Chapter 3

Model Description

3.1 Introduction

Painting a complete picture of the Product Development Project Model requires descriptions from several perspectives. In this chapter I begin this process by depicting the model from three relatively context-free vantage points. First the model itself is framed by defining its boundaries and level of aggregation. In the second and largest portion of this chapter the model's inner structure is depicted in increasing detail by describing phases, subsystems, and sectors. A table of the foundations of the important model structures completes the model structure section. An initial description and investigation of model behavior is the third vantage point. The sensitivity tests identify the parameters which deserve special attention as the model is applied to specific contexts.

Other perspectives are applied to expand the description and investigation of the model in subsequent chapters. A signal processing model of a portion of the Product Development Project Model is described and used to illustrate an alternative modeling approach in chapter 4. The Python Development Project forms a specific context for the application of the model and its use for policy analysis in chapter 5.

3.2 Model Boundary and Level of Aggregation

The model's scope and focus are reflected in the model boundary. Figure 3-1 delineates the primary features included (endogenous), assumed (exogenous), and excluded (ignored) from the Product Development Project Model. Among the most important model boundaries are the edges of a single development project. This focuses the research on the inner workings of development projects. While the interaction of projects in a multiple-project development environment can be important (Wheelwright and Clark, 1992; Wheelwright and Sasser, 1991) an improved understanding of the structure and behavior of single development projects is needed as a basis for investigating multiple projects. Such as single project model can be replicated to build a multiple-project model to investigate project interactions.

A second important boundary assumption is a stable development environment, process, and organization throughout the project life. An example of an assumption about a stable development process is the use of exogenous constants to describe the average duration required to complete development activities. These values and function do not change during the simulation. The potential impacts of relaxing the boundary assumptions are discussed in the conclusions.

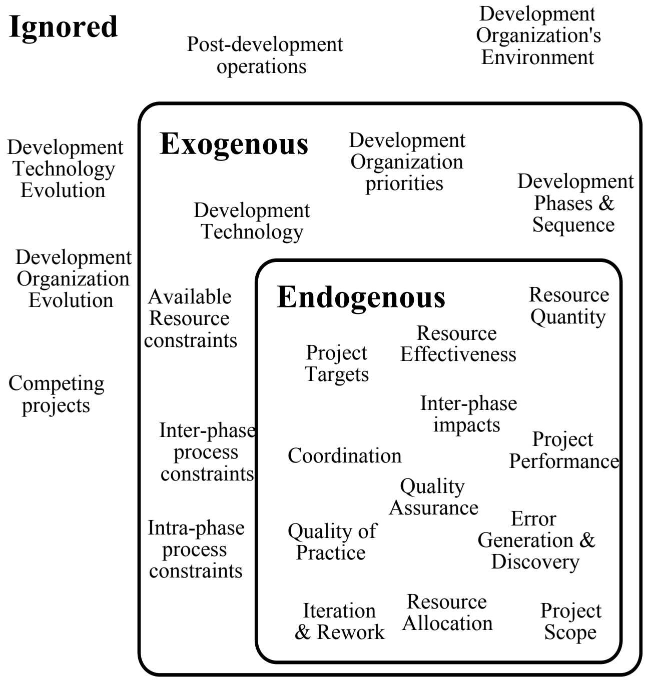

Ignored
Post-development operations
Development Organization's Environment
Development Technology Evolution
Development Organization Evolution
Competing projects
Exogenous
Development Organization priorities
Development Phases & Sequence
Development Technology
Available Resource constraints
Inter-phase process constraints
Intra-phase process constraints
Endogenous
Resource Quantity
Resource Effectiveness
Project Targets
Inter-phase impacts
Coordination
Project Performance
Quality Assurance
Quality of Practice
Error Generation & Discovery
Iteration & Rework
Resource Allocation
Project Scope

Figure 3-1: Product Development Project Model Boundary Diagram

Within the model boundary the level of aggregation focuses the research and model purpose. For example the model simulates multiple interdependent development phases within a project.

Phases are defined around similar development activities such as product definition, design, testing, and installation. Examples of a single development phase include the preparation of construction drawings in a real estate development project, the writing of software code and the testing of product prototypes.

Another important level of aggregation assumption concerns the fundamental units which flow through projects. I assume that these units are "development tasks". Conceptually a development task is an atomic unit of development work. Examples of development tasks might include the selection of a pump, writing a line of computer code and installing a steel beam. The unit of work used to describe a development task may differ among project phases. For example a product definition phase might use product specifications as the basis for tasks whereas the design phase of the same project might use lines of computer code. Tasks are assumed to be uniform in size and fungible. This assumption becomes more accurate as task size decreases. Therefore relatively small pieces of development work are selected as tasks. Fungibility is an inherent characteristic of some development tasks (e.g. the delivery and placement of soils for a roadbed). Many other development phases have interdependent but fungible tasks (e.g. software code as in Abdel-Hamid, 1984). The Product Development Project Model provides for the description of task dependencies both within and among phases, as described subsequently in the Development Tasks sector. Tasks are also assumed to be small enough to be flawed or correct but not partially flawed. This assumption also becomes more accurate as task size becomes smaller. These assumptions concerning tasks help identify divisions among development phases and development tasks.

I have disaggregated development within each phase into four activities: basework, quality assurance, rework and coordination. Basework is the completion of a development task the first time. Subsequent completions which are required to correct flaws or iterate for quality are referred to as Rework. Rework includes all forms of iteration regardless of cause. The search for flaws is Quality Assurance (QA) or Inspection. Flaws include errors which must be corrected for product functionality and optional improvements for quality. Coordination is the integration of the product development project among phases due to releasing and inheriting flawed tasks from other phases.

Resources for each phase have been aggregated into a single labor type. This reflects an assumption that other development resources such as testing machines and administrative support are used in proportion to development labor. The primary model assumptions concerning the level of aggregation are listed below.

- Development projects consist of a network of dependent phases under a project management structure.
- The progress of a development phase can be reflected in the flow of development tasks through and among development phases.
- Development tasks are small, uniform in size and either flawed or correct but not partially flawed.
- Development occurs through four activities: basework, quality assurance, rework, and coordination.
- The repair of flawed tasks for basic product functionality and optional iteration for quality have similar characteristics and can be modeled together as Rework.
- Different resource types can be aggregated into a single labor group for each development phase.

The Product development Project Model's boundary and level of aggregation focus the research. The internal structure of the model which simulates a development project is described in the next section.

3.3 Model Structure

3.3.1 Introduction

Operationally the model is a set of nonlinear ordinary differential equations. Appendix 3.1: Model Equations provides a complete listing of those equations. The equations are arrayed to allow the simulation of a flexible number of development phases and include many equations to manage the modeling of multiple phases. This results in a high number of model parameters and relationships in a network which is too complex to illustrate with multiple phase diagrams. However diagrams of a single phase will be used for model description purposes with explanations of deviations for multiple phases. Definitions of the model parameters used in the model equations are given in Appendix 3.2: Model Parameters.

The model consists of a set of interrelated development phases and a set of project management features. Each phase is customized to reflect a specific stage of product development. A phase dependency network describes the forward flow of work through the project. Figure 3-2 shows a simple phase dependency network for a real estate development project.

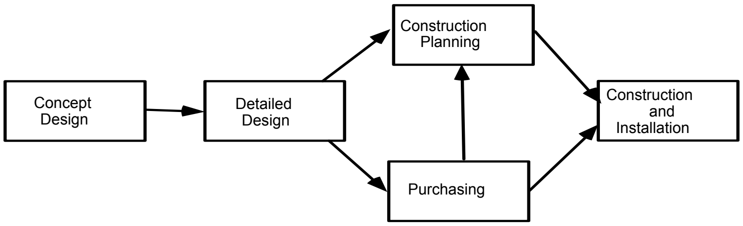


Figure 3-2: A Phase Dependency Network for a Development Project

Project phases are linked in several ways:

- Work flows in upstream phases constrain progress in dependent downstream phases. The locations of these constraints are shown by the arrows in the project's phase dependency network (Figure 3-1).
- Errors inherited by downstream phases from upstream phases corrupt downstream work.
- Inherited errors that are discovered in downstream phases are returned for correction to the phase where the error was generated.
- Coordination with other phases is required by discovering inherited errors or having errors generated within a phase discovered by downstream phases.
- Completion and expected completion dates of phases influence the project deadline. The project deadline in turn influences phase deadlines.
- Poor schedule, quality, and cost performance in any phase increases the impacts of non conformance to the project targets. Those project level impacts influence individual phase targets.

