Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

2004

Use of system dynamics and Easel for simulation of the software development process

Alicia Strupp

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Strupp, Alicia, "Use of system dynamics and Easel for simulation of the software development process" (2004). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

USE OF SYSTEM DYNAMICS AND EASEL FOR SIMULATION OF THE SOFTWARE DEVELOPMENT PROCESS

By

Alicia Strupp

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Information Technology

Rochester Institute of Technology

B. Thomas Golisano College
of
Computing and Information Sciences

August 16, 2004

Rochester Institute of Technology

B. Thomas Golisano College of Computing and Information Sciences

Master of Science in Information Technology

Thesis Approval Form

Student Name: Alicia Strupp

Thesis Title: Using Systems Dynamics and Easel to Study the Software Development Process

Thesis Committee

Professor Jeffrey A. Lasky — Jeffrey A. Lasky — 8/16/04
Chair

Professor Deborah G. Coleman — Deborah G. Coleman — 8/16/04
Committee Member

Professor Timothy D. Wells — Timothy D. Wells — 8/16/04
Committee Member

Thesis Reproduction Permission Form

Rochester Institute of Technology

B. Thomas Golisano College
of
Computing and Information Sciences

Master of Science in Information Technology

USE OF SYSTEM DYNAMICS AND EASEL FOR

SIMULATION OF THE

SOFTWARE DEVELOPMENT PROCESS

I, Alicia Strupp, hereby grant permission to the Wallace Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part. Any reproduction must not be for commercial use or profit.

Date: 9/15/04 Signature of Author: Alicia Strupp

ABSTRACT

Team software development is a complex and mostly unpredictable process and is characterized by inefficient use of staff and calendar resources. Given the magnitude of software development costs, a deeper understanding of the process may suggest ways to improve resource utilization.

Simulation modeling is a useful approach to study the dynamics of complex systems. System dynamics characterizes systems as collections of interacting, non-linear feedback loops. The foundations of system dynamics were developed at MIT in the early 1950s. Since that time, system dynamics has been applied to a large number of complex system domains. In the early 1980s, the system dynamics simulation method was first used at MIT to develop a software development process model.

A different approach to modeling complex systems is to use an actor, or property-based programming language. In a property-based model, the behaviors of individual entities are represented as concurrently executing threads, and discrete event clocks are used to simulate time. Easel is a new property-based programming language developed at the Software Engineering Institute housed at Carnegie Mellon University. Although determining the survivability of large-scale networks was the motivation to develop Easel, the SEI has conducted some initial work in applying Easel to the software development process domain.

This thesis compared the use of system dynamics and Easel as tools to study the software development process. Both modeling approaches were used to test the validity of Brooks's Law under different hiring strategies for small, medium, and large-scale projects. The models produced nearly identical results, and so provided a high level of confidence that the models were logically equivalent.

The thesis concludes with a comparison of the two techniques based on background knowledge required, object representation, debugging difficulty, model maintainability, scalability, and timing control. A summary about the applicability of each technique is presented and recommendations for future work are offered.

TABLE OF CONTENTS

ABSTRACT 2

LIST OF TABLES 6

LIST OF FIGURES 7

CHAPTER 1 9

INTRODUCTION 9

1.1 The Nature of the Software Development Process *9*
1.1.1 Study of Problems 10
1.1.1.1 Cost 10
1.1.1.2 Schedule 11
1.1.1.3 Quality 11
1.1.2 Process Improvement 11
1.1.3 Software Development System Complexity 13
1.1.4 The Proposed Solution 15

1.2 Opportunities and Challenges *16*
1.2.1 Opportunities 16
1.2.2 Challenges 19

1.3 Simulation of Software Development: Two Approaches *20*
1.3.1 System Dynamics 20
1.3.1.1 Background 20
1.3.1.2 CLDs 22
1.3.1.3 Flow Diagrams 23
1.3.1.4 Advantages 25
1.3.1.5 Disadvantages 25
1.3.1.6 Mapping the software development lifecycle model using system dynamics 26
1.3.1.7 Prior applications of system dynamics to software development 29
1.3.2 Easel 30
1.3.2.1 Background 30
1.3.2.2 Advantages 32
1.3.2.3 Disadvantages 32
1.3.2.4 Mapping the Software Development Life Cycle Model Using Easel 33
1.3.2.5 Prior Applications of Easel to Software Development 36

CHAPTER 2 37

LITERATURE REVIEW 37

2.1 System Dynamics Literature Review *37*
2.1.1 Beginning of System Dynamics 37
2.1.2 Applications of System Dynamics 39
2.1.2.1 Corporate management modeling 39
2.1.2.2 Biological and medical modeling 40
2.1.2.3 Energy and environmental modeling 41
2.1.2.4 Dynamic decision making modeling 41
2.1.2.5 Supply chain management modeling 42
2.1.2.6 Software engineering modeling 42
2.1.3 Organizational Studies 43
2.1.4 Tutorials 48
2.1.5 Advantages 50

2.1.6 Disadvantages ... 50

2.2 Easel Literature Review ... *52*
2.2.1 Actor-based Languages ... 53
2.2.1.1 Characteristics. ... 54
2.2.2 Beginning of Easel ... 55
2.2.3 Easel and Software Development ... 56
2.2.4 Advantages ... 57
2.2.5 Disadvantages ... 58

2.3 Brooks's Law Literature Review ... *58*

CHAPTER 3 ... **63**

METHODOLOGY ... **63**

3.1 Introduction ... *63*
3.1.1 Brooks's Law ... 63

3.2 Model Descriptions ... *64*
3.2.1 The System Dynamics Model ... 64
3.2.2 The Easel Model ... 68

3.3 Simulation Design ... *69*
3.3.1 System Dynamics Simulation ... 70
3.2.2 Easel Simulation ... 72

3.4 Experimental Design ... *76*

CHAPTER 4 ... **79**

DATA ANAYLSIS ... **79**

4.1 Definitions ... *79*

4.2 Small Project Analysis ... *80*
4.2.1 Hiring New Developers Midschedule ... 81
4.2.2 Small Project with Requirements Creep ... 88
4.2.3 Small Project Conclusions ... 90

4.3 Medium Project Analysis ... *90*
4.3.1 Hiring New Developers Midschedule ... 92
4.3.2 Medium Project with Requirements Creep ... 96
4.3.3 Medium Project Conclusions ... 98

4.4 Large Project Analysis ... *98*
4.4.1 Large Project Conclusions ... 103

CHAPTER 5 ... **105**

FINDINGS, CONCLUSIONS, AND FUTURE WORK ... **105**

5.1 Findings ... *105*
5.1.1 Knowledge Required ... 106
5.1.2 Debugging ... 108
5.1.3 Maintainability ... 109
5.1.4 Scalability ... 109
5.1.5 Object Representation ... 110
5.1.6 Timing Control ... 112

5.2 Conclusions ... *113*

5.3 Recommendations for Future Work *115*

APPENDIX A **117**

APPENDIX B **120**

APPENDIX C **135**

APPENDIX D **140**

BIBLIOGRAPHY **143**

LIST OF TABLES

Table 1 – Performance vs. CMM Level (King & Diaz, 2000) 13
Table 2 – Simulation Initial Parameters 77
Table 3 – Small Project Simulation Results 81
Table 4 – Results from Hiring 3 Months into the Project (with and Without Requirements Creep) 88
Table 5 – Medium Project Simulation Results 91
Table 6 – Results from Hiring at Day 222 (with and Without Requirements Creep) 97
Table 7 – Large Project Simulation Results 99

LIST OF FIGURES

FIGURE 1 – SCHEDULE FEEDBACK LOOP (ABDEL-HAMID AND MADNICK, 1983) 21
FIGURE 2 – SIMPLE CLD (CANO, 2003) 22
FIGURE 3 – A MORE COMPLICATED CLD (BUSTARD, 2000) 23
FIGURE 4 – FLOW DIAGRAM (BUSTARD, 2000) 24
FIGURE 5 – SAMPLE SYSTEM DYNAMICS OUTPUT FOR BROOK'S LAW MODEL (MADACHY, 2003) 25
FIGURE 6 – SOFTWARE DEVELOPMENT SUBSYSTEMS (P. 22) 27
FIGURE 7 – SOFTWARE DEVELOPMENT SECTOR (P. 78) 28
FIGURE 8 – PARTIAL TYPE HIERARCHY OF EASEL (FISHER, 1999) 31
FIGURE 9 – DEVELOPER ACTOR CODE 34
FIGURE 10 – PROJECT MANAGER ACTOR CODE (CHRISTIE, 2002) 35
FIGURE 11 – SYSTEM DYNAMICS BROOKS'S LAW MODEL: FLOW DIAGRAM 65
FIGURE 12 – BROOKS'S LAW WITH REQUIREMENTS CREEP 66
FIGURE 13 – BROOKS'S LAW MODEL FOR DEVELOPERS BROKEN INTO TEAMS 67
FIGURE 14 – EASEL BROOKS'S LAW MODEL: CLASS DIAGRAM (WITH ENHANCEMENTS) 68
FIGURE 15 – MANAGER CODE FOR HIRING NEW DEVELOPERS 73
FIGURE 16 – PROCEDURE FOR ADDING NEW DEVELOPERS 73
FIGURE 17 – PROCEDURE FOR REASSIGNMENT OF MODULES 74
FIGURE 18 – ASSIMILATION PROCEDURE CODE 75
FIGURE 19 – CLIENT CODE 75
FIGURE 20 – SYSTEM DYNAMICS – HIRING AT DAY 212 82
FIGURE 21 – OVERHEAD FOR BOTH SIMULATIONS 83
FIGURE 22 – SYSTEM DYNAMICS – HIRING AT 3 MONTHS 83
FIGURE 23 – EASEL – HIRING 2 NEW DEVELOPERS AT DAY 212 84
FIGURE 24 – EASEL – HIRING 5 NEW DEVELOPERS AT DAY 212 85
FIGURE 25 – EASEL – HIRING 7 NEW DEVELOPERS AT DAY 212 85
FIGURE 26 – EASEL – HIRING 10 NEW DEVELOPERS AT DAY 212 86
FIGURE 27 – SYSTEM DYNAMICS – HIRING NEW DEVELOPERS AT DAY 423 87
FIGURE 28 – SYSTEM DYNAMICS – HIRING NEW DEVELOPERS AT DAY 635 88
FIGURE 29 – SYSTEM DYNAMICS (WITH REQUIREMENTS CREEP) – HIRING AT DAY 90 89
FIGURE 30 – SYSTEM DYNAMICS (WITH REQUIREMENTS CREEP) – HIRING AT DAY 212 89
FIGURE 31 – SYSTEM DYNAMICS – HIRING NEW DEVELOPERS AT DAY 222 92
FIGURE 32 – OVERHEAD FOR BOTH SIMULATIONS 93
FIGURE 33 – SYSTEM DYNAMICS – HIRING NEW DEVELOPERS AT DAY 445 94
FIGURE 34 – SYSTEM DYNAMICS – HIRING NEW DEVELOPERS AT DAY 668 94
FIGURE 35 – HIRING 5 NEW DEVELOPERS AT DAY 222 95
FIGURE 36 – EASEL – HIRING 10 NEW DEVELOPERS AT DAY 222 95
FIGURE 37 – EASEL – HIRING 15 NEW DEVELOPERS AT DAY 222 96
FIGURE 38 – EASEL – HIRING 20 NEW DEVELOPERS AT DAY 22 96
FIGURE 39 – SYSTEM DYNAMICS (WITH REQUIREMENTS CREEP) – HIRING NEW DEVELOPERS AT DAY 222 97
FIGURE 40 – RESULTS FROM THE LARGE PROJECT SIMULATIONS 100
FIGURE 41 – SYSTEM DYNAMICS RESULTS FOR TEAM SIZE OF 10 101
FIGURE 42 – LARGE PROJECT – HIRING AT DAY 222 102
FIGURE 43 – TEAM DEVELOPMENT RATE FOR A MEDIUM PROJECT 104
FIGURE 44 - COMPARISON OF PROGRAMMING LANGUAGES 106
FIGURE 45 - ADDING A TEAM SIZE FEATURE 109
FIGURE 46 - PERSONNEL LEVELS 110
FIGURE 47 – EASEL BROOKS'S LAW CODE: DEVELOPER PROPERTIES 111
FIGURE 48 - EASEL - HIRING 2 NEW DEVELOPERS AT DAY 90 135
FIGURE 49 - EASEL - HIRING 7 NEW DEVELOPERS AT DAY 90 135
FIGURE 50 - EASEL - HIRING 7 NEW DEVELOPERS AT DAY 90 136
FIGURE 51 - EASEL - HIRING 10 NEW DEVELOPERS AT DAY 90 136
FIGURE 52 - EASEL – HIRING 2 NEW DEVELOPERS AT DAY 423 136

Figure 53 - EASEL - Hiring 5 New Developers at Day 423 137
Figure 54 - EASEL - Hiring 7 New Developers at Day 423 137
Figure 55 - EASEL - Hiring 10 New Developers at Day 423 138
Figure 56 - EASEL - Hiring 2 New Developers at Day 635 138
Figure 57 - EASEL - Hiring 5 New Developers at Day 635 138
Figure 58 - EASEL - Hiring 7 New Developers at Day 635 139
Figure 59 - EASEL - Hiring 10 New Developers at Day 635 139
Figure 60- EASEL - Hiring 5 New Developers at Day 445 140
Figure 61 - EASEL - Hiring 10 New Developers at Day 445 140
Figure 62 - EASEL - Hiring 15 New Developers at Day 445 141
Figure 63 - EASEL - Hiring 20 New Developers at Day 445 141
Figure 64 - EASEL - Hiring 5 New Developers at Day 668 141
Figure 65 - EASEL - Hiring 10 New Developers at Day 668 142
Figure 66 - EASEL - Hiring 15 New Developers at Day 668 142
Figure 67 - EASEL - Hiring 20 New Developers at Day 668 142

CHAPTER 1

INTRODUCTION

The objective of this thesis is to compare, through experimentation and evaluation, two simulation techniques, System Dynamics and Easel, for simulating subsets of software development processes. Each technique is discussed in detail followed by supporting examples of their use in modeling software development processes.

1.1 The Nature of the Software Development Process

According to J. Raynus (1999), a software development project is a social system of interrelated components where developers share a common goal: to manufacture a defect-free software product on time and on budget. He defines a system as formally independent operations, or software processes, performed by those skilled in such operations. Therefore a software development organization is classified as a social system, for it consists of people responsible for allocating resources and performing and regulating activities. As an integrative and complex system that combines both management and software production methods, software development projects often produce unpredictable and problematic results.

These problems contribute to inefficient use of resources. A study conducted by the Standish Group in 2002 reported the failure of over 13,000 projects, about 15% of the total number of projects surveyed (Brock et al, 2003). This result suggests that more attention needs to be directed to software process improvement (Christie, 1999). Improved and optimized software development processes will lead to more predictably project outcomes and to more efficient use of resources. The following sections discuss the inefficiencies typically found in the use of software development resources.

1.1.1 Study of Problems

Two major organizations researched thousands of software development projects: the Standish Group and researchers at Oxford University. In its 2003 annual research report, the Standish Group found that only one third of the 13,522 projects observed were completed on time and on budget with the required features and functionality (Pearce, 2003). Further analysis shows that 70% of the projects were challenged (i.e., overbudget, overschedule, and/or without complete functionality and features), and 15% of these projects were abandoned. Another more recent study conducted by the United Kingdom revealed similar results. Out of 421 projects, only 15% of the projects were completed on time, on budget, and with all specified features and functions (Huber, 2003). One in ten of the projects was abandoned, thus wasting time, effort, and money.

1.1.1.1 Cost. The following statement, taken from a Government Accounting Office (GAO) report that summarized case studies involving software or software-related problems in the military, notes: "We have repeatedly reported on cost rising by millions of dollars, schedule delays of not months but years, and multi-billion-dollar systems that don't perform as envisioned" (Paulk, 1995). Costs exceed budgets due to overrun schedules and defects. As reported by the Standish Group in the same study, challenged projects were over budget by an average 43% (Pearce, 2003). In the UK study, only 41% of the 421 projects observed were within the set budget. Furthermore, the budgets were overrun by an average on 18% (Huber, 2003). As other technologies such as computer hardware decrease in cost, software programs remain both high in cost and low in quality. In 2002 alone, the U.S. spent a total of $255 billion on software projects, but $55 billion of this amount was wasted on cancelled and overrun projects (Pearce, 2003).

1.1.1.2 Schedule. Although schedule predictions have traditionally been based on prior experience, this method has proved inaccurate. One Department of Defense (DoD) software organization reviewed 17 major projects and found that "the average 28-month assigned schedule was missed by an average of 20 months," exceeding the estimated schedule time by more than 70%. (Paulk, 1995) Not a single project reviewed was delivered on time. In the Standish Group study, challenged projects resulted in schedule overruns averaging 82% (Pearce, 2003). In the UK study, only 55% of the 421 projects observed were on time and the average schedule variance was 23% of the estimated schedule (Huber, 2003).

1.1.1.3 Quality. Many organizations' products have high defect rates and a Software Engineering Institute (SEI) survey found that 60% of the organizations studied used inadequate measures of quality assurance (McConnell, 1996, p. 69). In 1986, Capers Jones, the chairman of Software Productivity Research Inc., studied five system codes. His data suggests that the average software defect density of those five systems was anywhere between 49.5 to 94.6% (Russel, 1991). The 2003 Standish Group study reported that in challenged projects only half of the features and functions envisioned for a project were successfully implemented (Pearce, 2003). In the UK study, 54% of the observed projects failed to deliver the requested functionality and features and only 5% of the projects functioned at a higher level than expected (Huber, 2003).

1.1.2 Process Improvement

Management is to blame for these lengthy schedules, increasing costs, and decreased quality. One way to deal with management problems is to shift the development focus from the product to the process. In 1986, the SEI began to coordinate efforts to improve software processes. The SEI partnered with the MITRE Corporation to develop a process maturity

framework, which led to the 1987 introduction of the Capability Maturity Model (CCM) for software (SEI/CMU, 1994, p. 5). "The CMM is based on actual practices, reflects the best of the state of the practice, reflects the needs of individuals performing software process improvement and software process appraisals, is documented, and is publicly available." (SEI/CMU, 1994, p. 6) The SEI not only highlighted the need for more reliable and predictable software development processes, but also provided an accessible, well-documented procedure for improving these processes.

