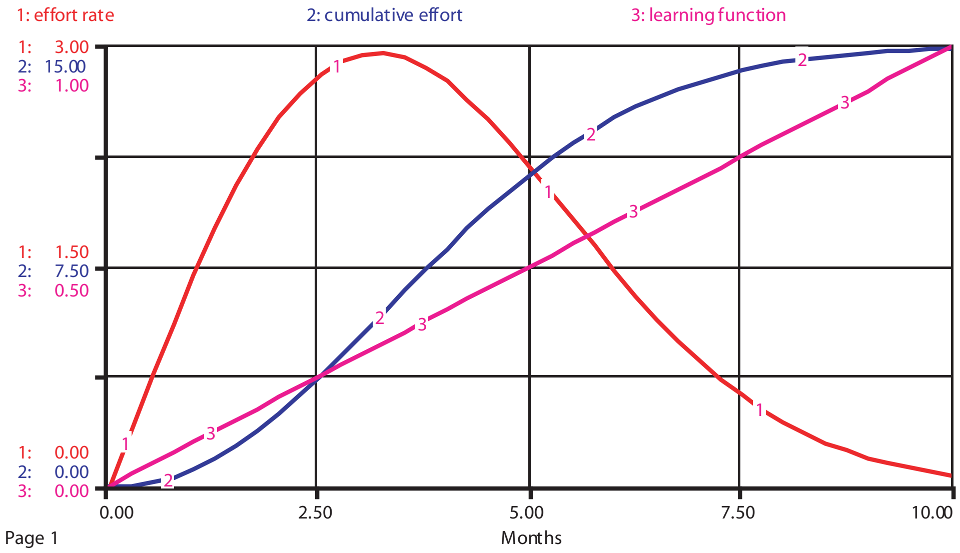


Figure 3.41. Rayleigh curve outputs.

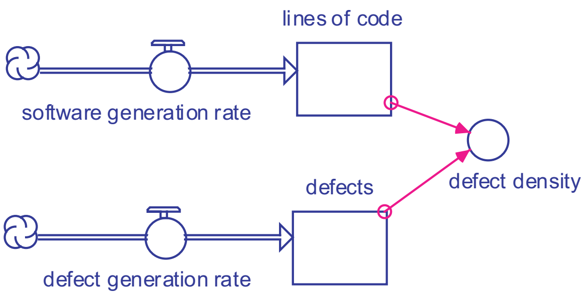


Figure 3.42. Attribute tracking infrastructure (defect density example). Equation:

defect density = defects/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

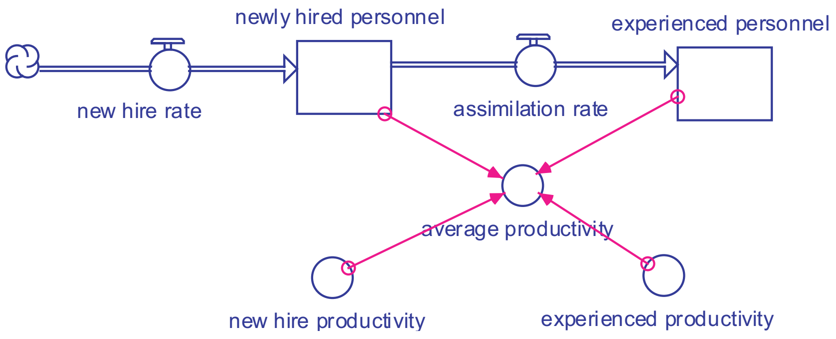


Figure 3.43. Attribute averaging infrastructure (software productivity example). Equation:

average productivity = [(newly hired personnel · new hire productivity) + (experienced personnel · experienced productivity)]/(newly hired personnel + 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-

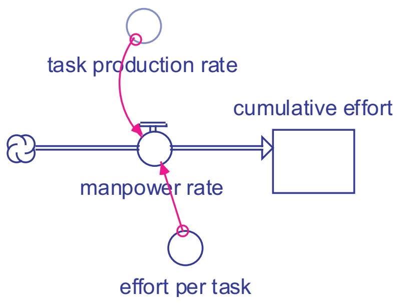


Figure 3.44. Effort expenditure coflow. Equation:

manpower rate = task production rate · effort per task

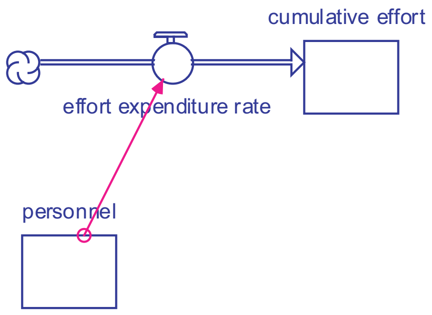


Figure 3.45. Effort expenditure linked to personnel level. Equation:

effort expenditure rate = personnel

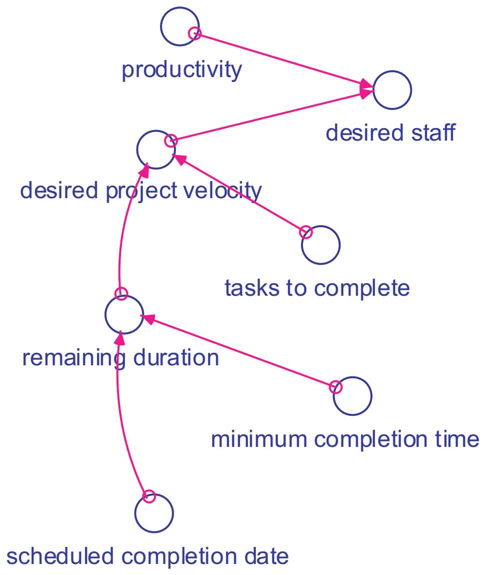


Figure 3.46. Desired staff structure. Equations:

desired staff = desired project velocity/productivity

desired project velocity = tasks to complete/remaining duration

remaining duration = 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

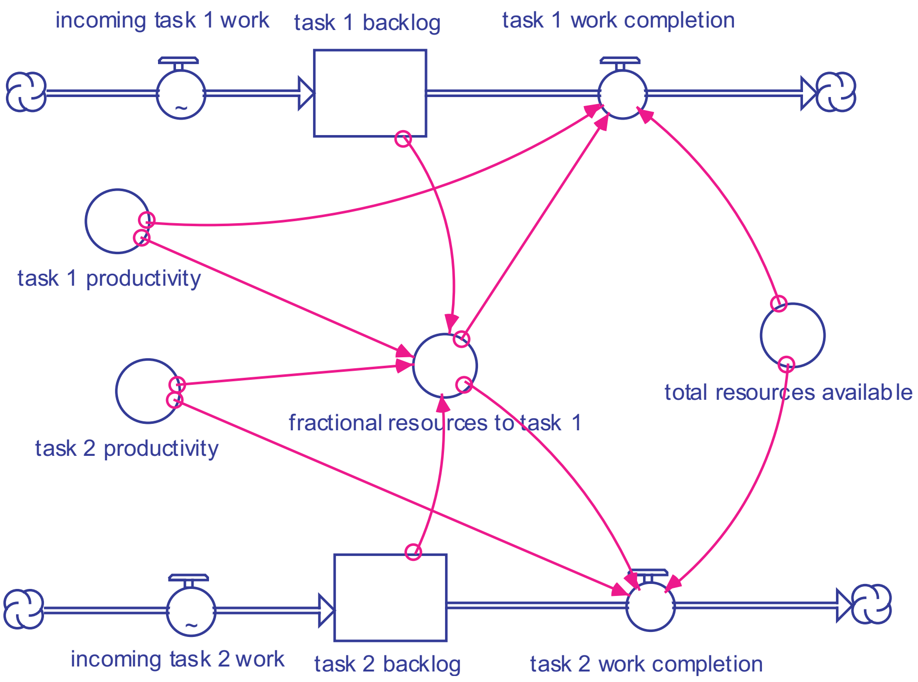


Figure 3.47. Resource allocation infrastructure. Equations:

fractional resources to task 1 = (task 1 backlog/task 1 productivity)/
[(task 1 backlog/task 1 productivity) + (task 2 backlog/task 2 productivity)]

task 1 work completion = (total resources available · fractional resources to task 1)
· task 1 productivity

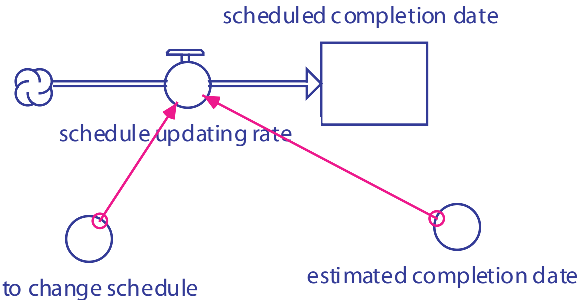


Figure 3.48. Scheduled completion date. Equation:

schedule updating rate = (estimated completion date
– scheduled completion date)/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 implic-

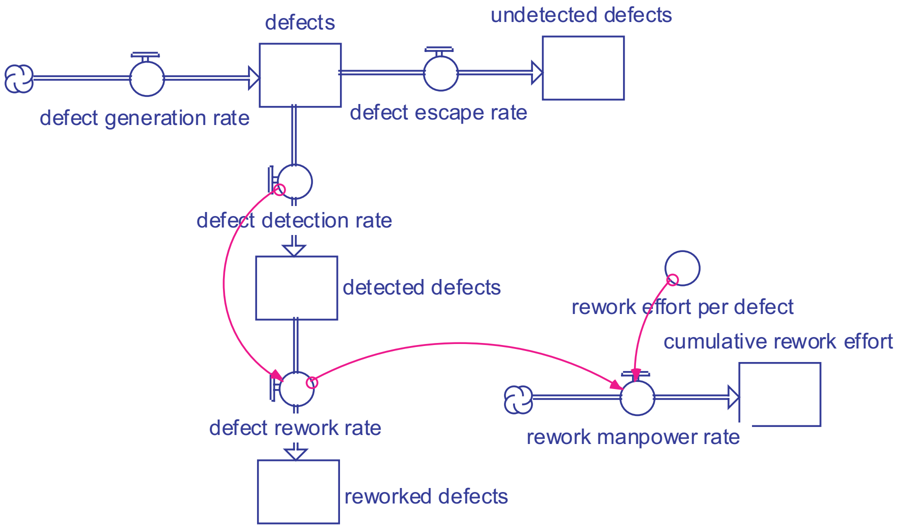


Figure 3.49. Rework all defects right away structure. Equation:

rework manpower rate = defect rework rate · 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-