The basis for the model structure is described below and summarized in Table 3-2, located after the model subsystem descriptions.

3.3.2 Model Subsystems

The model is relatively large with approximately twenty five stocks for each development phase and five project management stocks. For descriptive purposes the model has been disaggregated into five subsystems (Figure 3-3): process structure, scope, resources, targets, and performance. Subsystems have been further disaggregated into sectors. Subsystems and sectors are tightly linked through shared parameters. For clarity these parameters are shown in each sector diagram where the parameter is used.

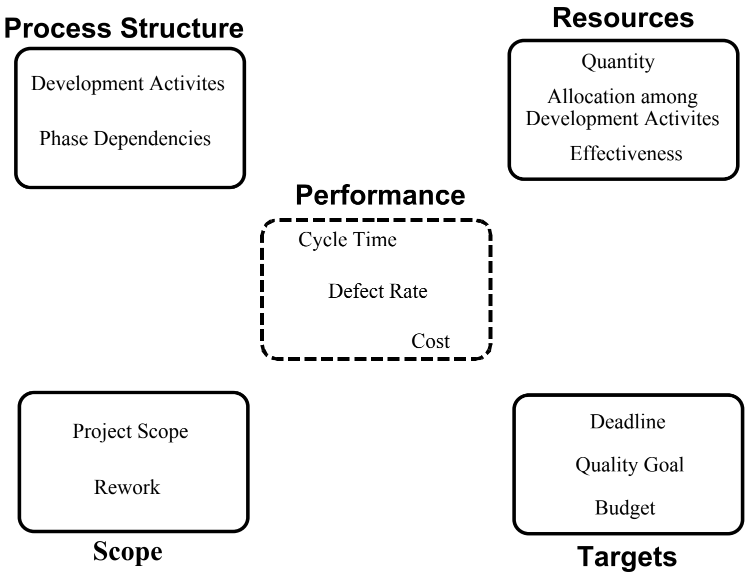


Figure 3-3: Model Subsystems

3.3.3 The Process Structure and Scope Subsystems

3.3.3.1 Introduction

Impacts of the development process and the amount of project work on performance are modeled with the Process Structure and Scope subsystems. The Development Tasks subsystem describes the nature of the development process, while the Scope subsystem simulates the original project scope and increases due to rework. These subsystems include the Development Tasks, Internal Errors, Upstream Errors and Downstream Errors sectors.

3.3.3.2 The Development Task Sector

The process structure portion of the model is one of the most important contributions of this research to model structure. The Development Task sector is the core of how the Product Development Project Model describes development processes. A diagram of the complete Development Tasks Sector in shown in Figure 3-15. One of the most important interactions of development process and resources occurs at the four development activities in the Development Tasks sector. Each activity proceeds at the minimum pace allowed by its process and resources, as modeled with the following equations:

```
QA_inspection_rate(Phase)=MIN(QA_Process_Limit(Phase),QA_Labor_Limit(Phase))

Rework(Phase)=MIN(RW_Process_Limit(Phase),RW_Labor_Limit(Phase))

Basework(Phase)=MIN(BW_Process_Limit(Phase),BW_Labor_Limit(Phase))

Coordination(Phase)=MIN(Coord_Process_Limit(Phase),Coord_Labor_Limit(Phase))
```

Figure 3-4 illustrates the two feedback loops which are the basis for the process structure. The balancing loop depicts the reduction in the number of tasks available for Basework as work is completed. In this loop the Basework rate is based on the Tasks Available for Basework and the Minimum Basework Duration. An increase in the Basework rate increases the number of Tasks Completed, which decreases the number of Tasks Available for Basework, which reduces the Basework rate. This loop introduces the first of two types of parameters used to describe the development process in a phase, the Minimum Activity Duration. The Minimum Activity Duration is the average time required to complete a task if all required information, materials and resources are available and no flaws are generated. It describes the purest time constraint which the process imposes on progress by answering the question "How fast on average can a task be completed if everything needed is available?" All four development activities (basework, quality assurance, rework and coordination) apply this concept. This allows more detailed and accurate descriptions of a development process than by modeling a single development activity. In Figure 3-4 the Minimum Activity Duration is applied to basework. Feedback loops similar to the balancing loop shown in Figure 3-4 are used to describe the role of the Minimum Quality Assurance Duration in discovering flawed tasks, the Minimum Rework Duration in correcting flawed tasks, and the Minimum Coordination Duration in integrating development project phases.

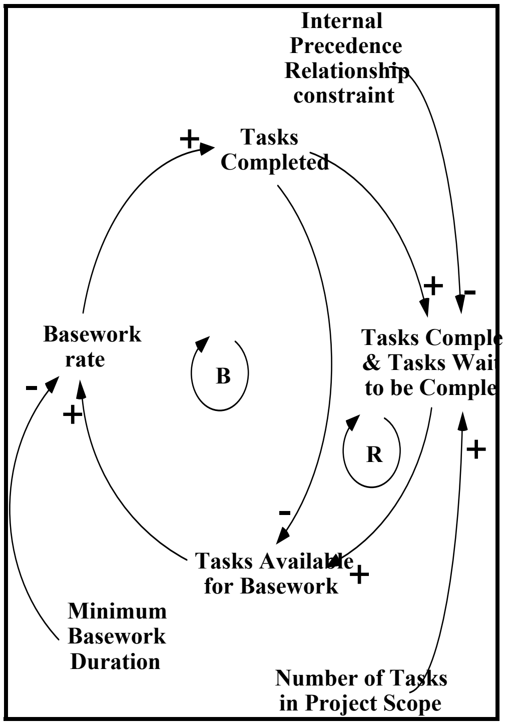
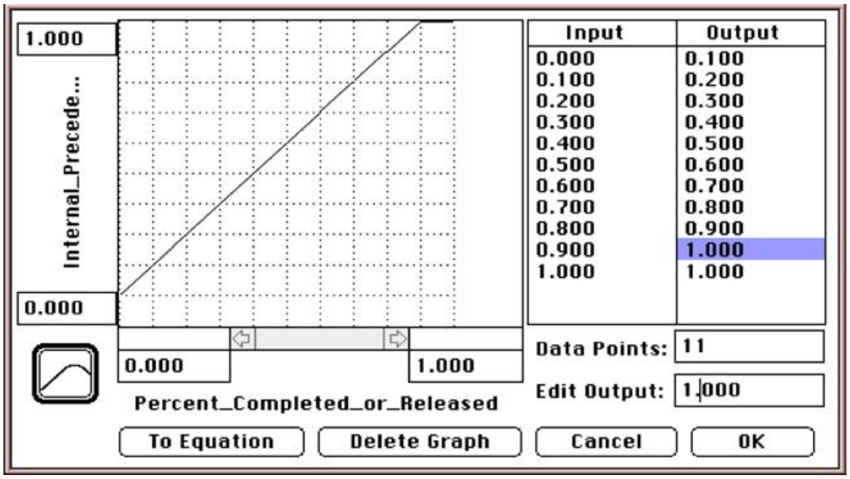

Internal Precedence Relationship constraint
+
Tasks Completed
+
-
Basework rate
B
Tasks Compl & Tasks Wai to be Comple
-
+
R
+
-
Tasks Available for Basework
+
Minimum Basework Duration
Number of Tasks in Project Scope

Figure 3-4: A Development Process Feedback Structure

The reinforcing loop shown in Figure 3-4 models the increase in the number of tasks which are available to the basework activity due to the completion of work. In this loop an increased basework rate raises the number of Tasks Completed, which raises the total number of tasks which can be completed. The total number of tasks which can be completed includes both tasks which have been completed and tasks which are available and waiting to be completed. This quantity of tasks is also dependent on the nature of the development process as described by the process's Internal Precedence Relationship. Increasing the number of Tasks Completed & Waiting to be Completed raises the Tasks Available for Basework and thereby further raises the Basework rate. The reinforcing loop introduces the second type of descriptor of specific development processes, the Internal Precedence Relationship. Internal Precedence Relationships describe the availability of work based solely on the amount of work which has been completed.