Inefficient or imperfect processes must be understood in order that they might be improved. While increased and active understanding of processes will not eliminate all problems, it will allow for greater levels of prediction, analysis, and control. By shifting focus to the process, CMM practices allow organizations to understand, define, measure, and continuously improve processes. It also allows management to gather data and make conscientious decisions (Raynus, 1999). In another Standish Group report, a number of IT executives established estimates using the following process: "First get their best estimate, multiply by two and then add half" (2001). However, this results in projects that are 150% over budget before they even begin. One motivation behind development of the CMM was to reduce reliance on non-data based estimating practices.

By adopting the CMM, numerous organizations have realized significant improvements in the outcomes of their software development projects. The SEI studied 13 organizations at various maturity levels to determine the overall benefits of process improvement though CMM. The overall average results were (Herbsleb et al., 1994):

- 35% productivity gain per year
- 22% of defects found in pretest per year

- 19% reduction in time to market
- 39% fewer field error reports per year
- 5:1 return on investment

Each level of process improvement provides substantial benefits. The General Dynamics Decision Systems Organization examined the benefits of implementing CMM at various levels for 20 development programs. Table 1 summarizes the General Dynamics CMM experience.

CMM Level	% of Developer Time Dedicated to Rework	% of Defects Contained to Creation Phase	Customer Reported Defects per KSLOC	Productivity
2	23.2	25.5	3.2	1x
3	14.3	41.5	0.9	2x
4	9.5	62.3	0.22	1.9x
5	6.8	87.3	0.19	2.9x

Table 1 – Performance vs. CMM Level (King & Diaz, 2000)

Schedule overruns and error-prone modules, common in the majority of software development systems in the United States, have proven costly. However, by emphasizing process improvement, SEI's CMM has taken strides in improving software development across the globe. In short, process improvement is the key to improving overall software development in terms of cost, time, and quality.

1.1.3 Software Development System Complexity

Software development inefficiency problems can be traced back to more than just product emphasis. Another source of problems is human nature. States Paulk, "Humans beings are fallible" (1995). Mixing people with technology can create a very complex structure that is both

unstable and unpredictable. For example, instead of using quantitative models to make decisions, people often trust their instincts or use previous experience as their sole decision-making basis (Paulk, 1995).

Although human error is to blame for many software development inefficiencies, the interconnected subsystems that are inherent in software development represent another significant factor contributing to development inefficiencies. A system that consists of several interconnected subsystems is referred to as a complex system. These complex systems are often difficult to document, test, and understand. In his paper on the architecture of complexity, Herbert A. Simon defines a complex system as a "system made up of a large number of parts that interact in a nonsimple way" (1981, p. 99). Software development complexity results from large numbers of interacting parts, ranging from humans to pieces of code. Grady Booch states that these complexities exist for the following reasons: the problem domain, software flexibility, and complexity of the process (1993).

Problem domain, or the definition of a problem, plays a vital role in software development complexity. Initial data gathering is a necessary requirement of the software development process. However, this requirement is often burdensome because users and developers will have conflicting views. Even after data has been collected, requirements likely will change. During the development of a typical project, a 25% change in requirements is not uncommon (McConnell, 1996).

Software flexibility also increases complexity. Due to a lack of standards in the software industry, many technologies can be used when creating a software product. This nearly unbounded list of choices compounds complexity.

Developing large-scale software requires a large group of people. Therefore, no single individual or team typically understands the entire system. As more people are added to a given project, communication and coordination increase both in complexity and importance. That is, while the process itself increases in complexity, management must struggle to efficiently distribute resources and oversee development.

Due to the sheer volume of interacting components, and the large number of variables, it is problematic to characterize the behavior of these systems. If a developer makes poor design decisions early in the project, it is likely to affect work at a later date. Design errors may also impact components built by other developers in unforeseen ways. For example, once a design flaw is inserted into a project, it is nearly impossible to predict when the flaw will be discovered and how the flaw will impact the project. Often, rework will need to be undertaken once the flaw is ultimately identified. This illustrates a feedback loop, which can exist in several forms. They can be found between people, between a program and a programmer, and between a project and a process. Feedback loops are difficult for developers and managers to grasp, especially if there is a substantial time delay between the creation of the problem and the resultant effects.

Consequences of these complexities include late project delivery, budget burdens, and an inability to meet user requirements.

1.1.4 The Proposed Solution

The GAO report that identified several categories of software development problems also identified a major problem with the software industry: "The understanding of software as a product and of software development as a process is not keeping pace with the growing complexity and software dependence of existing and emerging mission-critical systems" (Paulk, 1995). Although not all software products fall into the mission-critical category, the lack of

understanding about software development as a process is evidenced by rising costs, missed deadlines, and decreased quality of many software development systems.

According to Alan Christie (1999) of the SEI, "systems are more than the sum of their components." Simulations of software development processes force one to think in global terms and to provide insight into complex behavior patterns. Ultimately, simulations can improve the decision-making process, provide greater insight into development processes, and increase the overall understanding of these processes. Multiple feedback loops results can appear anywhere between minutes and years after the cause. Humans do not have the capacity to predict these results without additional help and, in addition, simulations allow humans to get results within a shortened period of time. Traditional process analysis does not address behavioral issues, and this omission can be corrected through simulation. Christie asserts that with simulation, we can learn about a process without directly observing the process, thereby circumventing potentially costly errors (1999).

1.2 Opportunities and Challenges

Simulations provide users with opportunities to learn about and improve processes within an organization. However, many challenges accompany modeling complex systems.

1.2.1 Opportunities

Creating a simulation involves first defining a problem and then modeling that problem. The model can then be programmed and verified. After the model is verified, the simulation, or computerized model, can be used for experimentation. This is an inexpensive way for

organizations to assess processes when the manipulation of a real system is not possible (Kellner et al., 1999).

Kellner et al. provides many reasons why organizations use simulations:

- *Strategic management* – Simulations aid in making development decisions; for example, they indicate when to create components or when to use commercial off-the-shelf (COTS) products.
- *Planning* – Simulations can be used to predict certain development factors such as effort, resources, and risks. This can then be used to determine the best processes for implementation.
- *Control and operation management* – In order to monitor and control processes, simulations can help in planning milestones and/or determining when corrective action needs to be taken.
- *Process improvement and technology adoption* – Simulations aid in making decisions that will improve processes and also help to determine the effects of adapting a tool.
- *Understanding* – Using a simulation helps one understand what makes a process successful or unsuccessful.
- *Training and learning* – Through the use of simulations, future managers can visualize the impact of classic management mistakes.

The following discussion refers again to the work of Christie (1999) who states that several areas within a system can be improved through simulation. "Simulations can mimic the performance characteristics of software components and their interactions, the effects of time delays and feedbacks, and of finite capacities and resource bottlenecks." One way that developers can capitalize on simulations is to use them to develop and manage requirements.

Quantitative measures can be gathered through simulations to define the initial requirements. Since requirements are in constant flux, simulations allow one to view the results of a possible modification before actual implementation.

As mentioned, individuals are to blame for many, if not all, software development problems. The two most important project estimates are schedule and cost. Simulations provide quantitative results that support improved management decisions.

In software development, learning from experience is expensive. Christie states that, "Simulation can provide considerable insights into how a process will work, prior to its implementation." Because processes can be observed without actual process execution, organizations can observe and assess the potential benefits of adapting and improving a given process. Another benefit of simulations is that they can used anywhere. Allowing a manager to train at a desk versus a remote training location also saves money.

A few other areas in which simulation can be helpful are COTS product adoption, product-line practices, risk management, and acquisitions management. Simulation can be used to examine the effects of utilizing a COTS product, such as resource usage, usability, and timing. Some simulation tools aid in the calculation of product-line costs and help determine which product-line practices provide the most cost effective solutions. Since risks appear throughout the software development lifecycle, simulations can help to predict these risks early on. Christie also states that simulations can estimate the expected progress of contract workers, which leads to honesty and efficiency.

Simulations can also determine the impact of actions, decisions, and environmental factors on a product's success. They can be further utilized to test the effect of certain factors on a system. Simulations may also predict the outcome of certain processes and determine a

system's sensitivity to internal or external factors. Thus, in general, simulations enable their users to illuminate all the relationships within a system, and thus to effectively understand the system as a whole.

1.2.2 Challenges

One challenge associated with software development is the modeling of soft or people variables. Since software is an intangible product, human capability is difficult to model (Abdel-Hamid & Madnick, 1991, p. 119). For example, employee knowledge is difficult to quantify. Thus, to avoid problems and errors with calculation, an arbitrary scale is used to quantify questionable variables (Bustard, 2000).

A technical report on simulation in high-maturity organizations recommended handling soft variables, such as personal attitudes and learning curves, by first scaling them and then correlating them to generate observable quantities (Burke, 1997, p. 42). However, how does an individual estimate whether they are 40% or 95% finished with a particular task? Abdel-Hamid and Stuart Madnick have concluded that, in the early phases of software development, progress can be measured by the "rate of expenditure of resources rather than by the count of accomplishments" (p. 119). For example, if a developer has been allocated 40 days to accomplish a task, it is estimated that he will have completed 25% of the task by the tenth day. During later phases of development, the developer's perception of how much of the task is completed determines progress. Thus, it is possible to model soft variables using a reasonable and scalable approach.

A problem that occurs with novice modelers is that they attempt to model the whole system instead of just a specific problem. Furthermore, simulation development environments

such as iThink make the inclusion of several variables easy, which tends to generate many unnecessary variables.

Not all variables can be effectively modeled. Some concepts that have an effect on a system, such as politics and resistance to change, are challenging to model (Bustard, 2000). The variables that are chosen should maintain a high level of significance. In short, people can make systems less complex if they model with discretion.

1.3 Simulation of Software Development: Two Approaches

Of the several simulation tools available, this thesis examines two existing approaches: System Dynamics and Easel. System dynamics has been available for over 50 years and has proved effective across a wide range of complex systems. Easel is a new approach developed by the SEI that uses property-based programming to simulate software-driven organizations.

1.3.1 System Dynamics

Developed in the 1950's by Jay Forrester, system dynamics is a modeling methodology. Tarek K. Abdel-Hamid describes system dynamics as "application of feedback control systems principles and techniques to managerial and organizational problems" (1991).

1.3.1.1 Background. This modeling technique is based on the idea that the structure of an organization has a direct effect on the behavior of the organization. What distinguishes this modeling methodology from others is that rather than concentrating on the flow of information in a complex system, it also incorporates soft issues such as motivation and knowledge level. By illustrating the total picture of a particular problem, system dynamics allows organizations to understand all the forces influencing its causes. This also furthers an organization's understanding of the relationship between a system and its environment.

System dynamics is an approach commonly used when soft variables are suspected of playing a role in an organization's problems. Determining ways to alter the situation requires a precise measurement of soft variables. However, other problem factors should be examined before choosing system dynamics as a modeling tool.

In order for the system dynamics approach to be suitable for examining a problem or situation, three key attributes must be present:

1. It must be complex.
2. It must be dynamic (involving quantities that change over a period of time).
3. It must contain feedback loops. Feedback is when the action of an individual or things comes back to affect that person or thing again. See Figure 1 for a feedback example.

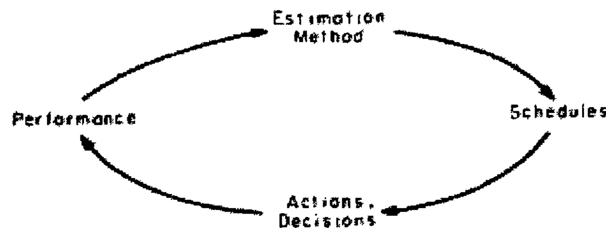


Figure 1 – Schedule Feedback Loop (Abdel-Hamid and Madnick, 1983)

System dynamics is a very good approach for determining the cause and effect of organizational problems. Most individuals find it difficult understand multiple causes and effects when they are separated in time. That is, system dynamics allows people to understand the causes and effects of a problem even when time is a factor.

The steps of the simulation process described above still apply when using system dynamics to simulate a problem area. The initial modeling is done with causal loop diagrams (CLDs), which represent the problem by arrows that show relationships between actions. The model translation, or the program of the model in a computer language, is based on the use of

flow diagrams. These diagrams incorporate equations to represent the system. Creating a simulation using system dynamics requires the creation of causal loop and flow diagrams.

1.3.1.2 CLDs. "CLDs consist of variables connected by arrows denoting causal influence among variables"(Cano, 2003); that is, these diagrams show how one set of variables affects another set of variables. An arrow indicates the movement between variables. The arrowhead has a plus or negative polarity. The plus (+) indicates positive change, while the minus (-) indicates negative change (Houston, 1996).

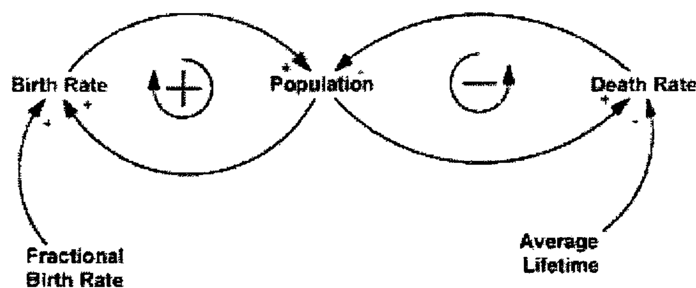


Figure 2 – Simple CLD (Cano, 2003)

The arrows then further connect to create loops. Bustard et al. describe some of the effects that loops can have on a given system. These can demonstrate how one cause may have an escalating or diminishing effect on a system variable. A cause in a loop can also be cancelled out by another cause in the loop. This is called a balancing or negative feedback loop. An increase in action by the loop is called a vicious cycle and a decrease in action is called a virtuous cycle. Individual loops can have both polarities and cycle names.

This modeling technique helps organization members learn about a given problem. To create these models, individuals within the organization must be involved to gain a full understanding of all the variables. Cano comments that a single person within an organization should never create a CLD (2003). Many members should observe the multifaceted dimensions

of a problem to determine the strongest variables. The CLD is likely to have a certain number of variables creating causal loops that interconnect with other loops. As in Figure 3, the more variables that are brought into a model, the more complicated the resulting web.

Figure 3 – A more complicated CLD (Bustard, 2000)

Because CLDs provide an overview of the problem, tracing through the loops often explains the effect of a variable in a given system. However, when time is involved, it becomes difficult to see the overall effect that a variable, or multiple variables, may have on a system. This leads to software programs designed to diagram variables over time.

1.3.1.3 Flow Diagrams. To understand the dynamic changes of variables within a system the use of a modeling tool is needed. Though flow diagrams closely resemble CLDs, they use a slightly different notation method as well as mathematical notation.

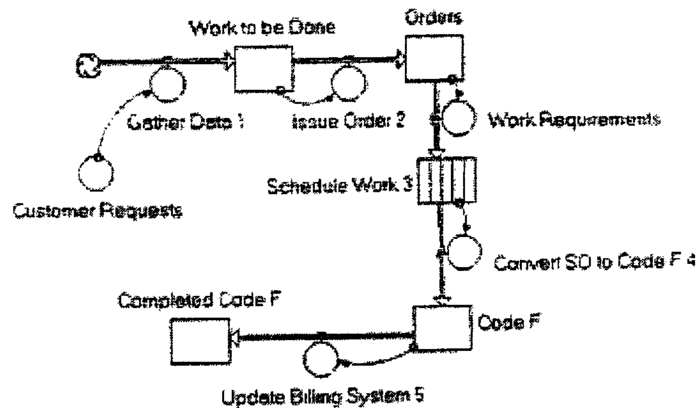


Figure 4 – Flow Diagram (Bustard, 2000)

Bustard et al. describe the different components of flow diagrams and how they relate (2000). In flow diagrams, the square—often called a level—represents an accumulation of physical or logical factors. Flows, or lines, represent the addition or deletion of material stored within the square. The flows often represent human actions or decisions. Houston comments that decision functions, or valves, control these flows (1996). The link between stocks and flows resembles CLDs. Influences, depicted as circles on the model diagrams, act as outside influences on valves. Clouds, which represent sources, denote concepts that lay outside the scope of the problem.

These models and their mathematical representation can then be generated with the use of simulation software. The results of simulation software form a line graph depicting the change in a variable over time.

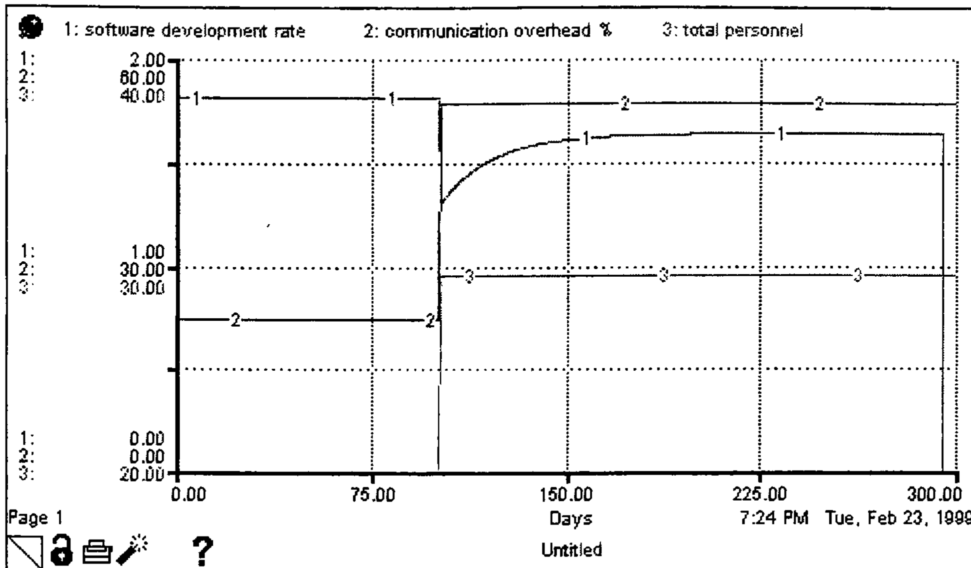


Figure 5 – Sample System Dynamics Output for Brook's Law Model (Madachy, 2003)

1.3.1.4 Advantages. Carr explains why the software development process may benefit from the use of system dynamics (1990). To review, system dynamics requires three attributes: complexity, dynamism, and feedback. Because the software development process is complex, it involves coordinating several variables, including soft variables. The process is also dynamic because the values of these variables change over time. Finally, feedback is present in software development. Because software development satisfies these three requirements, it is a perfect candidate for system dynamics.