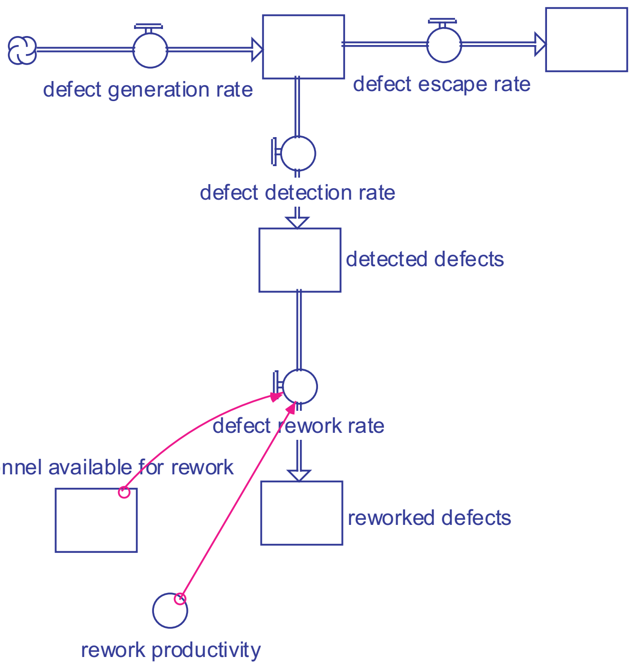


Figure 3.50. Personnel-constrained rework structure. Equation:

defect rework rate = personnel available for rework · 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.

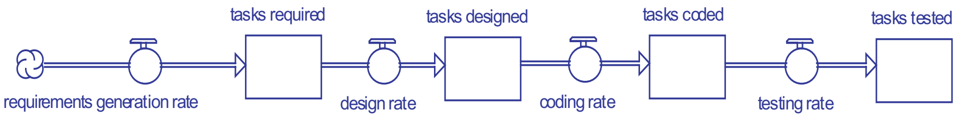


Figure 3.51. Product transformation chain.

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.

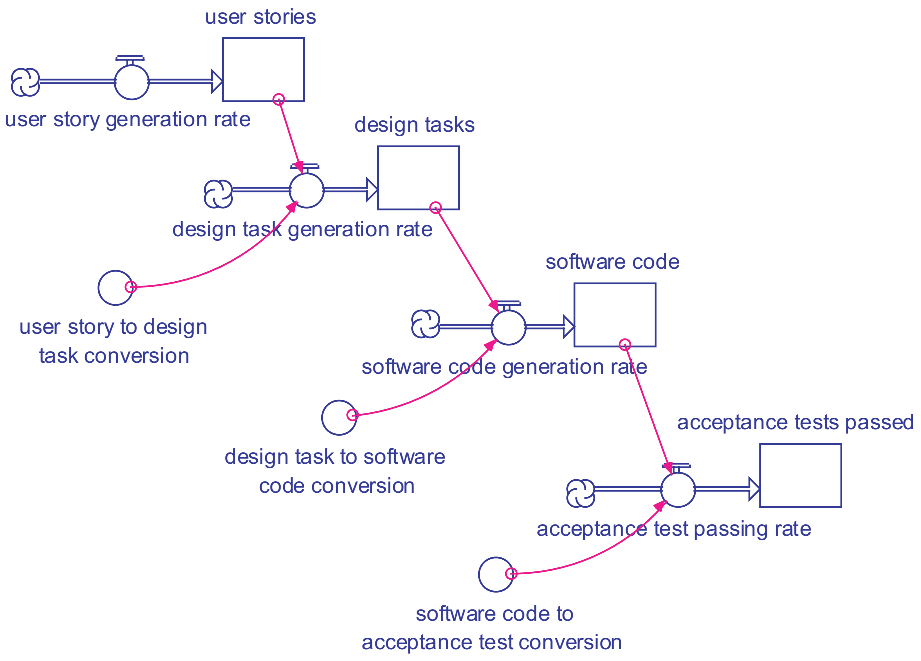

user stories
user story generation rate
design tasks
design task generation rate
user story to design task conversion
software code
software code generation rate
design task to software code conversion
acceptance tests passed
acceptance test passing rate
software code to acceptance test conversion

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

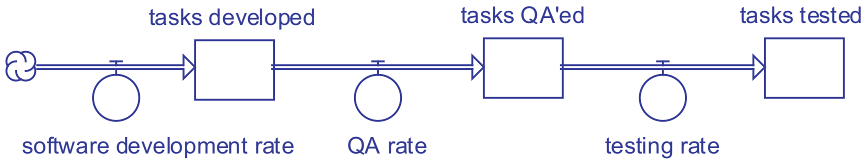


Figure 3.53. Product chain from [Abdel-Hamid, Madnick 1991].

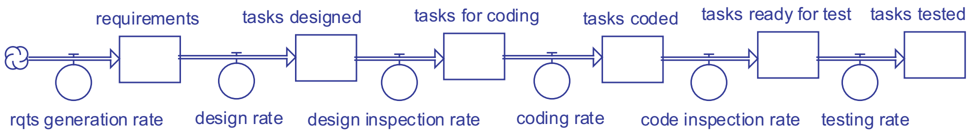


Figure 3.54. Product chain from [Madachy 1994b].