Erecting structural steel for a ten story building one story at a time from the ground up provides an example of an Internal Precedence Relationship. Initially 0% is Completed and only the first floor (10%) is Completed or Available to be Completed. When the first floor of steel is erected 10% is Completed and the second floor (another 10%) becomes available, making 20% Completed or Waiting to be Completed. This linear progression continues until the completion of the ninth floor (90% Completed) releases the final floor for completion (100% Completed or Waiting to be Completed). A graphic function which describes this Internal Precedence Relationship is shown in Figure 3-5. The Internal Precedence Relationship describes the available-work constraint which a development process imposes on itself by answering the question "How much work can be completed based upon how much work has been completed?"



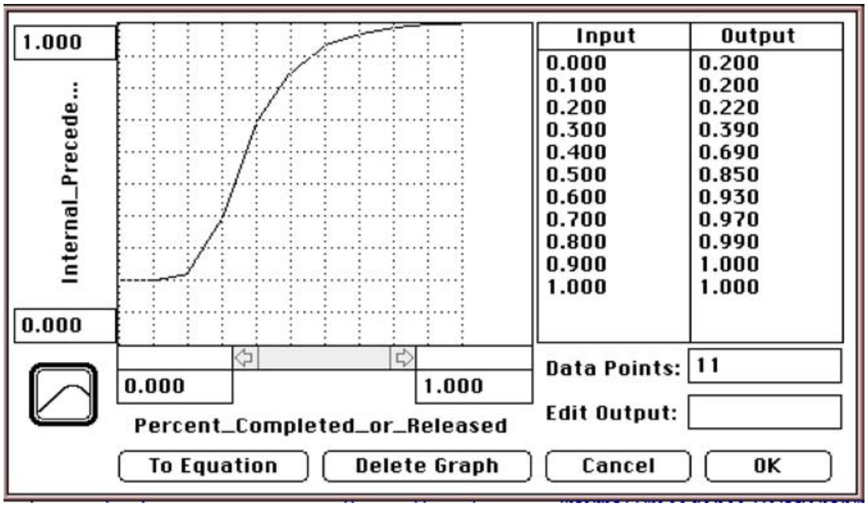
Input	Output
0.000	0.100
0.100	0.200
0.200	0.300
0.300	0.400
0.400	0.500
0.500	0.600
0.600	0.700
0.700	0.800
0.800	0.900
0.900	1.000
1.000	1.000

Data Points: 11

Edit Output: 1.000

Figure 3-5: A Linear Internal Precedence Relationship

An advantage of the system dynamics methodology and the model structure used here is the ability to describe nonlinear Internal Precedence Relationships (Graham 1980). This allows varying degrees of concurrent development within a single phase can be described with Internal Precedence Relationships by altering the shape of the curve in the graphical function. The Internal Precedence Relationship example shown in Figure 3-6 is based upon the design of code for the development of a computer chip. Documentation for the shape of the curve is provided in a later chapter. Initially a few very important blocks of code must be designed. This is the reason for the flat portion of the curve on the left side of the function. Their completion makes the design of many more blocks possible. This is the reasoning behind increasing rate of availability in the left portion of the curve. The increase in available work slows as the design nears completion and the blocks of code must be integrated. This produces the flat "tail" of the curve.



Input	Output
0.000	0.200
0.100	0.200
0.200	0.220
0.300	0.390
0.400	0.690
0.500	0.850
0.600	0.930
0.700	0.970
0.800	0.990
0.900	1.000
1.000	1.000

Data Points: 11

Edit Output:

Figure 3-6: A Nonlinear Internal Precedence Relationship

A development process's Internal Precedence Relationship describes the available work constraints which the tasks aggregated into a single phase impose on each other. It estimates the impacts of the dependency network which exists among the phase's tasks. For example if the erection of structural steel for a high-rise building was the phase the Internal Precedence Relationship would reflect that columns must be installed before the beams which they support. These constraints can act as a bottleneck in the availability of work. Most previously published system dynamics models of projects have assumed that all uncompleted tasks are available for completion (e.g. Abdel-Hamid, 1984; Richardson and Pugh, 1981; Roberts, 1974). This assumption implies that all tasks are independent and could be performed in parallel and that the nature of the development process imposes no constraints on the number of tasks available for completion. However product development research (e.g. Rosenthal, 1992; Clark and Fujimoto, 1991) and the steel erection example above show that processes can and frequently do internally constrain the availability of work.

The stock and flow structure of the development process within a single phase follows from the feedback structure shown in Figure 3-4. Development tasks flow into and through three stocks: the Completed not Checked, Known Rework, and Checked & Released stocks (Figure 3-7).

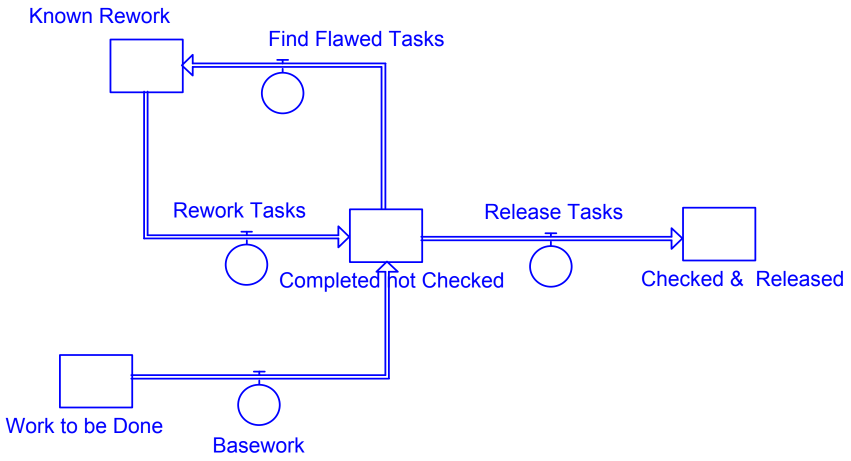


Figure 3-7: Process Structure Stocks and Flows (Single Phase)

Tasks are completed for the first time through the performance of basework. They accumulate in the Completed not Checked stock. If no tasks are flawed or those flaws are not found tasks leave the Completed not Checked stock and pass through the Release Tasks flow into the Checked & Released stock. This represents delivering tasks to the managers of downstream phases or to customers. Flawed tasks are modeled in the Errors sectors. Tasks which are found to be flawed flow to the Known Rework stock. The Rework flow returns corrected tasks to the Completed, not Checked stock for inspection. The following stock equations describe these accumulations explicitly.

Tasks_Completed(Phase)=Tasks_Completed(Phase)+dt*(Basework(Phase)+
Rework(Phase)-Release_Tasks(Phase)-RW_due_to_InPhase_QA(Phase))

Known_Rework(Phase)=Known_Rework(Phase)+dt*
(RW_due_to_InPhase_QA(Phase)-Rework(Phase))

Tasks_Released(Phase)=Tasks_Released(Phase)+dt*(Release_Tasks(Phase)

Figure 3-8 shows the process structure for a single phase with auxiliary parameters.