System dynamics provides the ability to model several soft variables that impact software development. These soft variables include qualitative factors such as motivation, workload, and advancement opportunities. Using system dynamics allows users to understand how certain decisions or environmental factors affect the software development lifecycle.

1.3.1.5 Disadvantages. There are several reasons why systems dynamics may not be the modeling choice for every complex system. Because several problems exist within the current state of system dynamics, it is likely to face problems in the future.

Some of the cited problems with the current and future state of system dynamics include: problems understanding model behavior and a shortage of quality practices, validating models, and education in the system dynamics field. The resolution of these problems would encourage organizations to adopt this approach.

The advantages and disadvantages of system dynamics are discussed in greater detail in Chapter 2.

1.3.1.6 Mapping the software development lifecycle model using system dynamics.

In their book on software project dynamics, Abdel-Hamid and Madnick examine how the software development life cycle model can be mapped into the systems dynamic model (1991). When simulating a system, the problem area and its scope must be established. They then examined the behavior of scheduling, productivity and staffing, and their interactions. The scope of the simulation, or model boundary, was confined to the development phases including testing. This model represents both developers and projects managers.

In order to effectively represent this complex system, Abdel-Hamid and Madnick divided it into four subsystems: human resource management, software production, planning, and controlling. They assigned each subsystem a certain set of responsibilities. The human resources subsystem was responsible for the "hiring, training, assimilation, and transfer of the project's human resources" (p. 21). The software production subsystem included development, quality assurance, rework, and testing. The information collected by the software production subsystem was subsequently passed to the planning system where schedule changes could be made as needed. Lastly, the controlling system allowed for the comparison between the project's estimates and its current status.

Figure 6 illustrates the communication across subsystems, allowing for a greater understanding of the feedback loops involved and the complexity of the system.

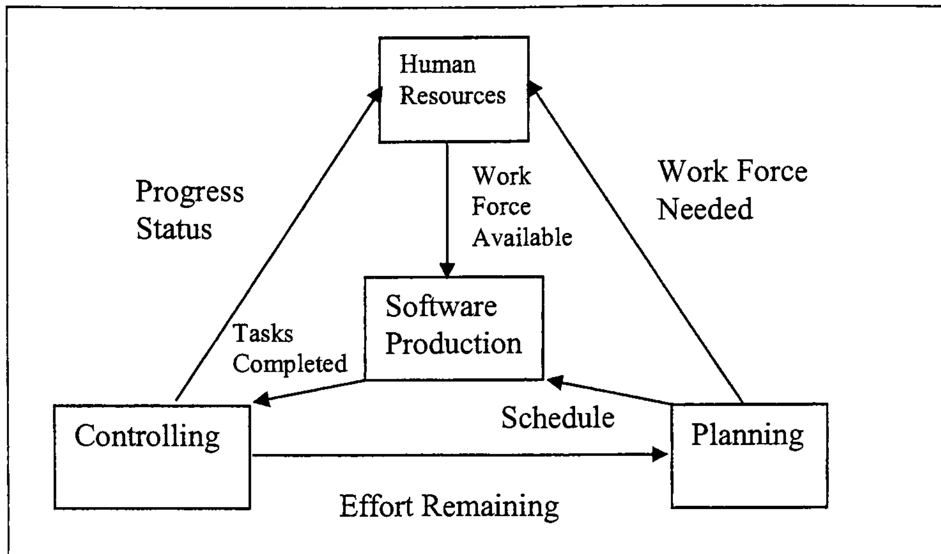


Figure 6 – Software Development Subsystems (p. 22)

To represent variables such as manpower and experienced work force, Abdel-Hamid and Madnick used levels and rates to change these value levels.

Figure 6 is a high-level view of the software development sector. There is one level representing the tasks developed. There is one rate representing the software development rate (SDVRT). There are two auxiliaries: the fraction of effort for system testing (FREFTS) and the daily manpower for software development (DMPSDV). The outside influences, or external variables influencing the subsystem include: perceived job size (PJBSZ), perceived task remaining size (TSKPRM), the daily manpower for software development/testing (DMPDVT), and software development productivity (SDVPRD). Outside influences often affect SDSVT, or the factor determining the level of tasks developed.

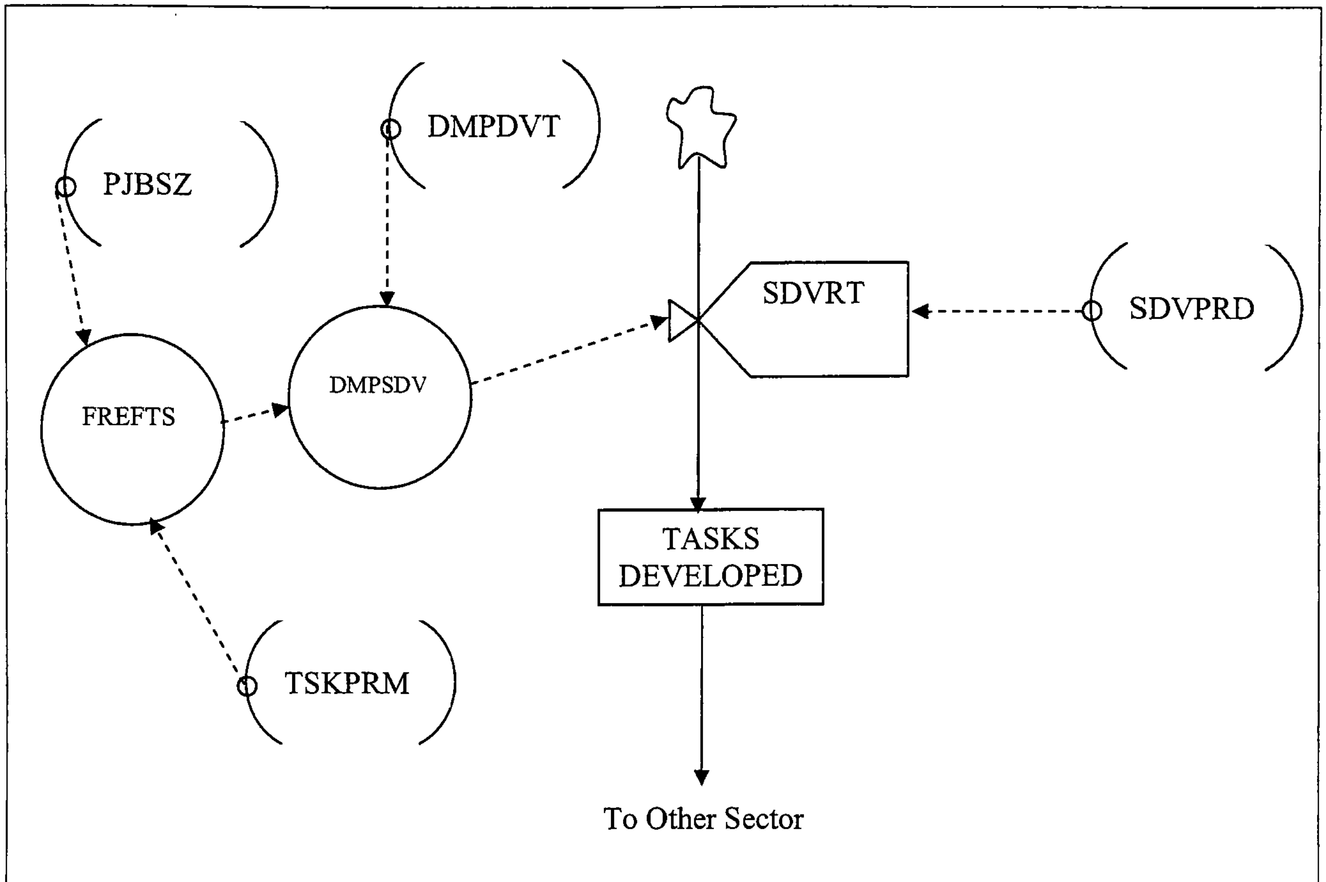


Figure 7 – Software Development Sector (p. 78)

Equations lie behind each of these components. For example, in the human resource subsystem the average employment time can be calculated by a single equation. Information represented in the equation includes the number of experienced work force (L), the time in years (t), and the average employment time in years (T). The following equation is then used to compute the average employment time.

$$L(t)= L(0) \times e^{(-1/T)}$$

Equation 1 – Average Employment Time (Abdel-Hamid & Madnick, 1991, p. 66)

The system dynamics model can simulate many facets of the software development process. In order to model these parts, one must know the variables affecting the simulation and, consequently, the larger problem at hand. When dealing with a large scope or problem area, it may be best to segment the processes into their appropriate subsystems. Subsystem variables

can then be represented through the use of levels, rates, and auxiliaries. Outside variables can also be represented.

The advantages and disadvantages of system dynamics along with its application to software development projects are further discussed in Chapter 2.

1.3.1.7 Prior applications of system dynamics to software development. Several individuals including Abdel-Hamid and Madnick have conducted studies where system dynamics was used to simulate software development organizations. The studies elaborated on in Chapter 2 include the following:

- *Software Project Dynamics: An Integrated Approach*, by Abdel-Hamid and Madnick (1991), explains system dynamics via its application to a problem in a software development organization.
- *System Dynamics Modeling and Simulation of Software Development* is a tutorial by Dan Houston (1996) that explores the simulation of systems, and then explains the system dynamics approach. Houston later uses diagrams by Abdel-Hamid to show how software development can be modeled using system dynamics.
- "Software Process Simulation Modeling: Why? What? How?" by Marc I. Kellner, Raymond J. Madachy, and David M. Raffo (1999) takes a look at why simulating software processes is worthwhile, what should be simulated, and how it should be simulated, with an emphasis on system dynamics.
- A Ph.D. dissertation proposal by Ioana Rus (1997) of Arizona State University discusses modeling problems that occur with software quality and how these problems can be modeled using system dynamics.

1.3.2 Easel

Easel, Emergent Algorithm Simulation Environment and Language, emerged from survivability research conducted by the SEI. System survivability describes a system's ability to withstand attacks, failures, or accidents to fulfill its purpose (Fisher, 1999). A system can be anything from a network to an actual computer. Easel simulations can operate under limited visibility, that is, with actors who are unable to see beyond the local scope of an action, an inability which ultimately leads to emergent behaviors.

1.3.2.1 Background. The idea behind emergence is that local, simple actions can produce global patterns without the help of administrative authorities (Fisher, 1999). These patterns emerge naturally. Easel helps to predict these patterns because it is able to represent a number of actors, or nodes. Much work has been done at the SEI with Easel to produce network patterns, but it is widely believed that Easel may also be used to predict software development patterns.

Easel may be used with almost any system that has emergent properties. It can be used to represent a dynamic and complex system, or a system with a multitude of variables. It can also be used to represent systems influenced by visibility between objects, where the consequences of local actions are hard to determine. A node's view of an organization or system is often restricted to a certain area, which Christie and Fisher refer to as "unboundness." Easel differs from other software development modeling software, because it can actually alter the topographical relationship between actions during simulation. To model these unbounded systems, Fisher believes that, "a loosely coupled multiprocessing model with near neighbor communication with parallel semantics is needed" (1999). Most simulation languages use a shared memory model with interleaved semantics instead (Fisher, 1999).

No single modeling technique is recommended for Easel. It is a property-based language and deals with property-based types, otherwise known as object-oriented language classes. Because everything in Easel is some form of a type, Easel supports many built-in types as well as the creation of additional types. Part of the type hierarchy is shown in Figure 8.

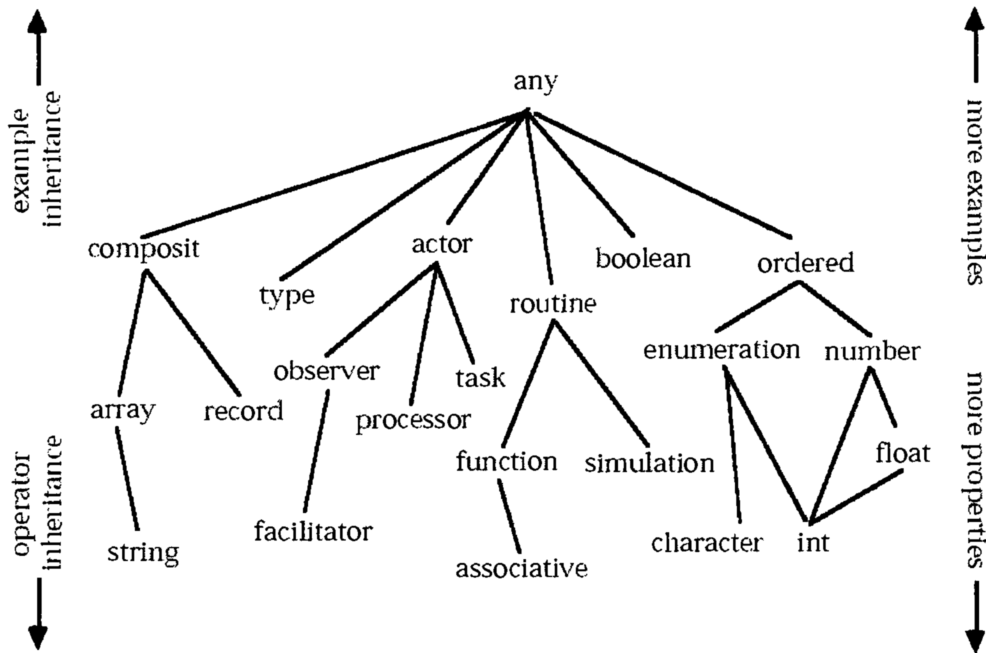


Figure 8 – Partial Type Hierarchy of Easel (Fisher, 1999)

One of these types, the actor, can be used to simulate entities in the real world. Actors can be used to represent almost anything, from a system administrator to an ant. Actors are unique in that their behavior is threaded and they have "neighbor" relationships. Since there is no global visibility, it is possible to define the relationships between actors. This actor can then affect the state of other actors, such as an observer, whose main purpose is to observe and then gather data from the simulation. Another actor is a facilitator whose job is to set up and control the simulation. When the simulation is executed, these actors begin and end their work in parallel-simulated time (Fisher, 1999).

Easel is very similar to other languages because it has many of the same data types and control structures. The Easel Language Reference Manual and Author Guide, created by Christie, Durkee, Fisher, and Mundie at the SEI, explains Easel's language structure and available types. It also provides examples useful to Easel programmers.

1.3.2.2 Advantages. In addition to the advantages of Easel mentioned above, Alan Christie and David Fisher (2000) state many reasons for Easel's attractiveness as a simulation tool. Among these are the following:

- It can support a large number of actors
- It has many conventional programming language features
- It simulates processes without all information present
- It provides many statistical functions
- It includes a simulation tool known as graphic visualization
- It supports an interactive user interface
- It provides a bird's eye view of a system

Its main advantage is the ability to show topographical relationships and provide actors with a limited view of data, which can lead to emergent behaviors.

1.3.2.3 Disadvantages. Because Easel was only recently developed, few case studies are available for review. Currently, the only resources about Easel available to developers are a limited number of published papers, the individuals responsible for creating Easel at SEI, and the forums hosted at the CERT Web site. However, information about Easel may increase with its recent production release.

The advantages and disadvantages of Easel are discussed in greater detail in Chapter 2.

1.3.2.4 Mapping the Software Development Life Cycle Model Using Easel. In their paper on the use of simulation in complex software-intensive organizations, Fisher and Christie support the claim that Easel can be used to simulate software development (2000). Because they define software development organizations as systems that exude emergent behaviors, Easel presents an ideal modeling tool. Individuals within these organizations can function without complete knowledge of the entire organization.

Easel provides the opportunity to model the factors that play a role in software development. One of the attractive qualities of Easel is that, through the use of actors, it can model a large number of physical world entities. These actors range from project managers to quality assurance team members. Thus actors allow for each person involved in the software development lifecycle to play a role in the simulation. The actors, like members of an organization, can work simultaneously, come into emergence late in the process or leave in the middle of the process, exhibit separate behaviors, and have a relationship with their neighbors. The tasks that the software developers are working on can also represent an Easel type. These types can both have attributes and be seen on a global scale and, if necessary, be hidden from other developers.

Figure 9 shows Easel code that defines developers (Christie, 2002).

```
developer(s:sm): actor type is
# developers implement modules
        devID: int := ?;
        mod::module := ?;
        num_mods::int := 0;
        devT::number := 0.0;
        modList:: list := new list module;
        modList1:: list := new list module;
        dev_state:: dev_states:= free;
        tm:number := 0.0;

        for every true do
                if (length modList) > 0 & dev_state = free then
                        mod := pop modList;
                        push(modList1, mod);
```

```
                              dev_state := occupied;
                              mod.startT := devT;
                              mod.endT := devT+mod.dev_time;
                              devT := max(mod.endT, tm);
                              num_mods:=num_mods-1;

                              outln("dev ID: ", devID, "   mod
ID: ", mod.modID, "   proj ID: ", mod.proj.projID);
                              outln("start time: ", mod.startT, "
end time: ",mod.endT, "   mods remaining ", num_mods);

outln("------------------------------------------------------");
                              drawMod(s, self, mod);

                              wait mod.dev_time;
                              mod.mod_state:= completed;
                              dev_state := free;
                        else
                              tm:= tm+s.dt;
                          wait s.dt;
```

Figure 9 – Developer Actor Code

In this code, the developer is first declared an actor. As indicated above, the developer has several different attributes including the following:

- A developer ID
- A module or task that the developer is currently working on
- A number of currently assigned modules
- A current development time
- A list of assigned modules
- A list of modules completed
- A development state defined as either "free" or "occupied"
- A current time

Below these attributes, Christie lists the action of the developer. If the developer is free and has several assigned modules, she must work on a module of her choosing. If there is no assigned work, the developer will simply wait. Like the developer, the module will also have attributes

such as an ID, a state, a development time, a start and end time, and an umbrella project to which it belongs.