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,

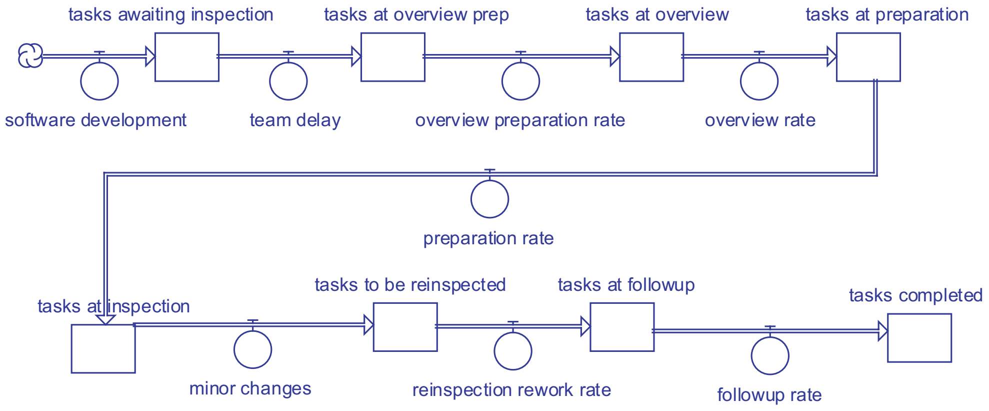


Figure 3.55. Product chain from [Tvedt 1995].

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.

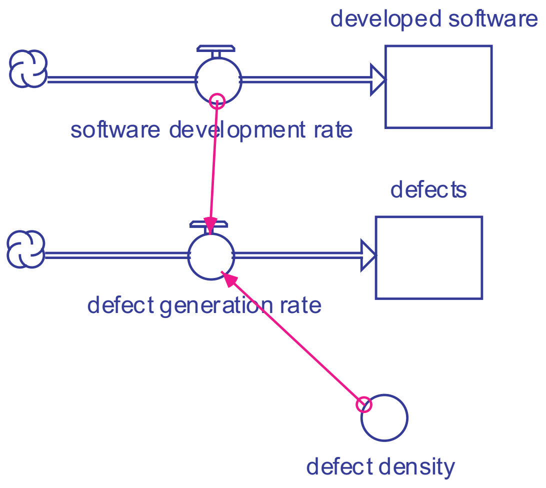


Figure 3.56. Defect generation coflow. Equation:

defect generation rate = software development rate · 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.

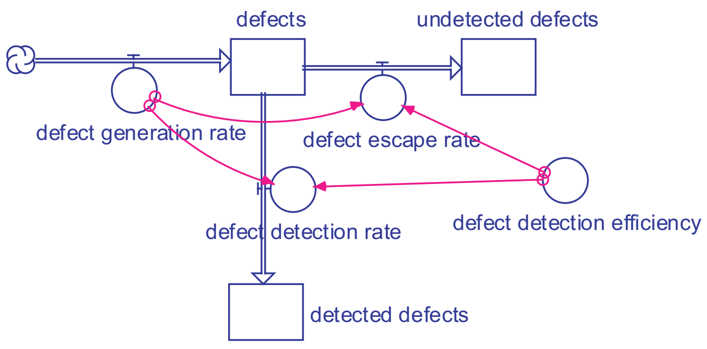


Figure 3.57. Defect detection structure. Equations:

defect escape rate = (1 – defect detection efficiency) · defect generation rate

defect detection rate = defect detection efficiency · defect generation rate

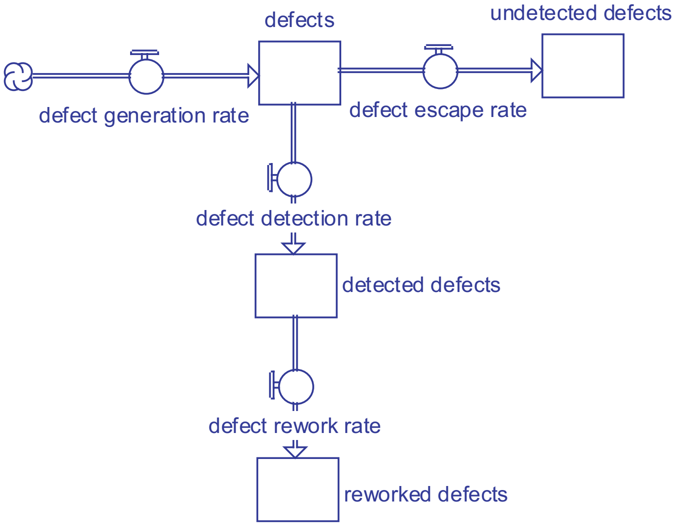


Figure 3.58. Defect detection and rework chain.

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.

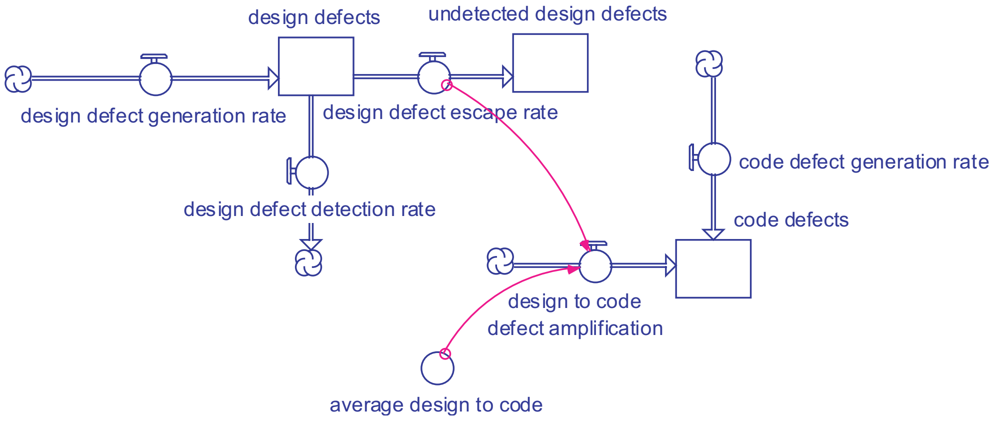

design defects
undetected design defects
design defect generation rate
design defect escape rate
design defect detection rate
code defect generation rate
code defects
design to code
defect amplification
average design to code