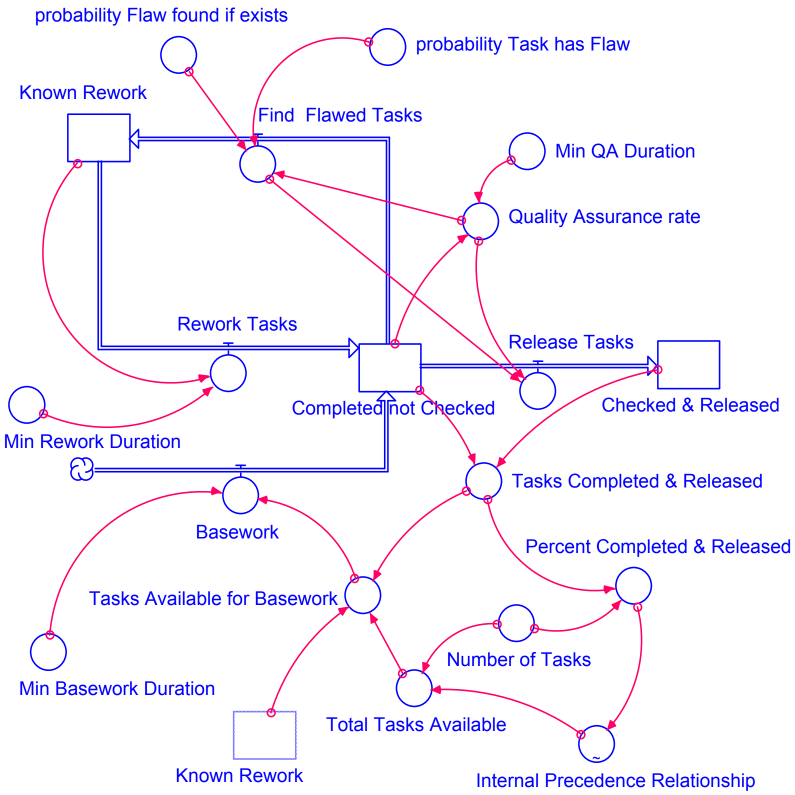


Figure 3-8: Process Structure (Single Phase)

The form taken by the limits on progress imposed by the development process structure for each of the four development activities are similar. A phase's demand for each activity (Basework, Quality Assurance, Rework and Coordination) is the number of tasks available for the activity. The general process limit equation is:

$$\text{Development Activity Process Limit} = \frac{\text{Tasks Available for the Activity}}{\text{Minimum Activity Duration}}$$

Minimum Activity Durations describe the relative difficulty of the four development activities. The structures which describe the development activities will be described in increasing complexity from Rework to Coordination to Quality Assurance to Basework.

The Rework flow structure is the simplest. The value of the Tasks Available for the Activity parameter in the equation above for the Rework flow is the number of tasks in the Known Rework stock. The formulation assumes that all tasks in the rework stock are independent. The Rework process limit equation is:

$$\text{Rework Process Limit} = \frac{\text{Known Rework}}{\text{Minimum Rework Duration}}$$

Demand for the Coordination activity is the Coordination Backlog. Since coordination is the interaction of developers across phases it is required only for multiple phases. This stock is the accumulation of the sum of the tasks which have been corrupted due to inheriting flawed tasks from upstream and the flawed tasks which have been released and returned by downstream development phases less the tasks which have been coordinated. Figure 3-15 shows this structure. The Coordination process limit equation is:

$$\text{Coordination Process Limit} = \frac{\text{Coordination Backlog}}{\text{Minimum Coordination Duration}}$$

The value of the Tasks Available for the Activity parameter for the Quality Assurance rate is the number of tasks which have been completed but not yet checked. This is the Completed not Checked stock. Therefore the Quality Assurance process limit equation is:

$$\text{Quality Assurance Process Limit} = \frac{\text{Completed not Checked}}{\text{Minimum QA Duration}}$$

Quality Assurance is the basis for two flows. The first flow is the Find Flawed Tasks flow which depends on the Quality Assurance rate and the effectiveness of those efforts at finding flawed tasks. Quality assurance effectiveness is measured with the probability of finding a flawed task. This is the product of the probability of finding a flaw if it exists and the probability of a task being flawed. Therefore the Find Flawed Tasks equation is:

$$\text{Find Flawed Tasks} = \frac{\text{Completed not Checked}}{\text{Minimum QA Duration}} * \text{p(Flaw found if exists)} * \text{p(Task is Flawed)}$$

The probability of a flawed task being found if it exists is based upon the adequacy of the quality assurance effort, as measured by the ratio of the quality assurance labor applied to the quality assurance labor required. This is described in the resources sector of the model. The probability that a task is flawed is the ratio of the number of unchecked flawed tasks (in the Internal Errors sector) to the total number of unchecked tasks.

The second flow driven by Quality Assurance is the Release Tasks flow. All tasks which are checked leave the Completed not Checked stock. Those that are found to be flawed are sent to the Known Rework stock as described above. Tasks which are found to be unflawed (those without flaws and those with flaws that were missed) are released. Therefore the equation for the process limit on the Release Tasks flow is the total number of tasks checked less those found to have flaws:

Release Tasks = Quality Assurance rate - Find Flawed Tasks

The structure of the Basework flow is the most complex of the four development activities. The demand for Basework is the total number of tasks which *can* be completed less the tasks which *have already* been completed at least once. Therefore the Basework process limit equation is:

$$\text{Basework Process Limit} = \frac{(\text{Total Tasks Available - Completed not Checked - Checked \& Released - Known Rework})}{\text{Minimum Basework Duration}}$$

The Total Tasks Available is the number of tasks which could be completed based upon the tasks which have been completed and released. This constraint is described with the phase's Internal Precedence Relationship and answers the question "What percent of the tasks are available for initial completion based upon the percent which have been completed and released?" Known Rework is not included in the Total Tasks Available because flawed tasks do not make additional work available. The equation for the Total Tasks Available is:

Total Tasks Available = Number of Tasks in Project Scope *
Min(Internal Precedence Relationship , External Precedence
Relationship)

The Internal and External Precedence Relationships are important characterizations of the nature of specific development process. They can be nonlinear in nature and is therefore described with graphic functions.

The Development Task sector includes two additional flows and one additional stock for modeling inter-phase linkages (Figure 3-15). The first of these flows to be described models the corruption of completed work in a focal phase due to inheriting flawed tasks from upstream.

```
RW_due_to_Corrupted_tasks(Phase)=(Net_Corrupted_and_Found_Tasks(Phase)-
(QA_inspection_rate(Phase)*((Net_Corrupted_and_Found_Tasks(Phase)/
(QA_inspection_rate(Phase)+1e-9))*prob_Task_Flawed_and_Found(Phase))))
```

Rework due to Corrupted Tasks moves tasks which are completed but not checked into the Known Rework stock based on the fraction of tasks found to be corrupted by inherited errors (described later). Relative sizes of the phases are used to scale upstream errors found into tasks corrupted in the focal phase.

The second flow required for multiple phases is Rework due to Errors Discovered by Downstream Phases.

RW_due_to__Dwnstrm_QA(Phase)=Total_Err_disc_by_Dn(Phase)+
Total_Corrupted_by_Upstream_Retraction(Phase)

Errors which are discovered downstream are aggregated with released work corrupted by upstream errors (described later), removed from the Tasks Released stock and added to the Known Rework stock. The revised Development Task sector stock equations which include these inter-phase flows are:

Tasks_Completed(Phase)=Tasks_Completed(Phase)+dt*(Basework(Phase)+
Rework(Phase)-Release_Tasks(Phase)-RW_due_to_InPhase_QA(Phase)-
RW_due_to_Corrupted_tasks(Phase))

Known_Rework(Phase)=Known_Rework(Phase)+dt*(RW_due_to_InPhase_QA(Phase)+
RW_due_to_Corrupted_tasks(Phase)+RW_due_to__Dwnstrm_QA(Phase)-Rework(Phase))

Tasks_Released(Phase)=Tasks_Released(Phase)+dt*(Release_Tasks(Phase)-
RW_due_to__Dwnstrm_QA(Phase))

The two inter-phase error flows control the model's fourth development activity, coordination, and determine the effort required to address inherited errors and released and discovered errors. Each of the two flows described above model development activities which require interaction between development phases. They generate a need for coordination.