Easel creates any entity requiring representation in a simulated module. Then, through an actor that represents a project manager, these entities are brought together as they would be in the real world. Figure 10 illustrates this bridge.

```
manager(s:sm): actor type is
# managers assign modules
        prj:project :≈ ?;
        for every true do
                for prj: every s.projList do
                        if prj.prj_state !≈ closed then
                                if prj.prj_state ≈ unallocated then
                                        assign_mods_to_developers(s, prj);
                                        prj.prj_state :≈ allocated;
                                else if prj.prj_state ≈ allocated &
all_mods_completed(s, prj) then
                                        prj.prj_state:≈ mods_completed;
                                        outln("all mods completed for
project ", prj.projID);

outln("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
                                else if prj.prj_state= mods_completed then
                                        prj.prj_state :≈ closed;
                wait s.dt;
```

Figure 10 – Project Manager Actor Code (Christie, 2002)

The code indicates that a manager, who can be assigned to one or more projects, distributes the tasks or modules to the developers on the team. The project manager can also monitor developers' progress. Because they are each provided threads of control, these actors can then run simultaneously and can communicate as needed.

With the ability to represent individuals, their tasks, and the appropriate attributes, Easel can simulate the software development life cycle. Individuals such as designers, developers, testers, quality assurance personnel, and managers are represented through actors, and the tasks they are working on can be represented through Easel types. Actors can interact socially and certain actors can monitor particular tasks. By factoring project development and representing

those involved with each stage of development, Easel realistically portrays what happens within the software development life cycle at several levels.

1.3.2.5 Prior Applications of Easel to Software Development. Christie and Fisher have investigated Easel and its application to the software development field. The following documents discuss their work with Easel and how it may be used to simulate the software development life cycle. Other studies are discussed in greater detail in Chapter 2.

- Christie and Fisher (2000) presented their paper, "Simulating the Emergent Behavior of Complex Software-Intensive Organizations," at the ProSim 2000 Workshop in London, England. It discusses the advantages of using the Easel language and why it can be used to model the software development life cycle.
- Christie and Fisher also created a PowerPoint presentation, "Easel—A New Simulation Language and Its Application to Software Process," that discusses the need for Easel, its language design, and its application to software development processes.

CHAPTER 2

LITERATURE REVIEW

2.1 System Dynamics Literature Review

Jay W. Forrester, Germeshausen Professor Emeritus and Senior Lecturer at MIT's Sloan School of Management developed the Systems Dynamics method. In his book, *The Beginning of System Dynamics* (1995), Forrester discusses the birth of the system dynamics field.

2.1.1 Beginning of System Dynamics

Forrester started the System Dynamics Group at MIT in 1956. The first application of system dynamics, known as industrial dynamics, emerged when Forrester joined with General Electric to discuss its recurring problem with employment instability. This problem proved particularly troublesome considering the high demand for GE products. Using only a pencil and paper and the information that he had gathered about hiring and inventory policies at General Electric, Forrester conducted a simulation. With this simulation, he determined that the system's instability was due to poor decision making at the management level.

Forrester was later asked to join the Digital Electric Corporation's board of directors, where, through the modeling of organizational structures, he gained insight into the nature of high technology corporations and expanded his choice of modeling variables (Forrester, 1995). In addition to looking at physical variables, he recognized that system dynamics techniques could be applied to organizational policies. In *Industrial Dynamics: a Major Breakthrough for Decision Makers* (1958), the first published report describing system dynamics, Forrester discussed how "industrial company success depends on the interaction between the flows of

information, materials, money, manpower, and capital equipment" (Roberts, 1978, p. 37). Forrester then published *Industrial Dynamics,* which was intended to be used as a textbook.

As noted in the book *Urban Dynamics*, Forrester began applying the system dynamics approach to urban dynamics in 1968. Although *Urban Dynamics* explored sensitive topics, the book was well received and demonstrated general applications of the system dynamics approach. The book also showed that the system dynamics technique could be applied to world dynamics. In 1970, Forrester was invited to a Club of Rome meeting. The Club of Rome is a nonprofit group that consists of various members from an array of professions that study global trends while avoiding the influence of political and monetary concerns. Using the issues discussed at that meeting, he built a model that appeared in his next book, *World Dynamics*. This book was widely acclaimed for illustrating the versatility of system dynamics and influenced the publishing of *The Limits of Growth* (Meadows, 1972), where a world model was utilized to examine "accelerating industrialization, rapid population growth, widespread malnutrition, depletion of nonrenewable resources, and a deteriorating environment" (Pestel, 1972). This model produced eye-opening conclusions culminating with the claim that the earth cannot withstand present growth trends which will eventually produce a catastrophic decline in population and industry (Pestel, 1972).

For years, Forrester had recognized the flexibility of industrial dynamics, and its ability to model large, complex systems (Edwards, 1997). Thus, to reflect the system's wide range of capabilities, he officially changed its name from industrial dynamics to system dynamics.

Currently, system dynamics research is in a state of rapid growth. Forrester is still actively involved in this field and continues to teach system dynamics at MIT, reaffirming that system dynamics provides a generalized approach to problem solving.

2.1.2 Applications of System Dynamics

System dynamics has been applied to many fields. Forrester's initial work has shown that system dynamics modeling and simulation can be applied to corporations (Forrester, 1962), the public (Forrester, 1969), and global industries (Forrester, 1971). Examples of the scope of system dynamics include its application to: corporate management, process improvement, economics, biology, medicine, energy and the environment, theory development (Dill, 1997), dynamic decision making, supply chain management, and software development. Several system dynamics application descriptions are presented below.

2.1.2.1 Corporate management modeling. Several system dynamics applications and case studies as they apply to corporate management are available. The texts *Management System Dynamics* and *Application System Dynamics* illustrate system dynamics application to corporate problems.

Management System Dynamics (Coyle, 1977) applies system dynamics to the corporate world. In his discussion of socio-economic systems, Coyle describes how the forces that cause dynamic change can be better understood when they are modeled as a system. The book also includes several system dynamic management case studies.

The other text, *Managerial Applications of System Dynamics* (Roberts, 1978), summarizes system dynamics tutorials based on early articles in the field. The book provides many models and sample code in DYNAMO, a simulation language, for system dynamics simulations.

Several other papers highlight system dynamics real-world use in manufacturing, marketing, and management. In a 1984 article in the *Strategic Management Journal,* J.D.W. Morecraft shows how simulations can "act out the consequences of strategy proposals in their

full organizational setting" (p. 215). He emphasizes how system dynamics models can help executives both make decisions and understand overall corporate strategy. System dynamics has been used to "support project bids, to identify risks, and to assess the benefit of several process and organizational changes," (Lyneis, Cooper, & Els, 2001, p. 237) allowing corporations to learn from previous complex projects in order to improve future performance.

Because system dynamics models allow managers to make better business decisions by reliably predicting factors such as demand, revenues, and profits, they provide management with a better understanding of industry behavior (Lyneis, 2000). Managers can then enforce polices and make decisions that improve an organization's overall performance.

These journal articles have widely supported the system dynamics approach and its application to corporate management. System dynamics enables management to make smarter decisions when deciding "which aspects of the performance of the process are to be measured and how these measurements are to be used to change the level or resources utilized in the process" (Powell, Schwaninger, & Trimble, 2001, p. 63). However, these articles also emphasize the need to improve business processes using the system dynamics approach. Thus, in short, although system dynamics has allowed management to explore the effects of modifying a single process, there remains room for improvement within business processes.

2.1.2.2 Biological and medical modeling. System dynamics has also been applied to the health field in an attempt to manage epidemics, thereby reducing the risk of health-related disasters. It has helped managers in the health field "understand the impact of alternative strategies for addressing disasters such as national epidemics," (p. 119) and was proven effective when it helped to control an outbreak of dengue fever in Mexico (Ritchie-Dunham & Galvan, 1999). System dynamics modeling can also inform healthcare officials on how to make the best

possible decision when faced with the outbreak of an infectious disease. For example, system dynamics has provided healthcare managers with the ability to understand the impact of certain policy changes and has taught them how to create scenarios portraying "future trends in reported AIDS cases" (Dangerfield & Roberts, 1999).

Furthermore, research is not limited to epidemics of infectious disease. In an article published in 2002, Tarek Abdel-Hamid explored the threat of an obesity epidemic by creating a model that reflected an individual's weight gain and weight loss. Using system dynamics, he included several variables, such as metabolism and exercise. Among other observations, he discovered that diets loaded with carbohydrates can still result in significant weight loss.

2.1.2.3 Energy and environmental modeling. In a few cases, system dynamics has provided a greater understanding of the environment. Previously, fractured bedrock flow was difficult to represent mathematically, but the coupling of system dynamics modeling with field and laboratory data, has allowed researchers to develop equations that accurately represent this complex system (Abbot & Stanley, 1999). System dynamics has also been used to illustrate the public's ignorance about global warming (Sterman & Sweeney, 2002). In a study that used system dynamics models, students were asked to predict the environmental impacts of $C0_2$ on different scenarios. The results showed that even highly educated students were unaware of the effect of $C0_2$ on the environment and its connection to global warming.

2.1.2.4 Dynamic decision making modeling. As we have seen, system dynamics can aid decision making in the corporate management arena. However, it can also be used to model human decision making in general. The Carnegie School conducted a study illustrating how system dynamics can be used to simulate the human decision-making process (Morecroft, 1983). This case study "reviews and contrasts the concept of bounded rationality as developed by

Herbert Simon" (Sastry & Sterman, 1992). Another study on human decision making at MIT linked the human decision-making process to the production of chaos (Sterman, 1989).

2.1.2.5 Supply chain management modeling. Some time after Forrester explored supply chain management in *Industrial Dynamics*, the use of system dynamics in supply chain management has begun to regain popularity (Angerhofer & Angelides, 2000). On a broader scale, studies have explored the application of system dynamics to international supply chain management strategies (Akkermans, Bogerd, & Vos, 1999) and "observed [the] road blocks" hindering international supply chains. Other studies have simulated emergent supply networks through the use of agent-based modeling and system dynamics (Akkermans, 2001). These studies have shown that system dynamics is important when dealing with inventory, management, and policy decisions. Further system dynamics research is needed to effectively combat the problems prevalent in partnership supply chains.

2.1.2.6 Software engineering modeling. Several studies have tested the application of system dynamics in software-development organizations. Abdel-Hamid & Madnick's *Software Project Dynamics: An Integrated Approach* (1991) uses system dynamics to simulate various sectors of a software-intensive organization and discuss the lessons learned through system dynamics modeling. *The Dynamics of Software Project Scheduling* (1983), an article also by Abdel-Hamid and Madnick, looks at the system dynamics approach in order to take a closer look at the variables that influence a software project's schedule. Tvedt's Ph.D. dissertation on process improvement (1996) explores various simulation techniques, with an emphasis on system dynamics, in order to explore various process-improvement approaches. Ioana Rus's Ph.D. dissertation proposal on software quality (1997) discusses modeling problems that occur with software quality and how these problems can be modeled using systems dynamics. Kahen,

Lehman, and Ramil's paper (2000) on long-term software management uses system dynamics to explore process improvement over a lengthy period of time while focusing on crucial project areas. These studies focus on specific processes, such as quality assurance and inspection, as well as policies dealing with variables such as scheduling and costs. A few of these software-development-related system dynamics studies are discussed below.

2.1.3 Organizational Studies

Tarek K. Abdel-Hamid and Stuart Madnick developed a system dynamics model for software development systems (Abdel-Hamid, 1984). In 1990, they published a case study that applied a system dynamics model to NASA's DE-A software project.

The DE-A software system was designed "for processing telemetry data and providing attitude determination and control for the DE-A satellite" (p. 40). A full 85% of the actual project budget was allocated to software development, and the remaining 15% to testing. Thirty percent of the development cost was allocated to quality assurance (QA). As this project needed to be finished on time, schedule slippage was not tolerated. Thus, when the project began to fall behind schedule, management added a number of new people to the project. In the end, although the project was larger and more expensive than expected, it was delivered on time.

System dynamics modeling allowed Abdel-Hamid and Madnick to explore several important topics. They were able to experiment with a previously developed model to determine the roles that staffing and QA played in this project. The model for software development had four subsystems: control, which included factors such as productivity and perceived project size, planning, which included factors such as schedule pressure and forecasted completion date, software production, which included factors such as error rate and actual productivity, and human resource management, which included factors such as hiring and turnover rates.

When experimenting with staff increases and decreases, Abdel-Hamid and Madnick discovered that Brooks's Law does not apply in every situation. Brooks's Law, which states that adding manpower to a late project only serves to make the project later, is discussed in more detail later in this chapter. They concluded that "the drop in productivity must be large enough to effectively render each additional person's net cumulative contribution" (p. 42) in order for Brooks's Law to be applicable. They ran several test cases using a constant 380-day schedule, and estimated that the project could be completed with 234 man days, thereby decreasing 10.6% of the total project cost.

They also tried lowering the time and budget spent on QA to determine the optimal amounts. They concluded that NASA could achieve the same level of quality even if they lowered their original 30% QA budget allotment to 15%.

Abdel-Hamid and Madnick hoped to identify some of the problems within the software development process to enable others to learn from NASA's mistakes, thus reducing the chance of similar errors in the future. They stressed that hiding from mistakes is detrimental to progress and that system dynamics simulating can provide a method by which to learn from mistakes.

In a study conducted by Raymond J. Madachy in 1996, the software development process was modeled to examine the effects of inspections on a project's factors such as cost and schedule. This model was intended to establish a "baseline for benchmarking process improvement" (p. 376), to collect data on different software development process tradeoffs, and to provide managers with tools enabling them to make streamlined plans.

To develop an accurate model, Madachy conducted an "extensive literature review, analysis of industrial data, and expert interviews" (p. 377). The scope of this model, which begins and ends with the appropriate phases of the waterfall life cycle, simulates only the

software development, quality assurance, and testing sectors. Madachy collected the data for this model from both Litton Data Systems and expert interviews with Litton personnel.

The model reflected activities starting with the design stage and ending with system testing. Inspections and system testing were the only methods of defect removal. It is worth noting that inspection-training costs were not included and that errors were introduced in both the design and coding phases. Variable input consisted of "job size, productivity, schedule constraints, and resource leveling constraints" (p. 378).

The model was then calibrated using the COCOMO model and the data collected from Litton Data Systems. Testing of the model, which dealt with factors such as error generation rates, job size, and schedule compression, was conducted using test cases and Litton data.

The results in some studies showed that inspections added about 10% effort in both the design and coding phases, while reducing test phase efforts by about 50%. Error generation rate test studies showed that "if there is a low defect density of inspected artifacts, then inspections take more effort per detected error and there are diminishing returns" (p. 381). The simulation showed that inspections were not beneficial when defects/KSLOC fell below 20%.

Simulation test cases also showed that a "fixed staff size entails a longer project schedule by about 30%" and schedule compression showed that "average personnel level increases and the overall cumulative cost goes up nonlinearly." Overall, it was shown that this model is scalable on several levels and shows a significant return on investment when inspections are introduced only after considering aspects such as "phase error inspection rate, error amplification, testing error fixing effort, and inspection efficiency." In his study's conclusion, Madachy suggests exploring many more project variables such as model validation and variations on staffing policies.

Gordon E. McCray and Thomas D. Clark Jr. explored outsourcing options using a system dynamics model (1999). Their model was created to quantitatively determine whether or not an IT organization should build or buy a product. The model, which depended on the organization structure, included factors such as "organizational policy, perceived benefits, and levels of knowledge," as well as market stability and competition.

The model they created included five sectors based on the "logical grouping of system variables," which included the following: internal software acquisition sector, external software acquisition sector, internal hardware acquisition sector, external hardware acquisition sector, and boundary management sector. The internal software acquisition sector included variables such as how the use of internal resources can meet the demand for an application. The external software acquisition sector included variables such as how outsourcing can impact internal variables. Both hardware sectors contain variables for processing and staff resources. The final sector, boundary management, consisted of variables that drive the other model sectors, such as knowledge transfer rate and cost control pressures.

Through experiments, the authors researched the "impact of organizational and environmental factors on the outsourcing decision" and "whether long-term benefits accrue to those firms that elect to outsource" (p. 358). They experimented with the model during two independent phases: the outsourcing of application development work phase and the outsourcing of processing capabilities phase. Some of the variables critical to the simulation included the technology market stability and competitive stability. Lesser variables included management pressures to control costs or retain control over development.

Simulations for application development outsourcing were run under both the encouragement and discouragement of outsourcing hypotheses. The experiment showed that

"high levels in both the technology market and the competitive market also precipitate low levels of outsourcing of development work" (p. 360). However, when there is management pressure to retain costs and internal control is relatively low, the likelihood of outsourcing increases significantly.

Overall, the experiments indicate that "success of the outsourcing is heavily dependent upon management policies" (p. 370) and that the wide adoption of outsourcing will ultimately result in lower service levels. The latter is due to the expected cost decreases associated with outsourcing. In other words, projects will be allocated fewer funds, which will result in lower service levels. This study sheds light on the advantages and disadvantages of outsourcing as well as all the variables that affect the buy or build decision.

A similar case study focused around an Australian service organization called Gigante (Bustard, Kawalek, & Norris, 2003). Though Gigante was releasing new products at ever increasing rates, its overall product quality was decreasing. They blamed the process, but remained unsure about why the process was failing.

In an attempt to solve the problem, a system dynamics approach was taken. First, researchers created data flow diagrams by interviewing Gigante employees. Then, they recorded the factors associated with tracking and preventing errors such as defect rate, customer satisfaction, training hours, and knowledge gap. They then created a system dynamics model with iThink that used all of the appropriate observed variables, including many soft variables. The apparent sectors included: training, knowledge base, workforce, and error detection and correction.

Running the simulation singled out variables affecting the error rate, such as the period between product releases. If the period was too short, the appropriate training of new staff

became nearly impossible and error rates increased. The number of staff also played an important role. For example, if an organization is short-staffed, employees cannot fix errors because they are overworked.

System dynamics modeling has shed much light on the IT and software development industries. Thus, knowledge about processes within these fields will continue to grow as system dynamics expands.

2.1.4 Tutorials

A recent text on systems thinking is Gharajedaghi's *Systems Thinking: Managing Chaos and Complexity* (1999). Gharajedaghi highlights several system principles, including the emergence property and dimensions. His examples also employ several types of modeling and relevant case studies. He compares learning systems thinking to a chess game, where the rules are simple yet the outcome is often unpredictable. Another useful book for understanding the importance of systems thinking in management is Haines' *The Systems Thinking Approach to Strategic Planning and Management* (2000). Haines uses systems thinking to provide a step-by-step guide to organizational improvement.

After becoming acquainted with systems thinking, the next step is to create a system dynamics simulation. To do this, one must learn about feedback loops, system dynamics modeling, and simulation tools. Several texts are available about one or more of these topics.