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

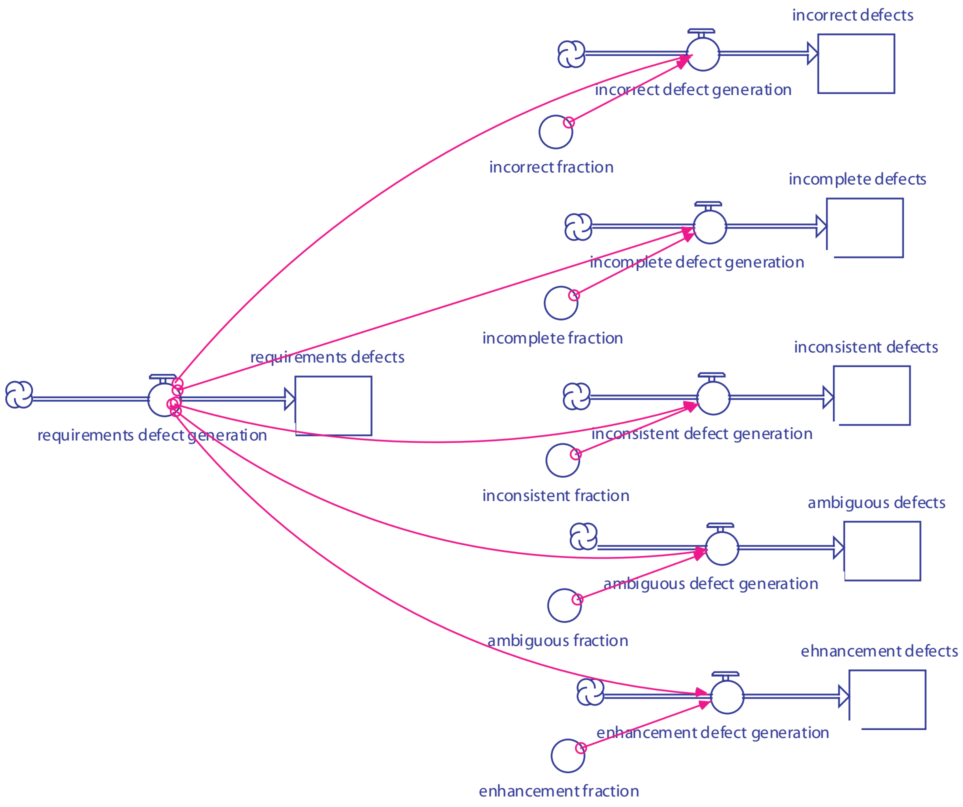


Figure 3.60. Example streams for ODC requirements defects. Equations:

incorrect defect generation = incorrect fraction · requirements defect generation

incomplete defect generation = incomplete fraction · requirements defect generation

inconsistent defect generation = inconsistent fraction · requirements defect generation

ambiguous defect generation = ambiguous fraction · requirements defect generation

enhancement defect generation = enhancement fraction · 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

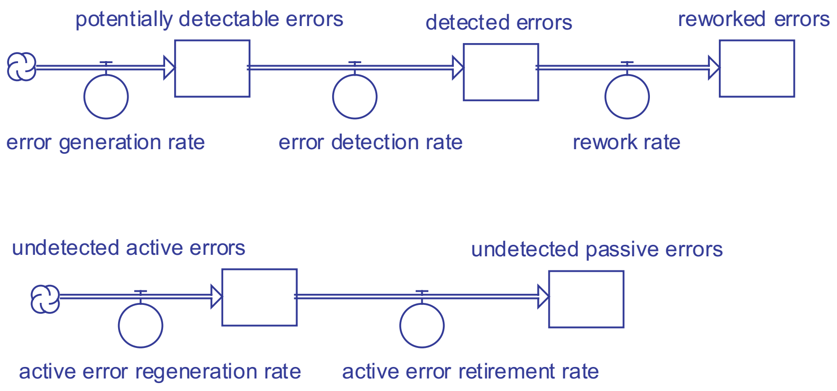

potentially detectable errors
detected errors
reworked errors
error generation rate
error detection rate
rework rate
undetected active errors
undetected passive errors
active error regeneration rate
active error retirement rate