Current_Coord_added(Phase)=RW_due_to__Dwnstrm_QA(Phase)+
RW_due_to_Corrupted_tasks(Phase)

The accumulation of these flows represents a backlog of coordination work needing to be completed. The performance of the coordination activity reduces the coordination backlog. The following equation describes the accumulation of coordination work.

```
Coord_Backlog(Phase)=Coord_Backlog(Phase)+dt*
(Current_Coord_added(Phase)-Coordination(Phase))
```

The Development Tasks sector also uses a Fraction Available due to External Gates parameter to model available-work constraints between phases (inter-phase constraints) . Tasks Available for Basework is based on the minimum of the internal and external gates. The External Precedence Relationships describe the available-work constraints between development phases in a manner analogous to the internal available-work constraint described by the Internal Precedence Relationships. An External Precedence Relationship describes the availability of work in a downstream phase based on the amount of work which has been released by an upstream phase. The input (abscissa) of an External Precedence Relationship is the Percent of Upstream Tasks Released. The output is the Percent of Downstream Tasks Available for Basework.

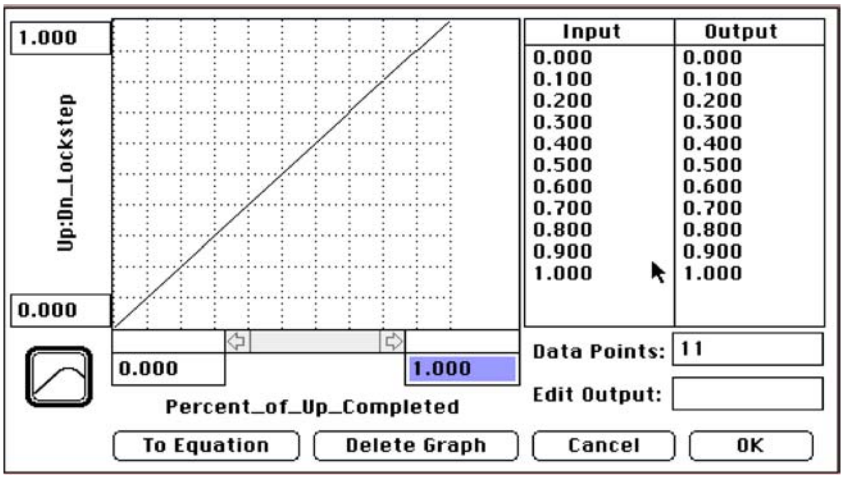
Like a development phase's Internal Precedence Relationship, an External Precedence Relationship between two development phases can act as a bottleneck in the availability of work. Most previously published system dynamics models of projects have assumed that all uncompleted tasks are available for completion (e.g. Abdel-Hamid and Madnick, 1991; Richardson and Pugh, 1981; Roberts, 1974). This assumption implies that the nature of the development process imposes no constraints on the number of tasks available for completion. However the success of the Critical Path and PERT methods in staticly modeling inter-phase dependencies in development projects and product development research (e.g. Rosenthal, 1992; Clark and Fujimoto, 1991; Eppinger et al., 1990) show that relationships among development phases can and often do constrain the availability of work.

The purpose of External Precedence Relationships is the same as the precedence relationships used in the Critical Path and PERT methods: to describe the dependencies of development phases on each other for the initial completion of work. However there are several important differences between External Precedence Relationships and precedences used in the Critical Path and PERT methods.

- External Precedence Relationships describe the dependency between two phases along the entire duration of the phases instead of only at the start and finish of the phases as in the Critical Path and PERT methods.
- External Precedence Relationships can be nonlinear.

- External Precedence Relationships describe a dynamic relationship between development phases by allowing the output (Percent Tasks Available for Basework) to fluctuate over the life of the project depending on the current conditions of the project, as described by the External Precedence Relationship's input (Percent Upstream Tasks Released).

External Precedence Relationships can be used to describe rich inter-phase relationships which cannot be described with Critical Path and PERT precedences. For example a downstream phase which is constrained by the release of upstream tasks throughout its duration (not only at the beginning or end of the phase) in a linear relationship can be described with a "lockstep" External Precedence Relationship, as shown in Figure 3-13.

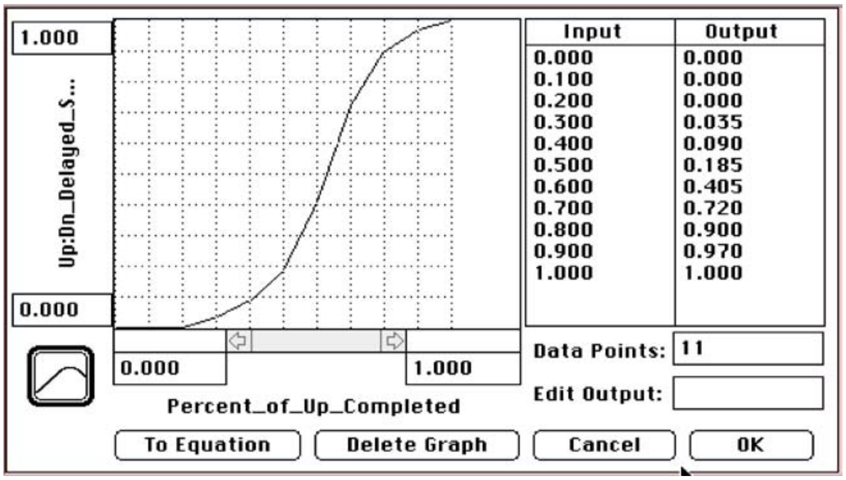


Input	Output
0.000	0.000
0.100	0.100
0.200	0.200
0.300	0.300
0.400	0.400
0.500	0.500
0.600	0.600
0.700	0.700
0.800	0.800
0.900	0.900
1.000	1.000

Data Points: 11

Figure 3-9: A "Lockstep" Constraint described with An External Precedence Relationship

The inter-phase relationship in Figure 3-9 is linear. One of the advantages of the model structure used here is the ability to describe nonlinear inter-phase constraints. Varying levels of concurrence in the development process can be described with External Precedence Relationships by altering the shape of the curve in the graphical function. Infinite order delays between phases can also be described by shifting the point along the abscissa at which the first downstream tasks become available. As an example the External Precedence Relationship shown in Figure 3-10 describes an inter-phase relationship in which the downstream phase must wait until the upstream phase has released 20% of its tasks and then can perform Basework relatively concurrently until near the completion of the downstream phase.



Input	Output
0.000	0.000
0.100	0.000
0.200	0.000
0.300	0.035
0.400	0.090
0.500	0.185
0.600	0.405
0.700	0.720
0.800	0.900
0.900	0.970
1.000	1.000

Data Points: 11

Edit Output:

Figure 3-10: A Delayed Concurrent Constraint described with An External Precedence Relationship

External Precedence Relationships are used in the Product Development Project Model to describe some of the complexity of the inner structure of a product development project and its impacts on project performance.

External Precedence Relationships reflect the concurrence of phases which are dependent.

Concurrence(up,down)=FIFZE(1.00,TABHL(T6(*,up,down), Fraction_Released(up),0,1,0.10),Dependency(up,down))

Fraction_Avail_due_to__Ext_gates(Phase)=MIN(FIFZE(1.00,Concurrence(1,Phase),Dependency(1,Phase)),FIFZE(1.00,Concurrence(2,Phase),Dependency(2,Phase)),FIFZE(1.00,Concurrence(3,Phase), Dependency(3,Phase)),FIFZE(1.00,Concurrence(4,Phase),Dependency(4,Phase)),FIFZE(1.00,Concurrence(5,Phase),Dependency(5,Phase)))

Fraction_Released(Phase)=Tasks_Released(Phase)/Task_List(Phase)

The Concurrence variable reflects the External Precedence Relationship as discussed above, using the Fraction Released by the upstream phase to constrain the work available in the downstream phase. The Fraction Available due to External Gates variable includes only the dependent upstream phases in determining the available work with a set of switches linked to the project network. The switches and network are linked with the Dependency variable, which is 1 if the phases are dependent and 0 if they are not.

The complete Development Tasks sector is shown in Figure 3-11.