Dan Houston of Arizona State University (ASU) created a brief tutorial (1996) on system dynamics modeling, particularly its uses with software development. This brief tutorial discusses systems thinking, the overall simulation process, and the basics of creating a simple system dynamics model. Houston completes the tutorial with an explanation of the software development model created by Abdel-Hamid.

Another publication from ASU (Kirkwood, 1998), which provides a brief overview of system dynamics modeling and simulation procedures, is designed to help novice users model business processes. This tutorial begins with a discussion of systems thinking and then outlines basic modeling approaches. By going into greater depth than Houston's tutorial, it provides the reader with more insight on the modeling process.

The System Dynamics Education Program under the direction of Forrester, created an online resource known as Road Maps (SDEP: Road Maps). This program, a nine-chapter user's guide to system dynamics, is recommended for both beginners and advanced system dynamics users. The program, which discusses everything from feedback loops to real-world examples, functions as an all-inclusive study guide. As such, it includes many examples and cites many papers dedicated to the study of this modeling technique.

A reference for complex system simulation using the simulation language DYNAMO is the book *Introduction to Computer Simulation: a System Dynamics Modeling Approach.* (Roberts, Anderson, Deal, Garet, and Shaffer, 1983) The book covers everything from the basics of system simulation and feedback loop dynamics to the development of more complex models. As such, a wide variety of complex models are demonstrated throughout the book. The simulation code is limited to the DYNAMO software package code.

Thus, there are several systems thinking and systems dynamics modeling and simulation tutorials and texts available for beginners. Each text provides readers with a basic understanding of system dynamics models, allowing them to gain an understanding of the system and develop an appropriate model for improvement.

2.1.5 Advantages

System dynamics's popularity is mainly due to its ability to model complex, nonlinear, and dynamic systems. However, several other reasons why the system dynamics approach has become popular are discussed below.

System dynamics has become widely accepted due to its ability to model a variety of systems and factors, including soft factors which contribute to the increase in a number of problems (Bustard, Kawalek, & Norris, 2003). Other modeling methodologies are not ideal for modeling soft factors such as knowledge base and schedule pressure. The relationships between soft and hard variables can also be examined through system dynamics simulation.

An organization's structure and policies might lie at the crux of a system's problems. Because system dynamics has the capability to illustrate the relationship between processes and the organization, it can be used to teach management policies through existing case studies. This will hopefully lead to better organization design (Legasto, Forrester, & Lyneis, 1980). System dynamics can also be used as a training tool, where management explores the impact of important organization policies and decisions. Furthermore, the visual nature of a system dynamic simulation's output facilitates comprehension.

2.1.6 Disadvantages

Due to the rapid growth of system dynamics over the last 40 years, a few problem areas have emerged. In an article published in 1996, George P. Richardson, editor of the *System Dynamics Review,* provides insight into some current and future problems with system dynamics. While these problems do not provide an exhaustive list, they do represent some of the most pressing problems in the field today.

The first problem Richardson discusses is model behavior. Because it is difficult to model a complex system, the system model must be developed iteratively. That is, each successive model increases complexity. This is a method by which to understand model behavior, but it is very time consuming. Because the system dynamics field lacks the technical support needed to make "the connections between model structure and behavior," the difficulty of model creation increases.

Richardson also notes that documentation on models is limited and remains largely inaccessible. Furthermore, no books or workshops deal with advanced modeling concepts. Richardson also points out that there is no accumulation of the results in the system dynamics field. Without an accumulation of the results in the system dynamics field, scholars cannot build on previous work but must reinvent old processes. Finally, because the field lacks a best practices collection, there is no organized system to aid novice modelers.

Another problematic issue that Richardson notes is the inability to determine when to use qualitative mapping versus formal modeling. Furthermore, once a model has been created, it should be validated. Currently, the most "comprehensive statement on model validation and user confidence in system dynamics modeling" is over twenty years old. The field needs new documentation on validation methods to rebuild eroding confidence.

The future of system dynamics relies on its ability to "widen the base" or educate the population, to understand the basic principles of system dynamics. Though some work has been done in this field, particularly at the high school education level, more work needs to be done to avoid future problems.

Other studies have shown different problematic issues. In the Gigante study (Bustard, Kawalek, and Norris, 2003) the authors noted a wide variety of potential problems, such as

problems concerning system conceptualization and modeling. There was also a "tendency to model the system rather than model the problem." Other problems followed, such as how to determine the level of the system to be modeled, the relevant software variables, and the proper modeling of those variables.

Because there is a steep learning curve for those who wish to learn system dynamics, organizations may not be willing to spend the time and money needed to effectively train employees in this system. Furthermore, the authors of the Gigante study found that "many of the recommendations are contrary to normally accepted management policies." Thus, the biggest concern for most system dynamics supporters is how to convince management to use the recommendations acquired from a given simulation.

Authors such as Forrester, Legasto, and Lyneis (1980) have noticed other problems with system dynamics such as model definition. Model definition, including scope and variables, remain hard to determine. Scholars disagree over what constitutes important feedback and accurate measurement techniques. The validation of models remains a hot topic for many researchers, yet represents a problematic area for system dynamics. For example, the model used for *World Dynamics* has yet to be validated.

Before the number of organizations adopting system dynamics can increase, these and other problems need to be addressed.

2.2 Easel Literature Review

Easel is a new simulation language developed at the Software Engineering Institute. Easel belongs to the class of actor-based languages and has the capability to simulate emergent behaviors in large-scale systems. Although Easel's development was motivated by a need for

advanced studies in network survivability, the language is well suited for software development process simulation.

2.2.1 Actor-based Languages

Actor-based languages became popular in artificial intelligence (AI) communities in the early 1980s. Actors represent a single entity containing a knowledge bank as well as the ability to communicate with other actors and run concurrently with other actors . Prior to the AI community's adoption of the term "actor," actors were previously referred to as "objects" The first actor-based systems were Smalltalk and MIT's language, PLASMA (Pugh, 1984).

Carl Hewitt (1977) of the MIT Artificial Intelligence Lab first mentioned the idea of actors in 1976. He wrote a lab memo where he "approached modeling intelligence in terms of a society of communicating knowledge-based problem-solving experts" in order to better understand control structure patterns. In this memo, Hewitt explains how actors can simulate these experts and pass messages to other known actors. He also discusses how the early language, PLASMA, can simulate these actor-filled systems. PLASMA, which was derived from declarative languages like Lisp, was one of the first actor languages. It was followed by several other languages including: Act1, Act2, ABCL/1, Actalk, and Lucy.

In an article published in 1984, John Pugh discusses an early simulation of an air battle known as SWIRL, which was conducted using a language called ROSS. The military used this simulation to recreate air battles. Pugh also discusses how several other languages such as Act 1, Director, Flavors, and LOOPs have emerged through AI studies.

Gul Agha of the MIT Artificial Intelligence Laboratory further elaborates on the importance of actor-based languages in his 1986 overview of actor-based languages. He highlights the concepts of encapsulation and inheritance as important real world experience-

simulating tools. In his discussion of the overall system, Agha defines three important actor-based concepts which include the ability to send messages, to create new actors, and "to specify a replacement which will accept the next communication" (p. 61). The semantics of actor languages—especially Act 1—and the development of higher-level actor languages are also section topics. A few years later, Agha further expounded on these ideas with another paper that dealt primarily with concurrent computing (1988).

Prior to the 1988 article, Agha (1986) wrote *Actors: A Model of Concurrent Computation in Distributed Systems*, where he discusses how actor languages can be used to concurrently model systems. He also discusses two actor languages: SAL and Act. Although the book functions as a guide to developing concurrent programs using actor languages, it does not provide many case studies or real-world applications of the languages.

2.2.1.1 Characteristics. An actor resembles a human being in that "each actor in a system can be thought of as playing out an active (acting) role not unlike the roles humans play in real-life systems" (Pugh, 1984). Actors can communicate with other actors by exchanging messages. They can respond to these messages by changing their behavior or by physically passing a message back to the sender. Each actor has a queue where messages are stored until they are retrieved. There is no central knowledge database, but rather each actor contains its own knowledge database that can be programmed to share information with other actors. Most importantly, because these actors have their own thread of control and work parallel to each other, each actor can run independently. As Frolund articulates, the actor model is a "distributed application that consists of a collection of asynchronous objects that execute concurrently." (1996, p. 3).

2.2.2 Beginning of Easel

During the 1980's, the Software Engineering Institute (SEI) needed to simulate unbounded and emergent systems to assert the survivability of network infrastructure given the failure of a node or system. From 1985 to 1990, the idea of a property-based language first emerged (Fisher, 2000). In a lecture about Informalism, David Fisher (1991) of Carnegie Mellon University explained the need for a new simulation method. Fisher argued that formal methods to that point had failed due to several reasons. He asserted that because systems were often unbounded and physical objects were not finite, they could not be "modeled completely." He concluded that "formal methods are inadequate for describing and reasoning about the physical world." Fisher envisioned a language that would be able to eliminate these modeling problems and more accurately represent the real world.

At that time, no simulation language was able to model changing relationships between elements within a simulation. These languages also failed to allow elements to alter relationships after changing an element's physical location. However, with the creation of Emergent Algorithm Simulation Environment and Language (Easel), researchers at the SEI were able, on the one hand, to simulate emergent and survivability systems, and on the other, to simulate a large number of cooperating actors where global visibility and centralized control were absent. (Belani, Das, & Fisher, 2002, p. 721). Thus, this property-based language helps to create a representative model of the real world.

David Fisher discusses the background and motivation behind the creation of Easel (1999). He notes that the interest in modeling systems for survivability reasons stems "from the concerns for infrastructure assurance" (Fisher, 1999). By early 1999, the experimental design and implementation of Easel had already begun, but the [runtime system] was not created until

the summer of the same year (Fisher, 2000). Alpha and Beta releases followed in 2001 (Fisher, 2000) and they are currently in their first production release.

The first real-world application of Easel was undertaken by the Defense Advanced Research Project Agency (DARPA) which simulated an "emergent algorithm for location-independent IP routing within a survivable routing infrastructure" (DeSantis, 2001). A few other applications followed, including two simulations about transportation methods in large cities.

Christie and Fisher, with the help of a few SEI colleagues, released an Easel language guide in 2003 that is available at the CERT website. This document thoroughly explains the syntax of the Easel language and aids the user in creating useful simulations.

2.2.3 Easel and Software Development

In 2000, David Fisher and Alan Christie gave a presentation about Easel and its application to the field of software development. Fisher began by showing that a software development organization is unbounded because complete information about the system is not known and the people within an organization are dynamic as opposed to static. Thus, Easel can more accurately model a software development organization than other languages because it operates with incomplete information and accommodates changing relationships.

Fisher and Christie also released a paper in 2000 on Easel's use in simulating "emergent behavior of complex software-intensive organizations." They emphasize the notion that most decisions in an organization are based on inaccurate information. Thus, the consequences of these decisions are not only largely unpredictable, but also fail to provide insight into their associated ramifications. Furthermore, stressful environments often plague software development organizations whose employees must function under strict deadlines and unclear

requirements. In response to these pressures, Easel can help organizations adapt to their organizational problems and the dynamically changing external environment.

2.2.4 Advantages

As mentioned, Easel can be used to simulate unbounded and emergent survivability systems that involve a large number of interacting actors. Christie and Fisher (2000) provide a number of reasons that illustrate the need for Easel.

Other simulating languages lack the features or functionality needed to simulate certain systems. Several packages, such as Swarm or MAML, have been created to work with languages such as C++ and Java, but there remains a shortage of simulation packages representing actor behaviors. The only simulation tool that can currently compete with Easel is StarLogo. However, unlike Easel, StarLogo is unable to simulate neighbor relationships or model complex data structures. System dynamics simulation tools offer these capabilities, but fix element locations prior to simulation. Conversely, Easel allows the location to vary throughout the simulation. Fisher and Christie believe that Easel's ability to move elements during simulation is crucial to the simulation of informal processes, such as social interactions.

In addition to changing neighbor relationships, Easel has several other advantages. Easel provides a threaded environment where all modeled entities are considered types. Unlike StarLogo which uses a grid-based graphics system, Easel allows interaction during simulation via its dynamic graphic display. Actors within the simulation may also communicate with other actors and are provided with limited visibility. Finally, Easel allows for the manipulation of entities in a way that is not possible with the use of traditional program objects.

In short, Easel was created at the SEI in order to fill the need for a simulation tool that would be able to simulate emergent, unbounded, and complex systems, where neighbor relationships and a dynamic interactive display would allow the user to observe processes.

2.2.5 Disadvantages

Because work with Easel as a simulation tool remains limited, only the SEI at the CERT Coordination Center Web site has documented its uses and results. Easel's relative newness also brings other drawbacks. Unlike for system dynamics, there exist no established methods for developing Easel models. Because most of the information available on Easel discusses its creation, structure, and syntax, developers working to create actual Easel models can only refer to the limited number of sample programs available on the Cert Web site. However, Easel has been incorporated into the curriculum at a few schools, thereby enhancing its growth potential.

2.3 Brooks's Law Literature Review

There have been several system dynamics applications of Brooks's law. These applications are discussed below.

In 1983, Abdel-Hamid and Madnick published an article on the dynamics of software project scheduling with regard to the inability and difficulty of many software development project managers to establish an accurate schedule estimate. They recognized that software development was a complex system where feedback was present. In this article, they discuss the model that they developed in order to simulate such systems.

They developed a model of a project with the size of about 1000 tasks, and modeled the project from the design phase up to and including the integration and test phase. Using the COCOMO model, an average staff size of 34 people is established, along with a schedule of 38

months. Rework is incorporated into the project and it is estimated that after month 13, management will look to hire additional staff in order to meet the scheduled completion date. With the addition of rework, the project is not completed until month 41.5 and is almost 2 million dollars over the estimated cost of $7,810,600.

Personnel turnover was also taken into account, and average employee commitment time was estimated at 24 months. With these details considered, the project finished at 51 months, and at a cost of $9,582,400.

Lastly, estimation error was introduced. When the project first began, the number of tasks was estimated by management to be 800. The actual number of tasks was 1000, which did not result in further schedule delay, but did increase the project cost by over $200,000.

Abdel-Hamid and Madnick stress that the dynamic consequences of feedback loops for complex software development projects are not intuitively obvious; hence simulations of such complex feedback systems are needed (p. 334). The model also helps to prove that adding people to a belated software project just lengthens the schedule time, and looks for other sources of schedule-estimation problems.

Abdel-Hamid and Madnick again approach the solution for this and other problems through the use of system dynamics in an article on the lessons they have learned from modeling software development dynamics (1989). They developed a comprehensive model that provided insight into the problems with project scheduling and other issues such as the 90% completion syndrome and quality assurance effort.

When dealing with adding more people to a late project, Abdel-Hamid and Madnick state that often overhead from communication is ignored when making the decision to higher additional staff. Higher overhead can lead to a lower production rate, which can only make the

project later. Schedule pressure also plays a role in the lateness in a project, for it may increase productivity, but it may also increase the error rate, leading to an increased amount of time spent on rework. Lastly, they comment that since software is intangible, managers must rely on progress estimates given to them by developers. These estimates are not always accurate, making the manager's task of schedule estimating more difficult.

The model used for this simulation consisted of four subsystems and is identical to the model used in their later book (1991). A DE-A case study was conducted at NASA's Goddard Space Flight Center. The project studied was 24,000 delivered source instructions (DSI) and programmed in FORTRAN. The initial schedule estimate was 2200 man-days and schedule slippage was not tolerated. The study showed that due to schedule pressure, developers worked at a higher rate toward the final stages of the project, yet the net progress rate slowed at the end of the project. The results of the studies were discussed in the 1991 article, which was discussed earlier in this chapter.

Project scheduling practices of U.S. minicomputer manufacturers where managers were rewarded for accurate project schedules were also discussed in this article. A project whose size was 64,000 DSI was initially estimated incorrectly at 43.88 KDSI. The project manager made the schedule estimate using a safety factor and COCOMO estimates which led to a smaller schedule estimation error.

Their research also showed that one of the causes of lower productivity at the end of a project is the communication overhead from hiring toward the end of a project. Overall their research showed how modeling can provide the insight needed to make accurate estimations.

In *Software Project Dynamics: An Integrated Approach*, Abdel-Hamid and Madnick devote a chapter to applying their system model in order to investigate Brooks's law. They

modify their original model to include aspects such as lowered average productivity for new employees and new employee assimilation rates. They also include a hiring delay of 40 days and an assimilation delay of 80 days and add variables that represent the management's willingness to hire new developers depending on the estimated project time remaining. Abdel-Hamid and Madnick repeat their claim that adding more people to a late project does not necessarily result in a late project, for the cumulative effect of new hires does always create a substantial drop in productivity (p. 218).

A recent study (Hsia, Hsu, & Kung, 1999) revisited Brooks's Law in order to determine at what point in time more manpower may be added to a late project without the project finishing late. In their experimentation using a system dynamics model, three factors of Brooks's Law were taken into consideration: time loss due to training of new staff, time loss due to teaching new staff, and time loss due to communication overhead. The authors believed that to apply Brooks's Law to a realistic situation, new people could be added only a few times during the project and that a sequential constraint be present during software development.

The model the authors developed permitted hiring once during the project, and added a sequential constraint. They then validated their model against the data from the Abdel-Hamid and Madnick model (1991). The results from the simulation showed that the sequential constraint plays a significant role in project development, and that the optimal time for adding people to a project "ranges from one-third to halfway into the project development." They concluded that while it is always costly to add people to a late project, there is an optimal time range to add people within which schedule delays can be eliminated or minimized.

Another study (Caulfield & Maj, 2002) looks at Brooks's Law and the importance of soft variables when simulating software development projects. The project simulated was estimated

to require 36 man-months, and had a schedule of six months. The project starts with five developers, although it is known at the beginning of the project that six developers are needed, and two of the five developers are new staff requiring about two months of training each. Overhead is represented as "one hour per developer per week per communications path" (p. 29).

The authors discuss that soft variables, such as stress and the stakeholders' perception of quality, must also be taken into consideration in order to get a more reliable schedule estimate and that the use of system dynamics makes this possible (p. 30). The results from the simulations proved that Brooks's Law is not always valid. As Abdel-Hamid and Madnick saw, adding developers early on in a project will not always result in a late project.