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

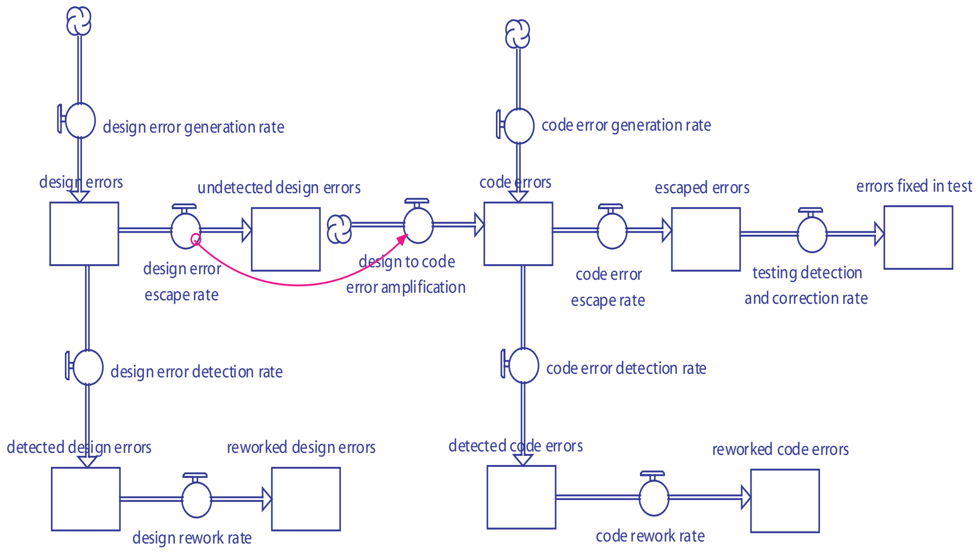


Figure 3.62. Defect chains from [Madachy 1994b].

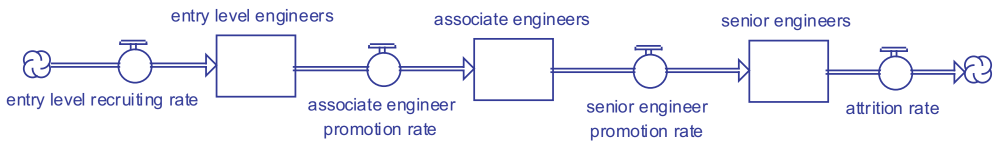

entry level engineers
associate engineers
senior engineers
entry level recruiting rate
associate engineer promotion rate
senior engineer promotion rate
attrition rate

Figure 3.63. Personnel chain (three levels).

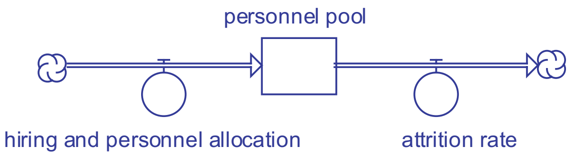


Figure 3.64. Personnel pool from [Madachy 1994b].

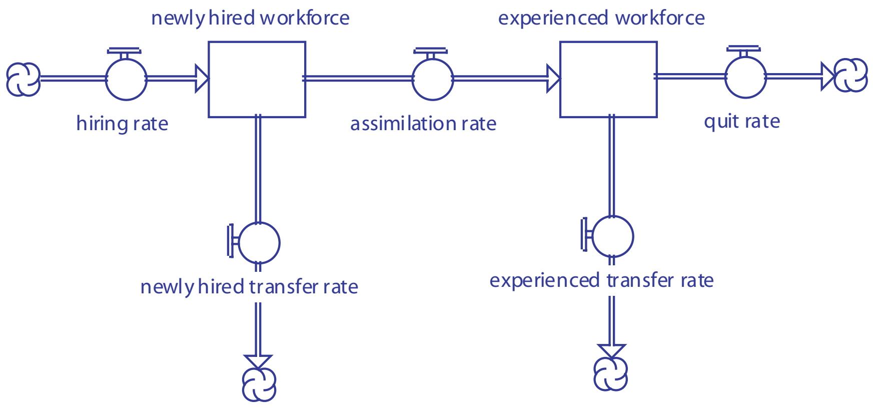


Figure 3.65. Personnel pool from [Abdel-Hamid, Madnick 1991].

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 germaine 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.

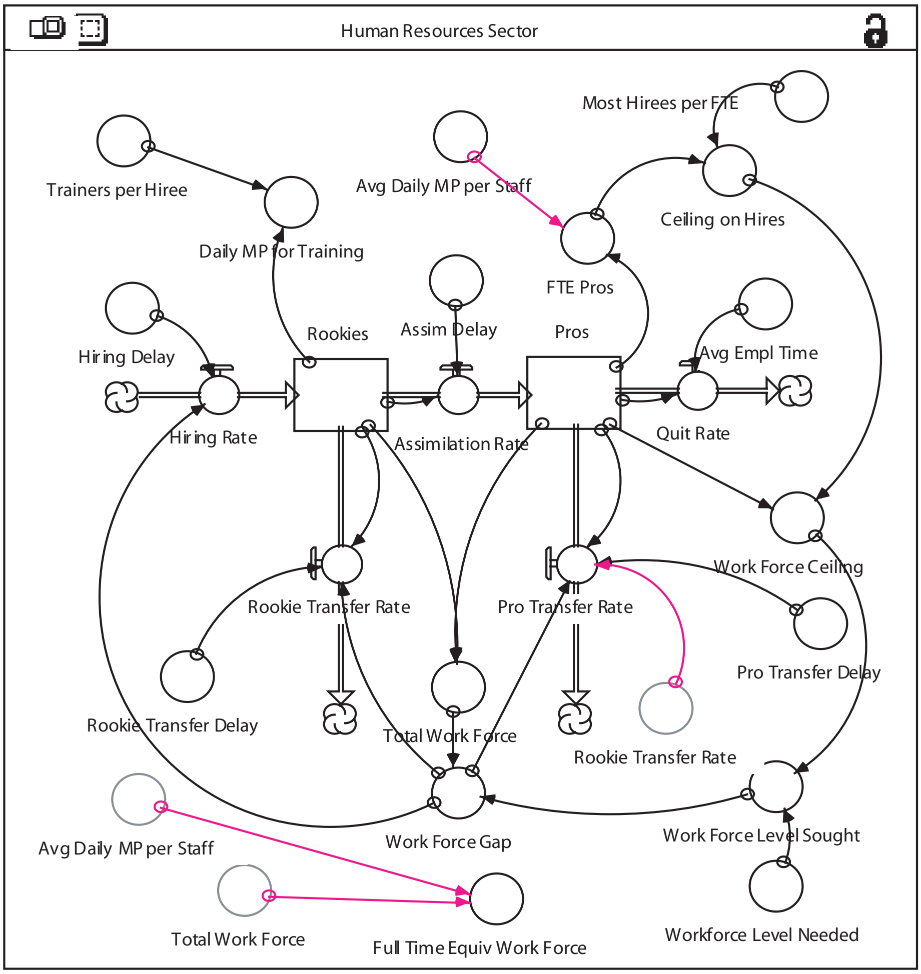
Some decision structures relevant to software projects were shown, including policies to allocate staff, adjust project goals/estimates as a project progresses, and defect rework policies. All of these structures can be reinterpreted to simulate policies for different decision contexts.