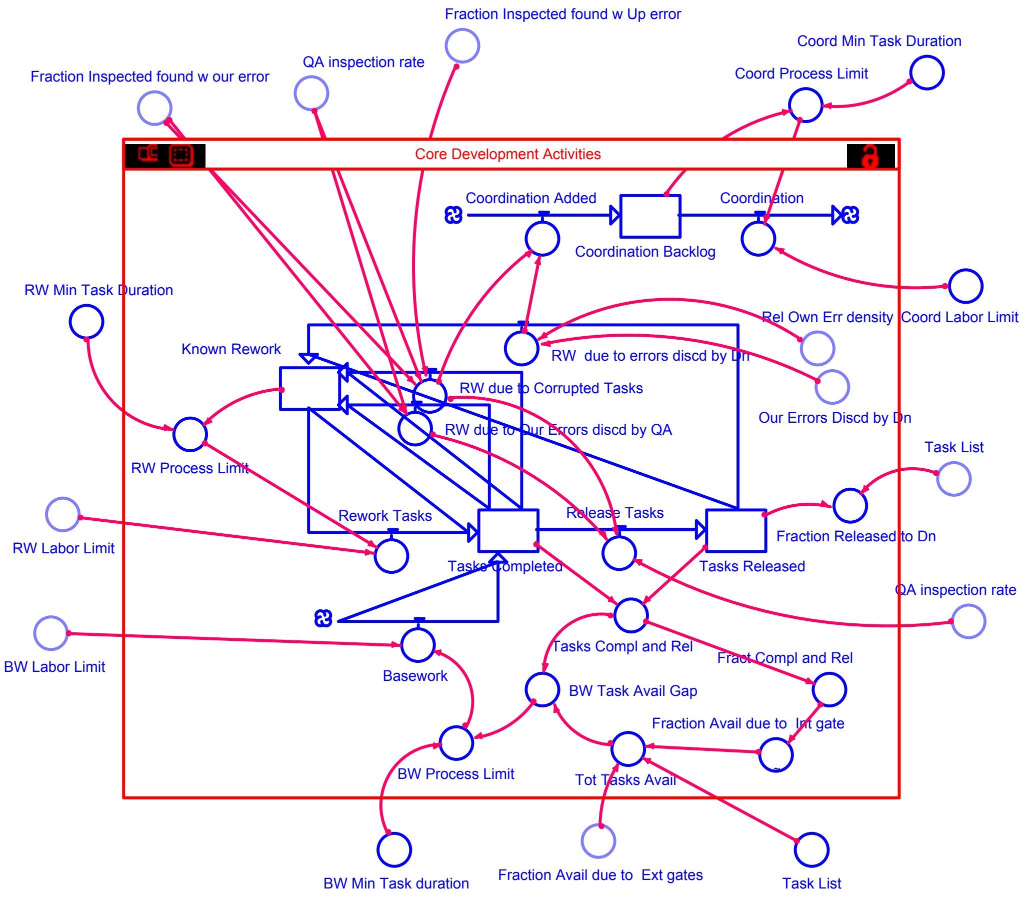


Figure 3-11: The Development Tasks Sector

3.3.3.3 The Errors Sectors

Three sectors model errors: the internal, inherited, and released errors sectors. Errors generated, discovered, and corrected within a single phase are modeled by the Internal Errors sector. A co-flow structure (Homer, 1983 Appendix Q) is used in which the stocks and flows are directly related to the stocks and flows in the Development Tasks sector. A comparison of Figure 3-11 and Figure 3-12 below shows their similarity. Flawed tasks are discovered through the Quality Assurance activity. As described previously tasks found to be flawed move through the Find Flawed Tasks flow from the Completed not Checked stock to a stock of Known Rework. These tasks are corrected through the Rework Tasks activity and returned to the Completed not Checked stock. The Internal Errors sector models the generation of flaws, which can be generated during both Basework and Rework. This means that a task being reworked to correct an existing defect may become flawed during the rework process. Because quality assurance efforts are not perfect some flaws are missed (i.e. Type 2 errors are allowed). Therefore some flawed tasks are mistakenly considered to be unflawed and are released with the unflawed tasks. These errors are inherited by the phase's dependent downstream phases. The model assumes that unflawed tasks mistakenly considered flawed are incorporated into the values of the Minimum Activity Durations.

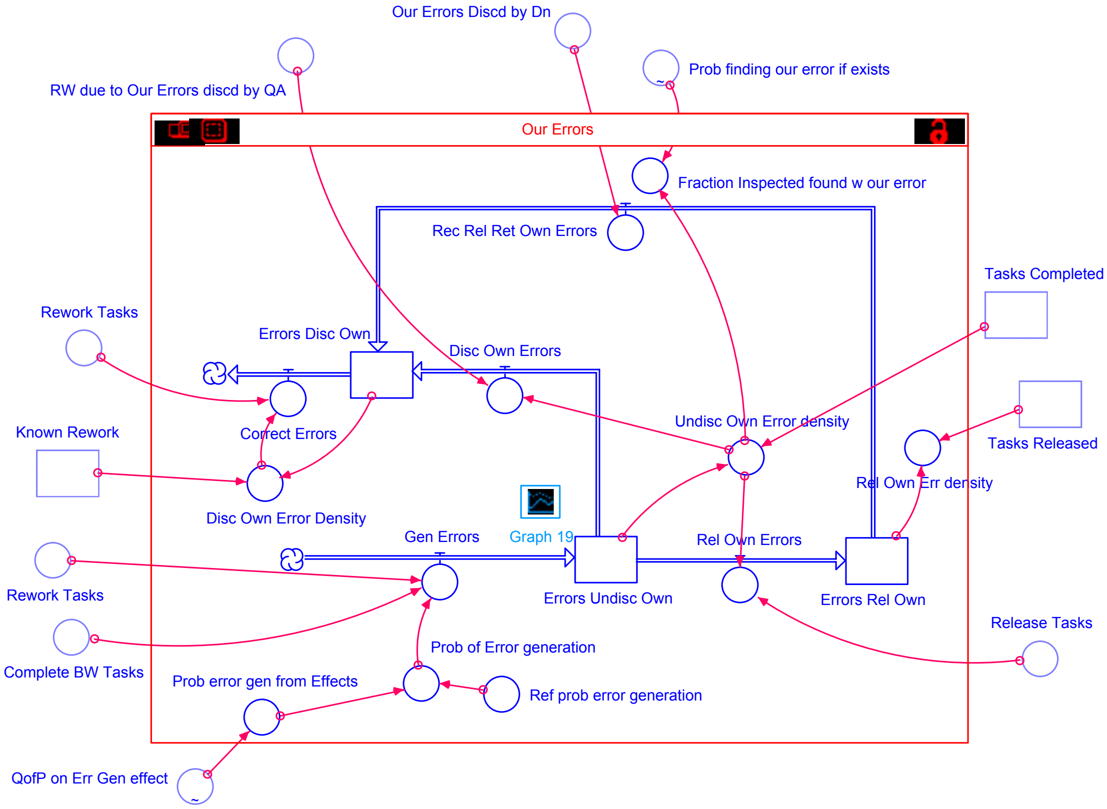


Figure 3-12: The Internal Errors Sector

The key equations used to model the internal errors sector are described below. Several of the tables used in the Product Development Project Model to describe nonlinear relationships between parameters have an "S" shape. These curves have upper and lower limits to output values, small unit changes in output near those limits and larger unit changes in output near the center of the input range. The reasoning behind this curve follows. One limit is at or near normal operating condition (e.g. labor provided approximates labor needed). Changes in input values near this limit generate no or small changes because developers do not perceive enough digression from normal conditions to trigger a significant response. Developer responses and output values change more as input conditions move further from normal and developers notice and react to the variance. Output unit changes decrease again as input values approach the other limit based on the assumption of smooth continuous developer response to limits.

```
Generate_Errors(Phase)=(Basework(Phase)+Rework(Phase))*prob_Task_flawed(Phase)
```

```
prob_Task_flawed(Phase)=1-((1-Basic_prob_flawed_Task(Phase))*(1-
prob_of_err_gen_by_QofP(Phase)))
```

Errors are generated during the basework and rework development activities. The probability of error generation is based on two factors which combine to cause errors. The inherent complexity of the task is reflected in the basic probability that a task is flawed. The impacts of the development work are reflected in the probability of an error being generated by the quality of practice. Each of these probabilities are used to find the probability of no error being generated by the task complexity or quality of practice. These "clean" probabilities are combined to find the probability of a task being flawed by neither of these factors. The resulting probability of no error is used to find the probability of an error by subtracting from 1.

```
prob_of_err_gen_by_QofP(Phase)=TABHL(T3,Quality_of_Practice
(Phase)/Ref_Qual_of_Practice(Phase),0,1,0.10)
T T3=0.55/0.45/0.36/0.28/0.21/0.15/0.10/0.06/0.03/0.01/0.00
```

The quality of practice influences the probability of error generation through a reverse "S" shaped curve which does not increase errors if the quality of practice is above a reference level. Excess quality of practice is assumed to not hurt a project. The curve rises to a maximum of 55% when the quality of practice is zero. This assumes that there is a limit to the harmful impacts of poor quality of practice on the generation of errors. This is based on the assumption that there is some minimum underlying ability in the developers to perform development tasks which cannot be totally eroded by the conditions of the project. This is a reasonable assumption when developers are professionally trained and the process interacts using Mintzberg's (1979) standardization of skills .