These applications have shown how Brooks's Law may be applied to the development of a software project through the use of a system dynamics model. Brooks's Law has also shown to be true in many of the trials, yet trials by Abdel-Hamid, Madnick, and a few others have shown that Brooks's Law does not always hold true. If additional manpower is added early enough, a late schedule may be avoided. Through the use of system dynamics modeling, better schedule estimates can be achieved by modeling complex feedback systems with soft variables.

CHAPTER 3

METHODOLOGY

Brooks's Law, popularized in The Mythical Man-Month, is a classic observation and speculation in the software engineering field. Brooks's Law will be modeled using both Systems Dynamics and Easel in order to compare and contrast the two simulation methods.

3.1 Introduction

The problem chosen for modeling is Brooks's law. The major model factors will be: communication overhead, training of new employees, and assimilation of new employees.

3.1.1 Brooks's Law

In Fred Brooks's collection of essays (1995) depicting his experiences as manager of IBM's OS/360 project, he discusses the analytical fruit that arose from his frustration with management and scheduling which has long since been referred to as Brooks's law. Brooks's Law states that "adding manpower to a late software project makes it later" (p. 25). The following is the foundation of this law: "Cost does indeed vary as the product of the number of men and the number of months. Progress does not. Hence the man-month as a unit for measuring the size of a job is a dangerous and deceptive myth. It implies that men and months are interchangeable" (p. 16). This law is often misunderstood. Brooks's implies that adding more people to a late project will only make the state of the project worse due to the fact that men and months are not interchangeable.

Brooks provides several reasons why men and months are not interchangeable. The first of these is the idea of sequential constraints. Some tasks may only be accomplished by one individual. Adding additional manpower can be useless in these situations. Secondly, tasks that

can be broken down will result in communication overhead between developers. Communication in this respect can be both training of new staff and intercommunication. In training, the new staff member must be made aware of the technology utilized, the project strategy, and goals (p. 18). Staff previously working on development must allocate time to train the new staff which adds to communication overhead. Although training is timely, Brooks states that intercommunication is more costly (p. 18). Coordination effort between developers working on a divided task increases $n(n-1)/2$. Partitioning of tasks results in an increased amount of development time spent solely on communication.

Brooks's Law warns managers about adding staff to a late project as a quick fix and his assumptions are generally supported in the works of others. Brooks's Law has long since been studied and applications of Brooks's Law involving system dynamics are discussed in chapter 2.

3.2 Model Descriptions

Before creating simulations using both methodologies, the system must first be translated into an appropriate model. The following section describes the models that will be the basis for the simulations.

3.2.1 The System Dynamics Model

The initial model chosen for the system dynamics simulation was created by Madachy (2003) and is shown in Figure 11.

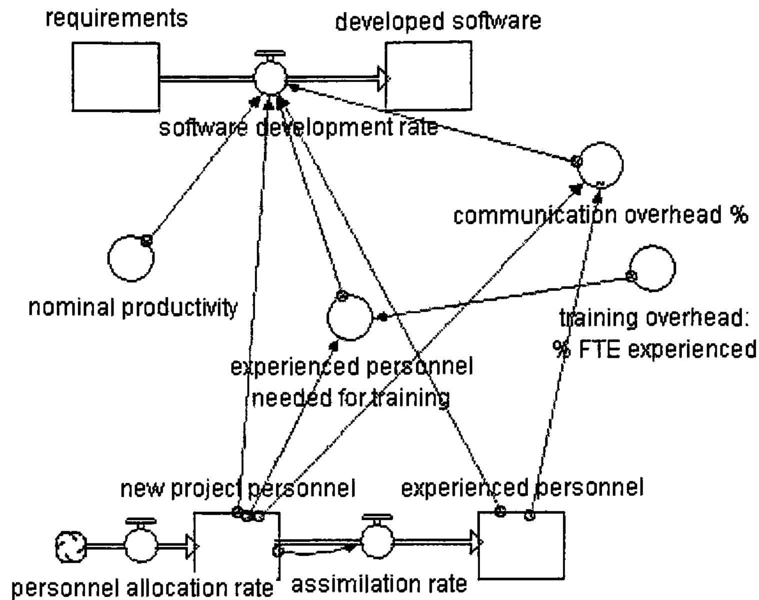


Figure 11 – System Dynamics Brooks's Law Model: Flow Diagram

This model depicts the development of software through function points. The development is not broken into phases; the scope primarily is on the development of the function points. Therefore, testing and quality assurance, along with any manager and client entities, are not represented. This is a simple small-scale model that will be enhanced for experimentation.

This flow diagram consists of several levels. The requirements level holds the number of function points required for the software project, while the developed software level includes the number of completed function points. The new project personnel level and the experienced personnel levels hold the number of new personnel and experienced personnel respectively. There are two valves in the flow diagram. One valve is responsible for the software development rate, while the other is responsible for the assimilation of new personnel which controls the flow from new project personnel to experienced project personnel. Other influences to the software development rate are shown as converters, or circles. There is a nominal rate which is influenced by three other factors: experienced personnel needed for training,

communication overhead, and training overhead. The three converters determine the software development rate to control the flow from the requirement level to the developed software level.

The small-scale model is based on a 500-function-point project, where 20 experienced employees are available at the start of the project. At the 100th day, new personnel may be added. This project is estimated to span 274 days with a constant staff of 20 developers. If five developers are added on the 100th day, the project is estimated to span 271 days. If ten developers are added on the 100th day, the project is estimated to span 296 days. The day and frequency at which new developers are added can be adjusted in addition to the number of the requirements and the number of new employees being hired.

The model will also include a factor not previously incorporated by the Madachy (2003) model. This addition will provide the capability to simulate the addition of new requirements after a project has begun. The model for Brooks's Law is shown again below with an additional valve that adds requirements throughout development in order to represent requirements creep.

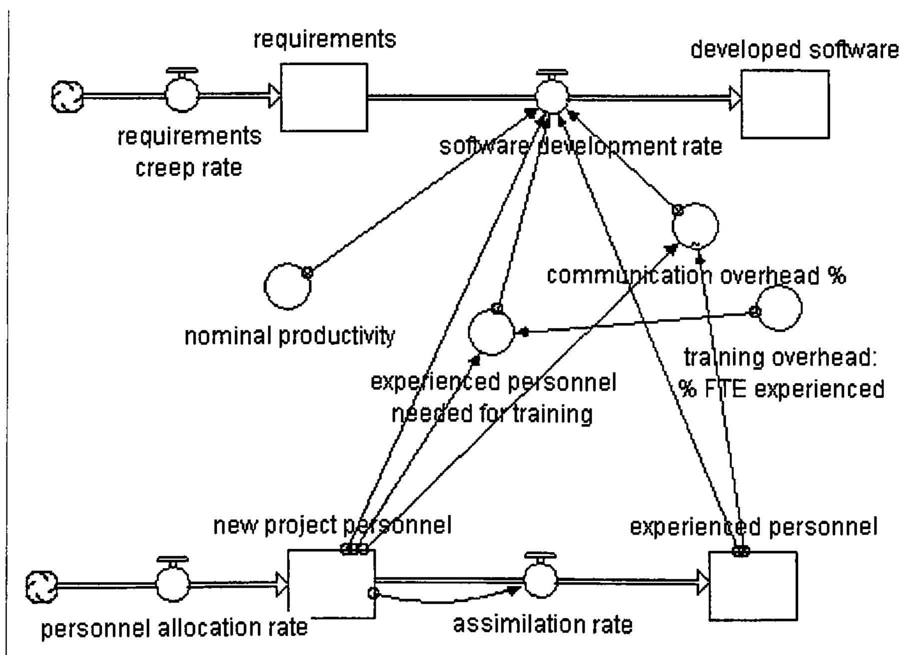


Figure 12 – Brooks's Law with Requirements Creep

Three project scenarios will be examined during experimentation. For both the small and medium project, the above model will be used. To better represent a large system, where developers are broken into teams and the amount of overall communication overhead is reduced, the model will need to be further refined. The model will also include a team-size level to represent the number of developers per team.

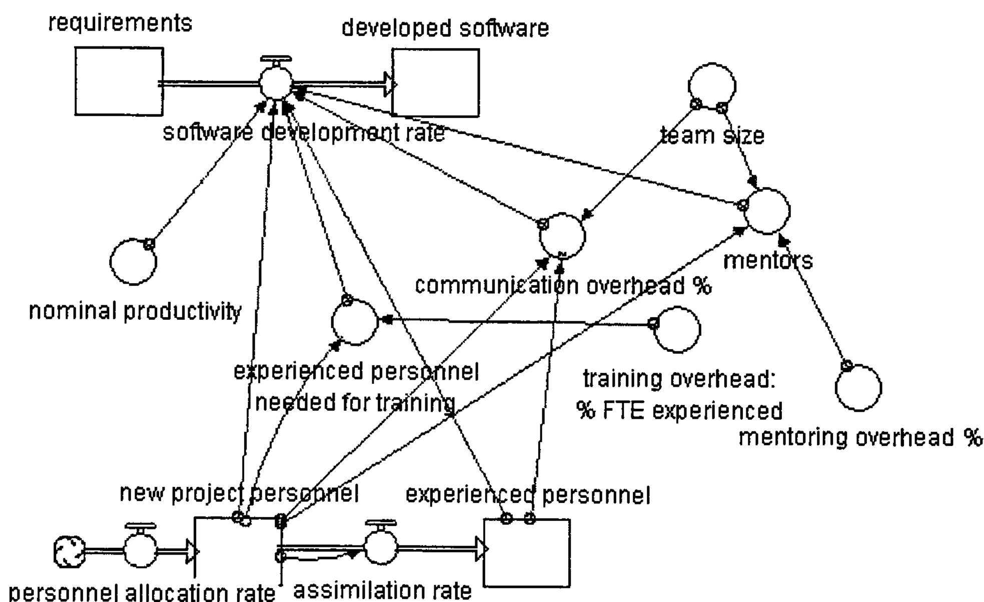


Figure 13 – Brooks's Law Model for Developers Broken into Teams

Since grouping the developers into teams will reduce the amount of communication overhead, the team size will determine this overhead.

Another factor introduced into the large project model is mentoring overhead percentage. This overhead is needed to account for new team creation and training of the new team leader. Experienced team leaders will reduce their productivity by spending a portion of their daily activities training a new team leader for four weeks. The training overhead involved is

determined by establishing the number of mentors needed, and then factoring this figure into the software development rate.

3.2.2 The Easel Model

The Easel model is based on a software development simulation created by Christie and it requires several enhancements to model Brooks's law. This model consists of several actors, including a manager and several developers. The model also consists of several types, including projects and modules. Since types and actors are comparable to an object in object-oriented languages, a class diagram of the model is shown in Figure 14.

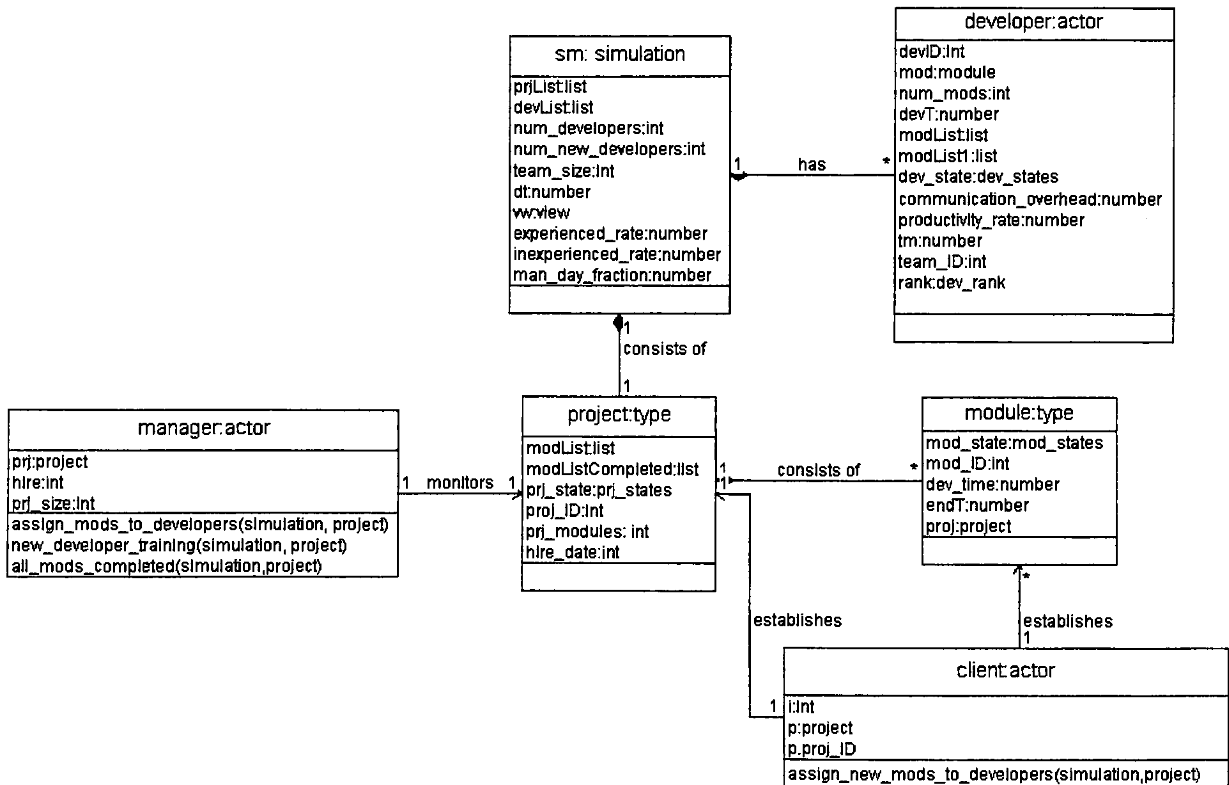


Figure 14 – Easel Brooks's Law Model: Class Diagram (with Enhancements)

The simulation type holds global variables that are easily accessible to other actors and procedures. The manager actor is given a project which is divided into several modules. The modules are established by a facilitator called the client. The manager may then allocate the

modules to the developers. The manager observes the developers in order to determine the completion of the project and assigns new developers to a project when needed. A developer actor can be assigned more than one module. The developers must develop each module assigned to them. Each developer has a state: occupied or free. If the developer's state is set to free, the developer may begin working on an assigned module. The client actor is responsible for creating the modules.

The project consists of several modules, and each module has a development time and state: unassigned, assigned, or completed. The module lengths may be set to variable or constant. The simulation will continue until all modules are completed and ultimately the project is in a completed state.

In addition to the elements established by Christie, other features will be developed in order to observe Brooks's law. Currently, the model lacks the functionality needed to add new developers after a project has begun. Allocated training time and reallocated project model functionality will also be needed. As in the system dynamics model, this model will also allow new requirements to be established after the project development has begun. As with the system dynamics model, requirements creep will also be implemented along with the code required to break developers into small teams. Mentoring of new team leaders by experienced team leaders will also be a new functionality of this model.

3.3 Simulation Design

With models for both methodologies established, both simulations can now be created. The next section discusses several experiment assumptions and how enhancements to existing models are applied.

3.3.1 System Dynamics Simulation

The model and equations required for the simulation have been created for the iThink simulation development environment. This model makes the following assumptions:

- New personnel are hired only once during development.
- The assimilation time for new employees is on average 80 days.
- A quarter of the experienced personnel will be needed to train the new personnel.
- The percentage communication overhead is calculated by squaring the total number of employees and multiplying that number by 0.06. This is the calculation previously used by Abdel-Hamid (1991) to compute communication overhead.
- Nominal productivity is set at 1, while 0.8 and 1.2 are used for experienced personnel and new personnel productivity respectively.
- A creeping requirements rate of 2% a month (Jones, 1998) is set.
- For the large project: The percentage of communication overhead is calculated by squaring the number of developers in a team and then multiplying that number by 0.06.
- For the large project: Mentoring overhead is calculated by taking 20 percent of an experienced team leader's time for each new team leader. Each experienced team leader will spend four weeks with the new team leader, while gradually reducing the percent of time mentoring to 0 percent.

The simulation was initially created by using a series of differential equations. In order to simulate the requirements creep rate, a new equation had to be added to the simulation. The requirements creep rate was implemented using the PULSE function which indicates that the number of requirements is to first increase by two percent of the initial requirements at the 30.5 day mark, and is to be increased again by the same amount at a regular interval of 30.5 days.

requirements_creep_rate = PULSE(*Intitial_Requirements**0.02,30.5,30.5)

Equation 2 – Requirements Creep

This then had to be added to the number of requirements initially stated in the requirement level in order to indicate the number of requirements at any given time.

requirements(t) = requirements(t - dt) + (requirements_creep_rate - software_development_rate) * dt

Equation 3 – Requirements with Creep

The new equation for the inflow and the modified equation for the level allow the simulation to model requirements creep.

To add the team size feature, the amount for the team size had to be initially declared and the communication overhead percentage had to be altered to include this figure. The team size was again needed when determining the amount of time spent mentoring. The mentoring percentage was first set to 20 percent. The number of mentors needed could then be calculated using the following equation.

mentoring_overhead_% = 20

mentors = ((new_project_personnel/team_size)*mentoring_overhead_%/100)

Equation 4 – Mentoring Overhead

The last addition was then to subtract the time dedicated to mentoring from the software development rate.

software_development_rate = nominal_productivity*(1-communication_overhead_%/100.) *(.8*new_project_personnel+1.2*(experienced_personnel-experienced_personnel_needed_for_training - mentors))

Equation 5 – Software Development Rate Minus Mentoring Overhead

The output can easily show the change of any equation variables over a period of time. Two variables of interest in this simulation will be the software development rate and the amount of developed software over a period of time. The equations for the entire simulation are found in Appendix A.

3.2.2 Easel Simulation

The model and simulation were originally designed by Christie to simulate the software development process. This simulation was modified in order to observe Brooks's law. Additional procedures were added to provide the addition of new employees during software development, including a procedure for training and module reassignment.