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.

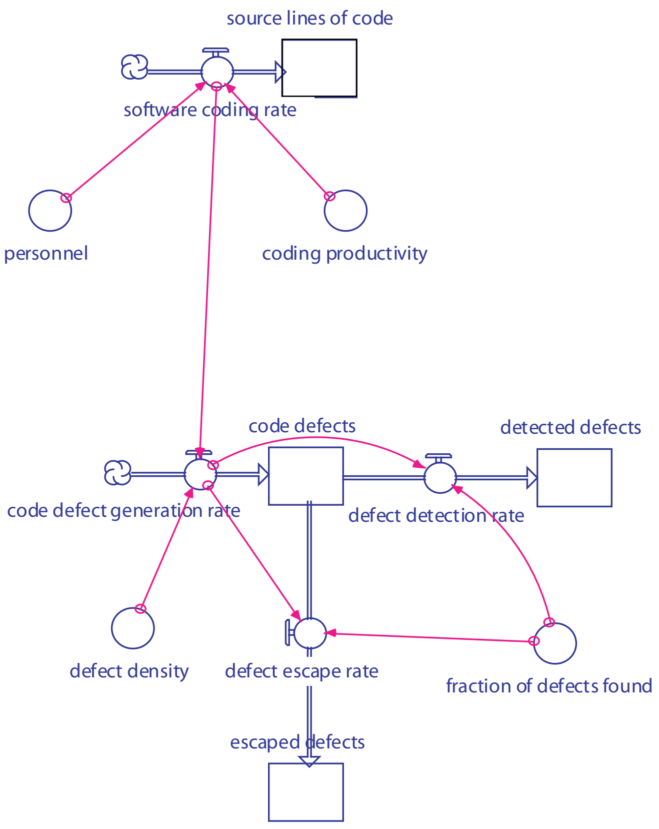
3.8 EXERCISES

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

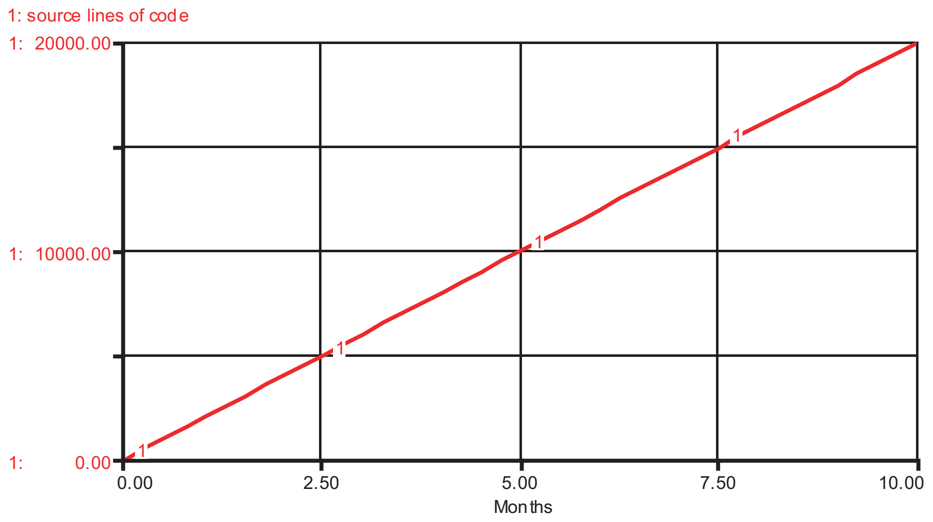
3.1. Below is the personnel structure from the Abdel-Hamid project model. Identify and trace out an example of a feedback loop in it. Each involved entity and connection must be identified.



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.