Several of the stocks and flows in the internal errors sector are directly analogous to stocks and flows in the development tasks sector. The error parameters differ from the task parameters due to the densities of flaws. The following equations are used to model those densities.

```
Compl_Task_error_density(Phase)=Our_Undiscd_Errors(Phase)/(Tasks_Completed(Phase))
```

```
Our_Discd_Error_density(Phase)=Our_Discd_Errors(Phase)/(Known_Rework(Phase)+1e-9)
```

```
Rel_Task__error_density(Phase)=Our_Errors_Released(Phase)/
(Tasks_Released(Phase)+1e-9)

Clean_Task_error_density(Phase)=(Compl_Task_error_density(Phase)*
(1-Prob_finding_our_error_if_exists(Phase)))/((1-Compl_Task_error_density(Phase))+
(Compl_Task_error_density(Phase)*(1-Prob_finding_our_error_if_exists(Phase)))+1e-9)
```

This last density is the number of unflawed tasks divided by the sum of the number unflawed and flawed but missed tasks. The numerator is the product of the probability of a task being flawed and the probability of finding a flaw if it exists. The denominator is the numerator plus the compliment of the probability of a task being flawed.

The error coflow flow equations are formed by combining the densities and the task flows. The error coflow stock equations are the accumulations of the flows.

```
Our_Undiscd_Errors(Phase)=Our_Undiscd_Errors(Phase)+dt*(Generate_Errors(Phase)-
Release_Errors(Phase)-Disc_Our_Errors(Phase)-Errors_lost_in_Corrupted_Tasks(Phase))

Disc_Our_Errors(Phase)=RW_due_to_InPhase_QA(Phase)

Errors_lost_in_Corrupted_Tasks(Phase)=Compl_Task_error_density(Phase)*
RW_due_to_Corrupted_tasks(Phase)

Release_Errors(Phase)=(Release_Tasks(Phase)*Clean_Task_error_density(Phase))
Our_Errors_Released(Phase)=Our_Errors_Released(Phase)+dt*
(Release_Errors(Phase)-Receive_Our_Errors_fr_Dn(Phase))

Receive_Our_Errors_fr_Dn(Phase)=Total_Err_disc_by_Dn(Phase)+
Total_Corrupted_by_Upstream_Retraction(Phase)

Our_Discd_Errors(Phase)=Our_Discd_Errors(Phase)+dt*(Receive_Our_Errors_fr_Dn(
Phase)+Disc_Our_Errors(Phase)-Correct_Our_Errors(Phase))

Correct_Our_Errors(Phase)=Rework(Phase)*Our_Discd_Error_density(Phase)

Total_Err_disc_by_Dn(Phase)=Up_Flawed_Tasks_found(Phase,1)+
Up_Flawed_Tasks_found(Phase,2)+Up_Flawed_Tasks_found(Phase,3)+
Up_Flawed_Tasks_found(Phase,4)+Up_Flawed_Tasks_found(Phase,5)
```

The total number of errors returned to a phase is the sum of the errors released by that phase and discovered by all the downstream phases.

```
Prob_finding_our_error_if_exists(Phase)=TABHL(T2,QA_Status(Phase),0,20,2.0))
T T2=0.00/0.335/0.535/0.685/0.81/0.88/0.925/0.96/0.985/1.00/1.00
```

The probability of finding an existing error is based on the adequacy of the quality of practice. No errors can be found if the quality of practice is zero. This assumes that the project conditions can degrade the quality of the work done by the developers to such a degree that they miss all the errors in the work they inspect. This is a reasonable lower bound. The probability of finding errors based on the adequacy of the actual quality of practice increases as the actual quality of practice rises above a reference value. An upper bound of finding all errors (assuming other factors do not prevent discovery) is approached as the quality of practice reaches 18 times the reference value.

```
prob_Task_flawed_and_Found(Phase)=Prob_finding_our_error_if_exists(Phase)*
Compl_Task_error_density(Phase)
```

The probability that a task is both flawed and discovered to be flawed is the product of the probabilities that it is flawed and the probabilities that the flaw is found.

Errors received by a phase from an upstream phase are modeled with the Upstream Errors sector. Figure 3-13 shows an example of this structure for a focal phase with two upstream phases. The Product Development Project Model can model the inheritance of errors from a flexible number of phases. These errors corrupt tasks done in the focal phase. Each phase discovers a portion of its tasks which have been corrupted by the errors it inherits based on the quantity and effectiveness of its quality assurance efforts. Multiple corruptions of the same task are eliminated and the net corrupted tasks are used in the Development Tasks and Internal Errors sectors.

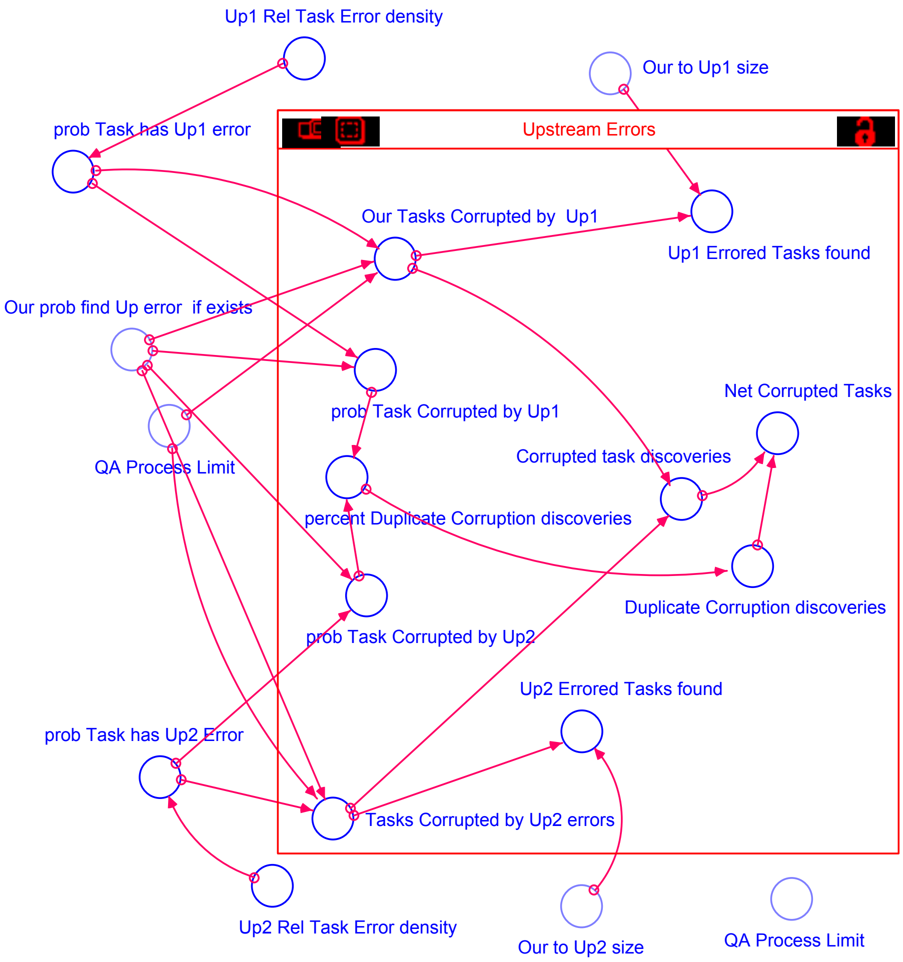

Up1 Rel Task Error density
Our to Up1 size
Upstream Errors
prob Task has Up1 error
Our Tasks Corrupted by Up1
Up1 Errored Tasks found
Our prob find Up error if exists
prob Task Corrupted by Up1
Net Corrupted Tasks
QA Process Limit
Corrupted task discoveries
percent Duplicate Corruption discoveries
Duplicate Corruption discoveries
prob Task Corrupted by Up2
Up2 Errored Tasks found
prob Task has Up2 Error
Tasks Corrupted by Up2 errors
Up2 Rel Task Error density
Our to Up2 size
QA Process Limit