The same assumptions that were made for the system dynamics simulation apply to the Easel simulation. In order to create this simulation, new functionality was added. It is assumed that new developers are only added once throughout development. In order to simulate this, the manager actor needed to observe the current time and then perform a new function based on that

time. The code below expresses that the manager checks to see if the project is still active and that if it is active at a certain period of time, new developers are to be hired.

```
if (prj_size!=0) then
        # Check simulation clock time and other variables
        if (s.skdr.clock > prj1.hireDate & hire = 0 & s.num_new_developers!=
        0) then
                hire := 1;    # since hiring only occurs once - set flag
                new_developer_training(s, prj1);
```

Figure 15 – Manager Code for Hiring New Developers

In order to add new developers, the following procedure creates new actors and assigns each new developer IDs and minimal productivity rates. This procedure is called the new developer training method.

```
addDevelopers(s:sm, p:project, numOfnewbies:int): action is
# Add specified number of developers to the project
        new_id:int :=length(s.devList);
        overhead_factor:int := 0.06;
        mod_list_len:int := 0;
        mod:module:=?;
        newDevList:: list := new list developer;
        s.num_developers := s.num_developers + s.num_new_developers;

        # Initialize each new developer
        for j:each 1..numOfnewbies do
                d:developer := new (s, developer(s));
                new_id  := new_id + 1;
                d.devID := new_id;
                d.devT := p.hireDate;       # time to bring in recruits
                d.productivity_rate := 0.8; # low productivity rate
                d.communication_overhead := (overhead_factor *
(s.num_developers) * (s.num_developers));
                push(s.devList, d);
                push(newDevList, d);
```

Figure 16 – Procedure for Adding New Developers

To reassign the modules, all modules not completed were collected from the previous developers in a reassignment procedure. The modules that were not completed were then randomly assigned using a method previously established by Christie.

```
reassignModules(s:sm, p:project): action is
# New developers or modules have been added, need reassignment of tasks
        newModList:: list := new list module;

        # Collect all the unfinished modules from developers
        for d: every s.devList do
                for m: every d.modList do
                        push(newModList, m);
                d.num_mods :=0;
                emptyModList:: list := new list module;
                d.modList := emptyModList;

        # Set unfinished modules as new project list
        p.modList := newModList;

        # Call procedure responsible for assigning modules equally
        assign_mods_to_developers(s,p);
```

Figure 17 – Procedure for Reassignment of Modules

The last method was responsible for gathering experienced developers and allocating their time to the training of new developers. The number of experienced developers gathered was a quarter of the number of experienced developers. The developers were randomly chosen and their rate was lowered to 0.8 during the twenty days designated to training. During the eighty days, the new developers' productivity rate would slowly increase. At the end of the training period, the experienced developers would again have a productivity rate of 1.2.

```
assimilation(s:sm, trainerList:list): action is
# This procedure is responsible for all training overhead
        last_rate::number :=0.0;
        assim::boolean:= true;
        percent::number :=0.0;
        min ::number := 0.00001;

        # Find amount of time needed for training
        training_prod ::number := (s.num_new_developers/(s.num_developers-
s.num_new_developers)*0.25*1.2);

        # Begin the assimilation process
        for every assim do
               rate := rate-(rate/20);
               percent := (1-rate/20)-(last_rate);
               outln("% for this round: ", rate);
               for d: every s.devList do

                        # Locate experienced developers
                        if (d.devID > (s.num_developers -
                  s.num_new_developers) & d.devID < s.num_developers+1) then
```

```
                                        d.productivity_rate:=d.productivity_rate +
(0.4*percent);

               last_rate := last_rate + percent;
               if rate < min then
                        assim:= false;

               # Reduce productivity by certain percentage
               for t: every trainerList do
                        t.productivity_rate:=t.productivity_rate +
(training_prod*percent);
               wait 1.0;
```

Figure 18 – Assimilation Procedure Code

A method was also created in order to simulate the addition of new requirements during development. This method adjusted the number of requirements every 20 days and then allocated the new requirements to the developers with the shortest module lists. This was done through the use of a client actor.

```
client(s:sm): actor type is
# Initialize the client and create project with modules
        i:int := 0;
        p:project := new project;
        p.projID := i;
        push(s.projList, p);
        for j:each 1..p.prj_modules do
                m::module := new module;
                m.modID := j;
                m.proj := p;
                push(p.modList, m);

        # Add new requirements every 30.5 days
        for every true do
                 if (p.prj_state != closed) then
                        if((mod(s.skdr.clock, 30.5)) < 1) then
                                assign_newmods_to_developers(s, p);
        wait 1.0;
```

Figure 19 – Client Code

For the large group, team size and mentoring were added to the already existing procedures. Team size was added to the simulation variables, while the mentoring was an addition to the training and assimilation procedures. In order to achieve the team effect, all developers were assigned a team number and each team was assigned a leader. When gathering

the mentors needed, the list of developers had to be searched until a team leader was found. That team leader's productivity rate was then reduced to reflect the time spent mentoring. The productivity was then increased gradually over a period of 4 weeks. To achieve this effect, new features were added in both the procedure for adding new developers and the assimilation code.

The original simulation output was removed, and a new graphical output substituted for it. The graph shows the team development rate (productivity) over time. The code provided for this section and the entire simulation is found in Appendix B.

3.4 Experimental Design

Both simulations have common input parameters that can be altered. The simulations will be run several times with various inputs. The values chosen are based on project size estimates given by Capers Jones in *Estimating Software Costs*. Three projects, each of different magnitudes, will be simulated. The input parameters for each experiment trial are provided in Table 2.

Project Size	Parameter	Value(s)
Small project	**Number of experienced employees**	5
	Number of function points	500
	Number of new employees hired	0,2,5,7,10
	Team size	Number of total employees
Medium project	**Number of experienced employees**	10
	Number of function points	1000
	Number of new employees hired	0,5,10,15,20
	Team size	Number of total employees
Large project	**Number of experienced employees**	100
	Number of function points	10000
	Number of new employees hired	0,25,50,100,150,200
	Team size	5,10,15,20

Table 2 – Simulation Initial Parameters

Notice that the project developers will not be broken up into small teams for the first two simulations, and that only in the large project will smaller teams be formed. The requirements creep will only be implemented in the small and medium projects. These two projects will be run first without the requirements creep and then with the requirements creep.

The simulations for each project size will first be executed with no hiring involved. These results will serve as the default completion date for each project size. For the small and larger project, hiring of new developers will take place at 3 different intervals. The intervals will be determined by taking 25, 50, and 75 percent of the completion time. For the large project, hiring will only take place at the date which is 25 percent of the default completion date.

Both simulations will be run with the same input parameter and the results gathered from the output will be compared. Through the graphical output, information about the following

output variables will be recorded: software development rate and project completion time. This information will aid in comparing each simulation method and its effectiveness in representing the software development process.

CHAPTER 4

DATA ANAYLSIS

In this chapter, analysis of the data gathered from the system dynamics and Easel simulations of Brooks's Law is discussed. The validity of Brooks's law was tested across a range of development project sizes: a small project (500 function points and five experienced developers), a medium-sized project (1,000 function points and ten experienced developers), and a large-scale project (10,000 function points and 100 developers).

4.1 Definitions

In order to facilitate the reader's understanding of terminology used in this chapter, the following definitions are provided. Additional information about specific rates and times can be found in Chapter 3.

Development rate: Function points completed per day by a developer

Team development rate: Function points completed per day by a team

Developer: An experienced developer

New developer: An inexperienced developer

Default completion time: Number of days required to complete a project without new hiring

Default team development rate: Function points completed per day by a team without hiring

Hire time: Day when new hiring occurs

4.2 Small Project Analysis

The experiment was conducted with the following initial parameters that represent a project half the size of an average commercial software development effort:

- 500 function points (500 FP)
- 5 experienced developers

The development rate for experienced developers is initially set to 0.12 FP/day. Communication overhead for a group of five developers is a constant 1.5 percent of developer time. The net effective development rate for an individual developer is (0.12 FP/day) * (1-0.015) = 0.1182 FP/developer day. Both simulation models were first executed with the communication overhead as the only negative factor impacting the development rate. Since no new developers were hired, there was no effort required for training and no increase in communication overhead. When measuring the default behavior of both simulations, the system dynamics simulation finished at 846 days (approximately 3 years), while the Easel simulation completed at 847 days. The resulting estimate of 3 years is consistent with Jones's minimum estimate of nine months development time for a project of this size (Jones, 106). Table 3 shows the results of each experiment trial.

Number of New Hires	Hire Time	SD Result	[illegible]
2	90	645	649
2	212	675	678
2	423	735	742
2	635	793	800
5	90	495	499
5	212	550	554
5	423	652	653
5	635	753	754
7	90	440	449
7	212	507	510
7	423	622	624
7	635	737	740
10	90	385	395
10	212	462	496
10	423	595	601
10	635	722	728

Table 3 – Small Project Simulation Results

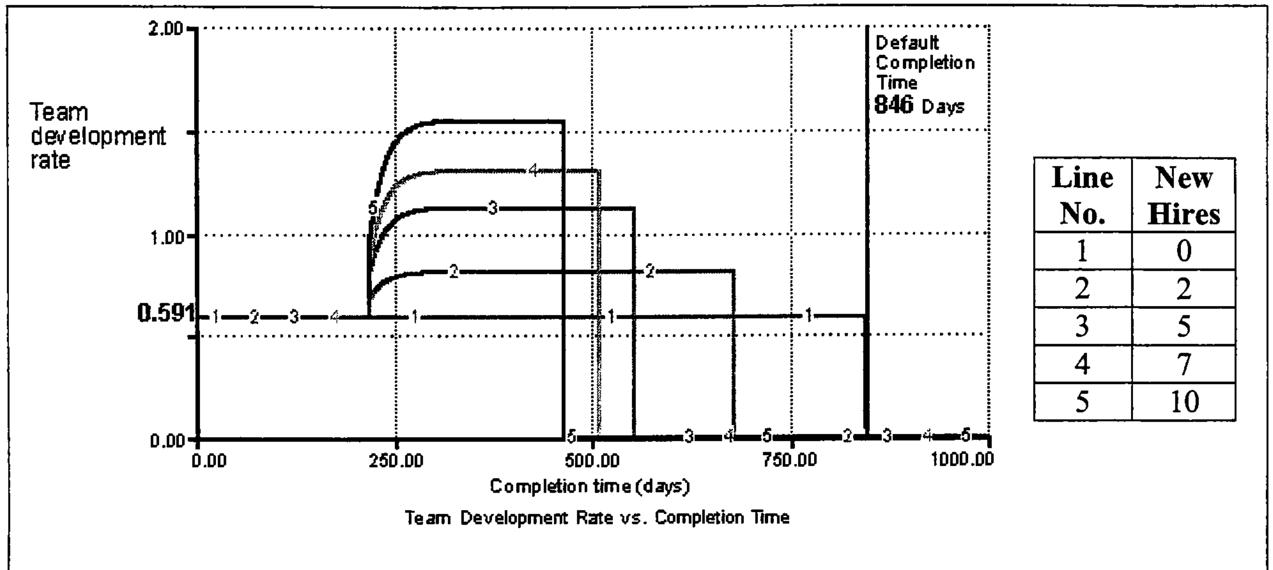
Completion times from both simulations disprove Brooks's law. Hiring an increasing number of developers at any of the four hiring times resulted in a completion time shorter than the default completion time. By doubling the number of developers early during development time (at day 90), the previous development time of 3 years is reduced to approximately 1.5 years. This shows that if additions to development staff are planned, it is more beneficial to add developers earlier in the project than later in the project. In the case where the number of developers was doubled late in the project (at day 635), the development time is reduced to approximately 2.5 years.

The impacts of hiring of new developers at different hiring are discussed in section 4.2.1.

4.2.1 Hiring New Developers Midschedule

In the first trial, new developers are added early during development. With a default completion time of 846 days, the 212th day marks the first quarter of the project. Figure 20

shows the software development rate at five different trials, each with a different number of developers hired at each trial.



Line No.	New Hires
1	0
2	2
3	5
4	7
5	10

Figure 20 – System Dynamics – Hiring at Day 212

The result of each trial was a project completion time shorter than the default completion time. The communication overhead caused by adding more staff reduces the software development rate. This overhead caused the completion times of the last three trials to differ by an average of 40.5 days.

Following Madachy (2003), the team development rate for a team of five experienced developers is calculated as Team Development Rate = Developer Rate * Number of Experienced Developers. In the present case, Team Development Rate = 0.1182*5 = 0.591. For the default trial, this resulted in a team development rate of 0.591 FP/day, as noted in Figure 20.

The communication overhead was not large enough to have a negative impact on the team development rate. The rate did not fall below the initial rate of 0.591 FP/day regardless of the number of new developers hired. Figure 21 depicts the communication overhead and

training overhead as a function of the number of new developers hired. Small project training overhead will have a greater impact on development rate than communication overhead in this case. The larger value of the training overhead had only a modest effect because the training lasted an average of 80 days, which is less than ten percent of the default completion date.

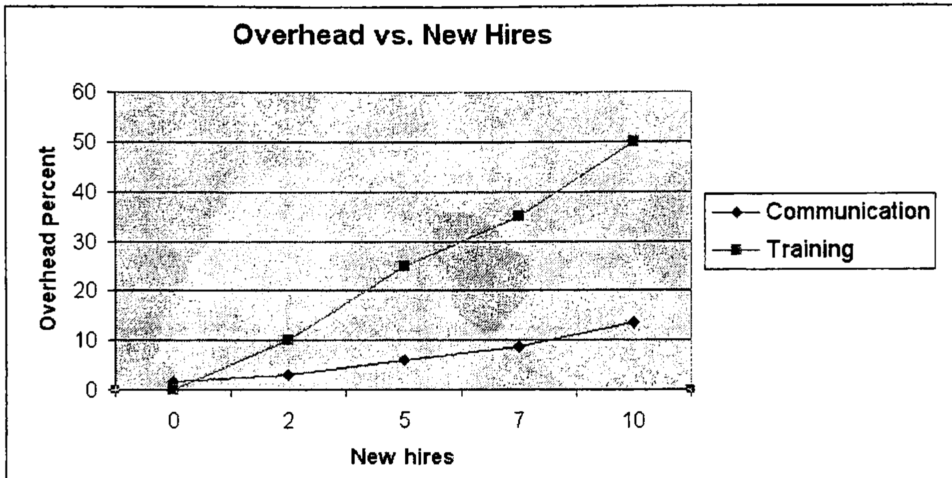
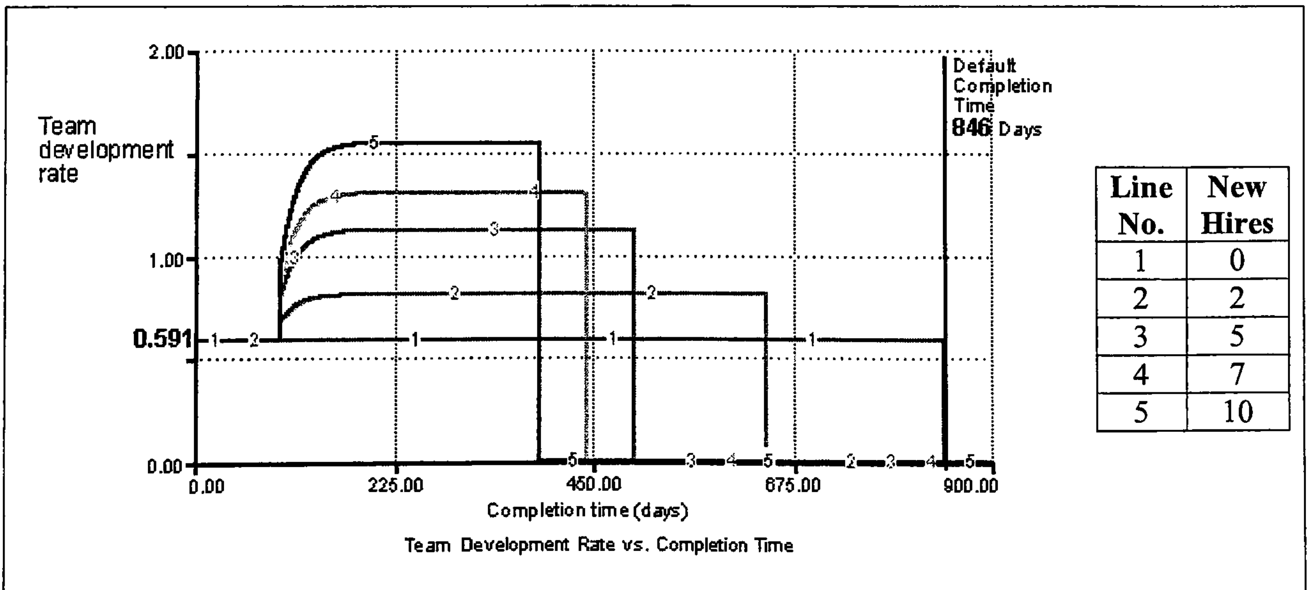


Figure 21 – Overhead for Both Simulations

To demonstrate the effect of hiring at a more realistic date, the system dynamics output for hiring three months into the project is shown in Figure 22.



Line No.	New Hires
1	0
2	2
3	5
4	7
5	10

Figure 22 – System Dynamics – Hiring at 3 Months

The output shows the same pattern found when hiring new developers at day 212. By hiring earlier in both simulations, the completion time was shortened from anywhere between 5 and 20 percent of the previous completion time, depending on the number of new hires.

In the Easel simulation, results also support the previously explained results. As shown in Figure 23, the hiring of new developers produces a completion time shorter than the default completion time.

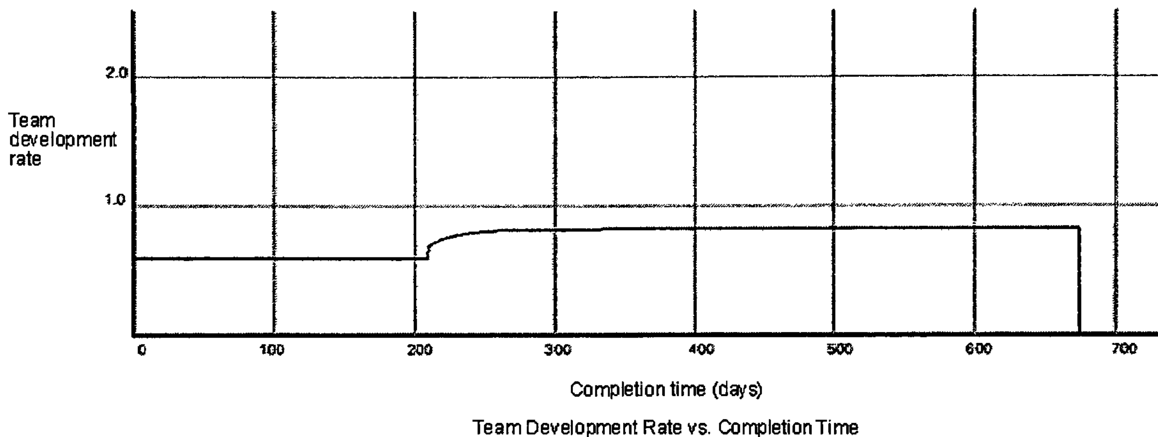


Figure 23 – Easel – Hiring 2 New Developers at Day 212

In Figure 4, the team development rate increases from 0.59 to 0.68 FP/day. The sudden development rate increase is the result of the following: the increase in the rate due to the number of new developers hired, the decrease in rate due to training overhead, and the decrease in rate due to the communication overhead. The increase in team development rate due to the addition of two new project developers is much greater than the two overheads involved, resulting in an increase in the team development rate. For over 100 days, the training period for new hires, the team development rate slowly increases until it reaches a steady rate of 0.86 FP/day.