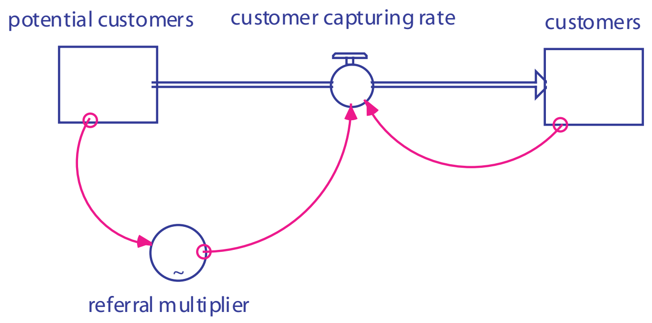


```
□ code_defects(t) = code_defects(t - dt) + (code_defect_generation_rate - defect_detection_rate -
  defect_escape_rate) * dt
  INIT code_defects = 0
   INFLOWS:
     code_defect_generation_rate = software_coding_rate*defect_density
   OUTFLOWS:
     defect_detection_rate = code_defect_generation_rate*fraction_of_defects_found
     defect_escape_rate = code_defect_generation_rate*(1-fraction_of_defects_found)
□ detected_defects(t) = detected_defects(t - dt) + (defect_detection_rate) * dt
  INIT detected_defects = 0
   INFLOWS:
     defect_detection_rate = code_defect_generation_rate*fraction_of_defects_found
□ escaped_defects(t) = escaped_defects(t - dt) + (defect_escape_rate) * dt
  INIT escaped_defects = 0
   INFLOWS:
     defect_escape_rate = code_defect_generation_rate*(1-fraction_of_defects_found)
□ source_lines_of_code(t) = source_lines_of_code(t - dt) + (software_coding_rate) * dt
  INIT source_lines_of_code = 0
   INFLOWS:
     software_coding_rate = coding_productivity*personnel
○ coding_productivity = 200
○ defect_density = .05
○ fraction_of_defects_found = .5
○ personnel = 10
```

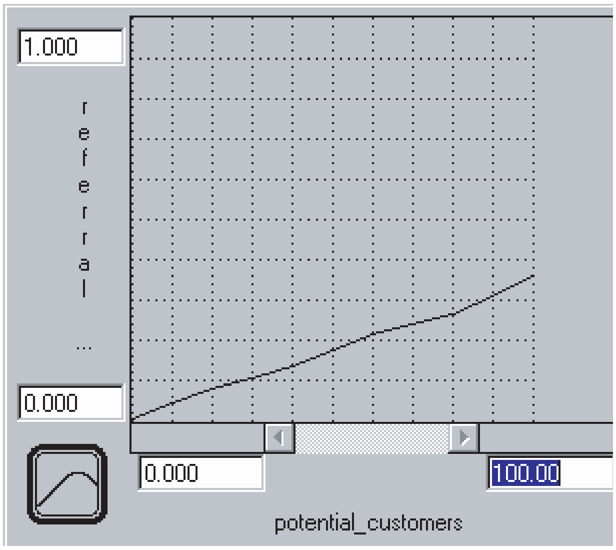


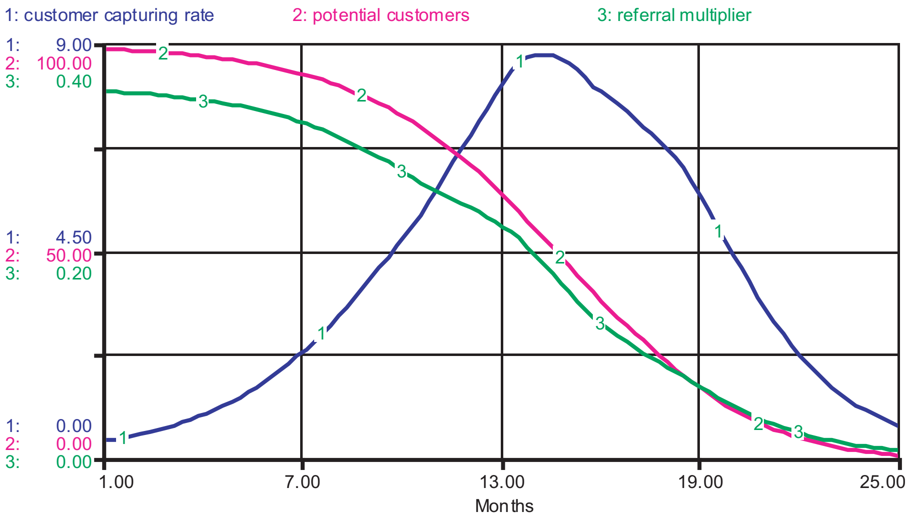
Sketch the three graphs of code defects, defect detection rate, and detected defects.

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.



```
□ customers(t) = customers(t - dt) + (customer_capturing_rate) * dt
  INIT customers = 1
  INFLOWS:
      customer_capturing_rate = customers*referral_multiplier
□ potential_customers(t) = potential_customers(t - dt) + (- customer_capturing_rate) * dt
  INIT potential_customers = 99
  OUTFLOWS:
      customer_capturing_rate = customers*referral_multiplier
  referral_multiplier = GRAPH(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.

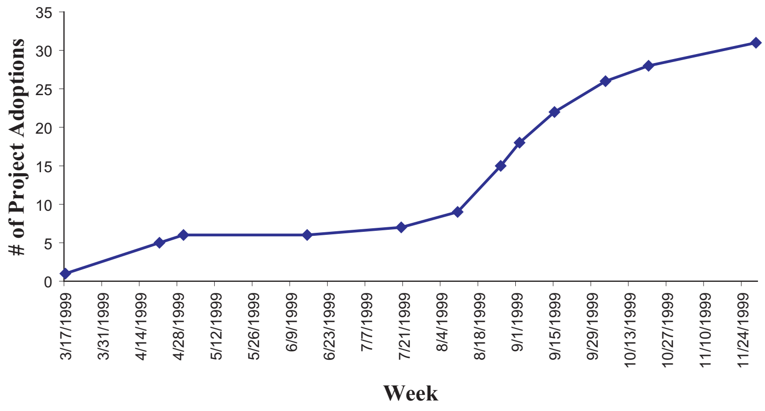
b. Vary the productivities.

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.