Figure 3-13: The Upstream Errors Sector

The key equations used to model the upstream errors sector are described below.

prob_Task_Corrupted_and_found(up,Phase)=prob_find_Up_error__if_exists(Phase)*
Rel_Task__error_density(up)*Dependency(up,Phase)

The probability that an inherited task is both flawed and discovered to be flawed is the product of the probabilities that it is flawed and the probabilities that the flaw is found.

Tasks_Corrupted_and_Found(up,Phase)=QA_inspection_rate(Phase)*
prob_Task_Corrupted_and_found(up,Phase)

The number of tasks found to be corrupted is the probability that finding corrupted tasks times the rate at which tasks are being inspected in the downstream phase.

Corrupted_task_discoveries(Phase)=Tasks_Corrupted_and_Found(1,Phase)+
Tasks_Corrupted_and_Found(2,Phase)+Tasks_Corrupted_and_Found(3,Phase)+
Tasks_Corrupted_and_Found(4,Phase)+Tasks_Corrupted_and_Found(5,Phase)

The total number of tasks discovered to be corrupted is the sum of the corrupted tasks found from all upstream dependent phases. Phases not linked to the focal phase have been assigned zero tasks corrupted. This value can include multiple discoveries of the same flawed task.

```
percent_Multiple_Corruption_discoveries(Phase)=
(prob_Task_Corrupted_and_found(1,Phase)*prob_Task_Corrupted_and_found(2,Phase))+
(prob_Task_Corrupted_and_found(1,Phase)*prob_Task_Corrupted_and_found(3,Phase))+
(prob_Task_Corrupted_and_found(1,Phase)*prob_Task_Corrupted_and_found(4,Phase))+
(prob_Task_Corrupted_and_found(1,Phase)*prob_Task_Corrupted_and_found(5,Phase))+
(prob_Task_Corrupted_and_found(2,Phase)*prob_Task_Corrupted_and_found(3,Phase))+
(prob_Task_Corrupted_and_found(2,Phase)*prob_Task_Corrupted_and_found(4,Phase))+
(prob_Task_Corrupted_and_found(2,Phase)*prob_Task_Corrupted_and_found(5,Phase))+
(prob_Task_Corrupted_and_found(3,Phase)*prob_Task_Corrupted_and_found(4,Phase))+
(prob_Task_Corrupted_and_found(3,Phase)*prob_Task_Corrupted_and_found(5,Phase))+
(prob_Task_Corrupted_and_found(4,Phase)*prob_Task_Corrupted_and_found(5,Phase))
```

The total probability of a flaw discovery being repeated is the sum of the probabilities of a multiple discovery by each of the possible phase interactions. The probability of a repeated discovery is the product of each of the individual discovery probabilities.

```
Multiple_Corruption_discoveries(Phase)=QA_inspection_rate(Phase)*
percent_Multiple_Corruption_discoveries(Phase)
```

Multiple discoveries occur at the rate of inspection times the percent of tasks found more than once to be flawed.

```
Net_Corrupted_and_Found_Tasks(Phase)=Corrupted_task_discoveries(Phase)-
Multiple_Corruption_discoveries(Phase)
```

The net number of tasks found to be corrupted is the actual number of corrupted and flawed tasks corrected for multiple error discoveries. This is the total number of discoveries less the number of multiple discoveries.

```
Up_Flawed_Tasks_found(up,Phase)=Tasks_Corrupted_and_Found(up,Phase)*
TaskListScale(up,Phase)

Total_Err_disc_by_Dn(Phase)=Up_Flawed_Tasks_found(Phase,1)+
Up_Flawed_Tasks_found(Phase,2)+Up_Flawed_Tasks_found(Phase,3)+
Up_Flawed_Tasks_found(Phase,4)+Up_Flawed_Tasks_found(Phase,5)
```

The flawed tasks returned to the upstream phase are those discovered by the downstream phases increased or decreased by the relative sizes of the upstream phase and each of the dependent downstream phases. The total returned flawed tasks is the sum of those returned from all downstream phases.

```
prob_find_Up_error__if_exists(Phase)=MIN(1,Coord_effect_on_Find_Up_Errors(Phase)*
QofP_Error_Disc_effect(Phase))
```

```
Coord_effect_on_Find_Up_Errors(Phase)=TABHL(T1,Coord_Status(Phase),0,1,0.10)
T T1=0.00/0.015/0.045/0.115/0.265/0.54/0.715/0.84/0.92/0.975/1.00
```

```
QofP_Error_Disc_effect(Phase)=TABHL(T4,Quality_of_Practice(Phase)/
Ref_Qual_of_Practice(Phase),0,1,0.10)
T T4=0.20/0.20/0.22/0.25/0.3/.4/.65/.7/.85/.95/1.0
```

The probability of finding an upstream error if it exists is impacted by the adequacy of coordination and the level of the quality of practice. Both relationships have an "S" shape. The lower limit of the coordination relationship is zero, indicating that if there is no coordination there is no chance of finding upstream errors. The quality of practice relationship has a lower limit of 0.20, indicating that some errors could be found if two teams were coordinating but one had a low quality of practice. Both upper limits are 1.0 since finding more than 100% of the existing errors is unreasonable.

Two types of project errors cause tasks to flow from the Tasks Released stock to the Known Rework stock. The first is errors which are released to downstream phases, discovered there, and returned for correction. The second type of project error is upstream errors which have been discovered and require the rework of completed and released tasks in downstream dependent phases. This represents the notification of other phases of an error by developers. This second type of errors reflects the following conditions: the middle phase in a sequential set of three phases completes its tasks before the final phase discovers errors created and released by the first phase and not discovered by the middle phase. The final phase returns those errors to the first phase for correction, generating rework for the first and final phases. Since the middle phase is completed these newly discovered errors will not be reflected in the corruption of tasks in the middle phase. In real projects these errors by the first phase will require the rework of released tasks in the middle phase as well. This relationship models the notification of downstream phases when an upstream phase is notified that it has made an error. The model uses the following equation to model this flow.

```
RW_due_to__Dwnstrm_QA(Phase)=Total_Err_disc_by_Dn(Phase)+
Total_Corrupted_by_Upstream_Retraction(Phase)
```

The flow is the sum of the flaws due to released errors discovered downstream and the flaws caused by the retraction of finished work upstream due to error discoveries. The two contributors to the sum are the totals of the errors for each of the upstream or downstream phases linked to the focal phase.

3.3.4 The Resources Subsystem

The structural components of the Resources subsystem are based on previously constructed and tested system dynamics models which are documented in the literature. Table 3-1 at the end of the model structure description lists the primary model references for each model sector. The Resources subsystem consists of the Gross Labor, Labor Allocation, Workweek, Experience, Quality of Practice, Development Limit and Expected Productivity sectors.

3.3.4.1 The Labor Sectors

The Gross Labor sector (Figure 3-14) models the quantity of labor used in a development phase. Headcount is assumed to be the number of full time, rested, experienced product developers. Therefore a rested experienced developer which spent half of his or her time developing the product would be considered 0.50 person. Headcount is changed within the range set by the minimum and maximum headcount to bring the actual headcount to