Hiring five, seven, and ten new developers yields similar results as shown in Figure 24, 25, and 26.

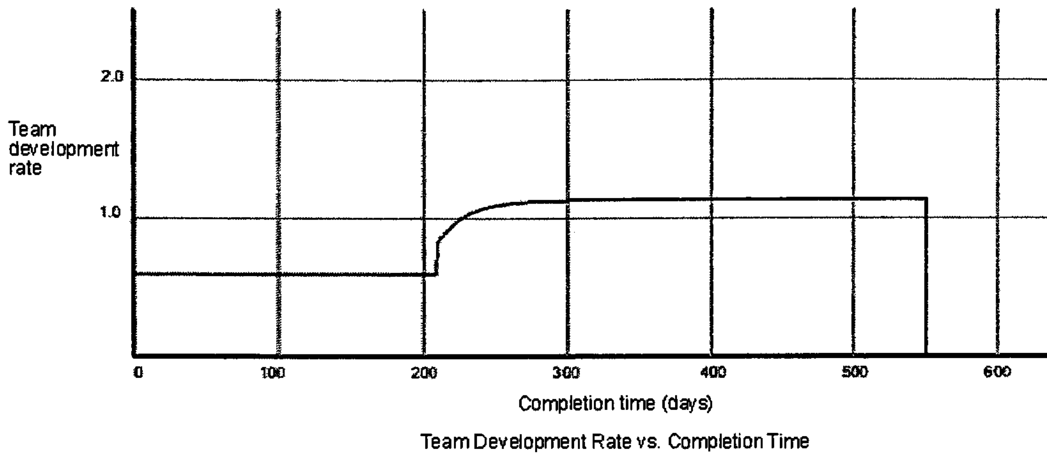


Figure 24 – Easel – Hiring 5 New Developers at Day 212

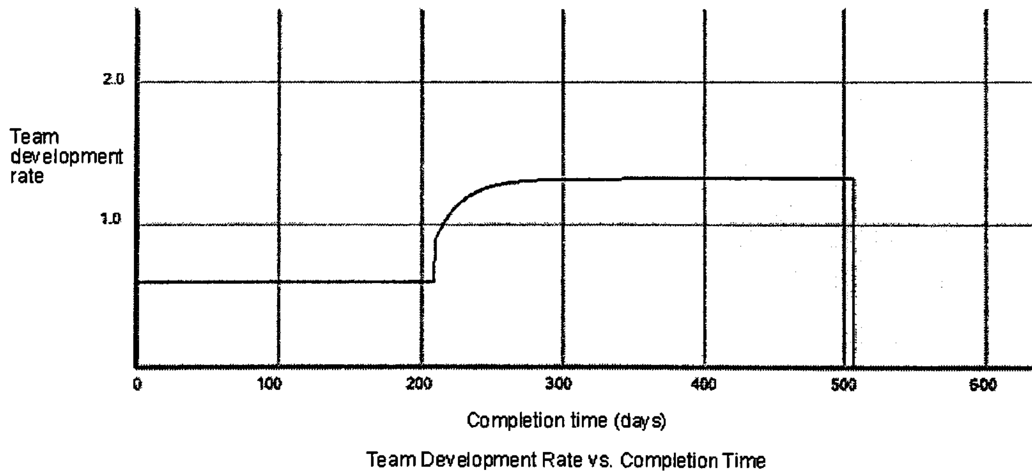


Figure 25 – Easel – Hiring 7 New Developers at Day 212

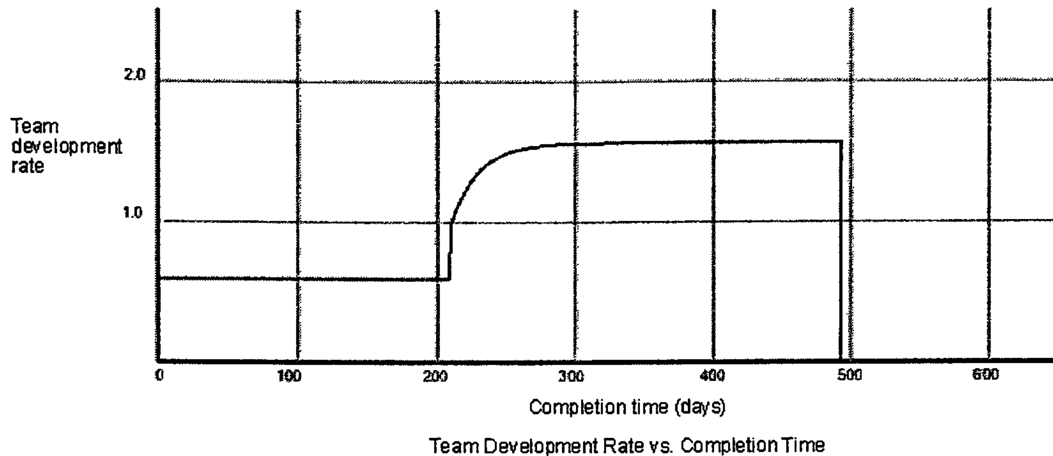
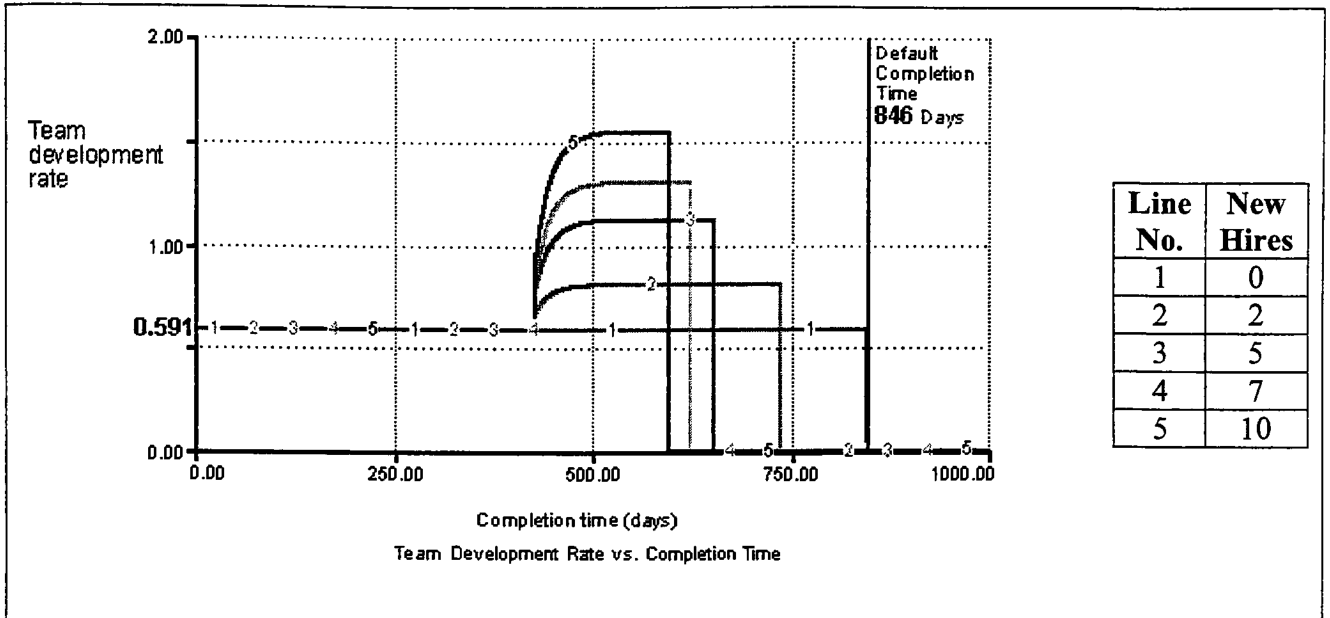


Figure 26 – Easel – Hiring 10 New Developers at Day 212

All three trials above show an initial increase in productivity, meaning that the communication and training overheads were too small to have a negative effect on productivity. For the last three Easel trials, the simulation depicted similar results, yet further along the project schedule. A discussion of additional Easel trials results can be found in Appendix C.

As expected, hiring at a later time during project development does not result in the accelerated completion times that we have observed in earlier hiring schedules. When hiring new developers at a later time, the completion times begin to converge into a narrower range compared to the completion time range observed when new developers are hired at an earlier time. As shown in Figure 27, the completion times for the last three trials are separated by an average of 28.5 days, which is 26.5 days shorter than the period separating the last three trials when hiring new developers at three months.



Line No.	New Hires
1	0
2	2
3	5
4	7
5	10

Figure 27 – System Dynamics – Hiring New Developers at Day 423

Lastly, hiring even later during the development period generates similar results, as shown in Figure 28. However, the difference between each completion time is now narrow and demonstrates how hiring late in the project, regardless of the number of new developers, may not result in schedule reductions. As shown below, there is only a month completion time difference that results from hiring seven developers instead of ten developers.

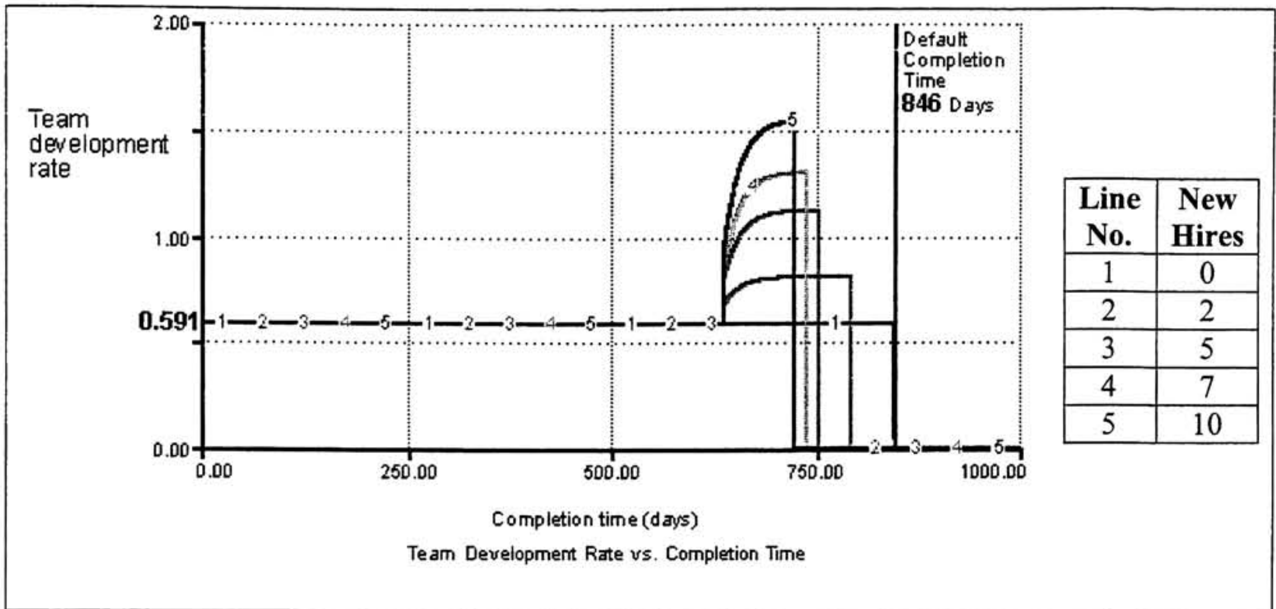


Figure 28 – System Dynamics – Hiring New Developers at Day 635

4.2.2 Small Project with Requirements Creep

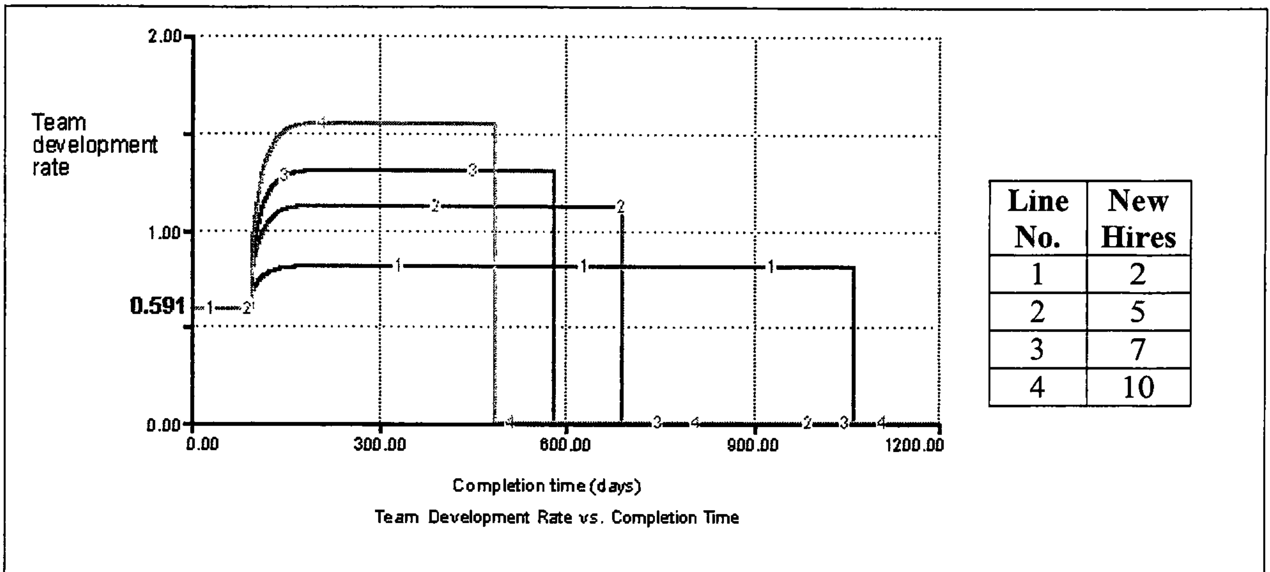
Requirements creep was added to selected simulations. The simulation involving hiring new developers at three months showed an increase in the completion time from the addition of new requirements throughout development. With no hiring of new developers the default completion time for both projects was estimated at 1900 days.

Number of New Hires	SD Result (No Creep)	Easel Result (No Creep)	SD Result (with Creep)	Easel Result (with Creep)
2	645	649	1060	1116
5	495	499	687	720
7	440	449	580	604
10	385	395	483	503

Table 4 – Results from Hiring 3 Months into the Project (with and Without Requirements Creep)

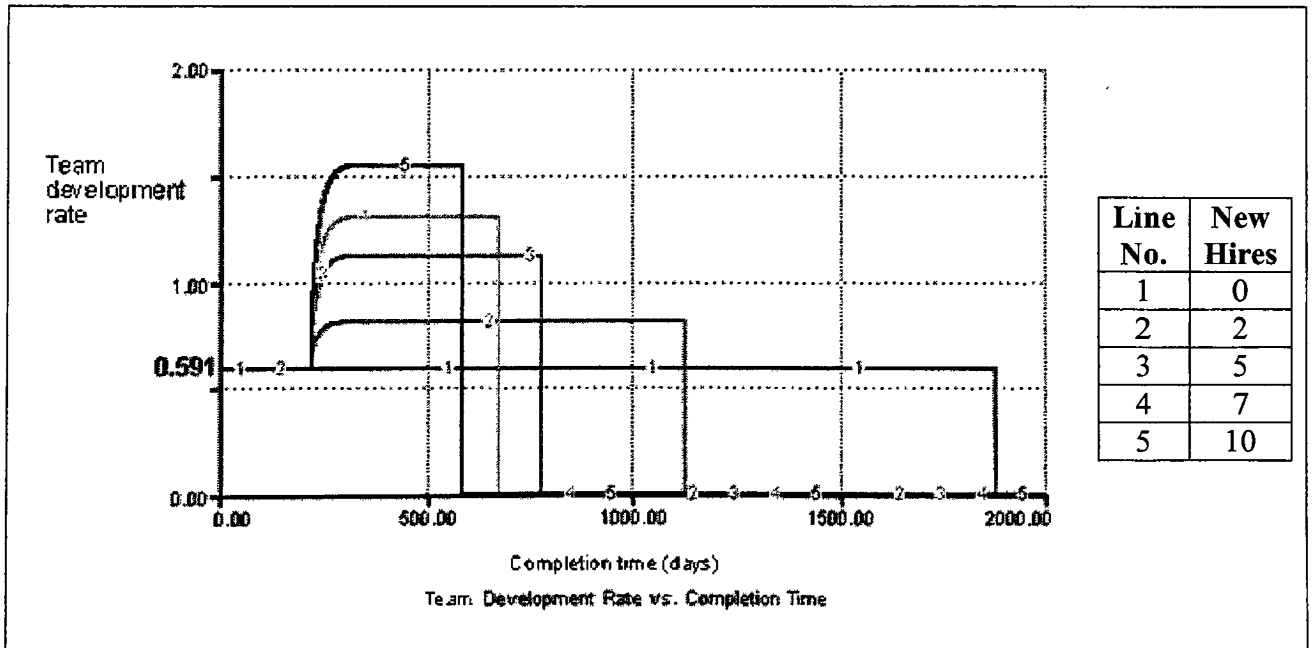
The completion time varied from between an estimated 100 to 400 days later than previous results, showing that a project completion time can be delayed an entire year because of new requirements. Thirty new requirements have been established by the end of the third month. On

average, a developer completes three FP per month. With the addition of ten new requirements per month, doubling the number of developers produces the shortest completion time considering that the effect from communication and training overhead is minimal.



Line No.	New Hires
1	2
2	5
3	7
4	10

Figure 29 – System Dynamics (with Requirements Creep) – Hiring at Day 90



Line No.	New Hires
1	0
2	2
3	5
4	7
5	10

Figure 30 – System Dynamics (with Requirements Creep) – Hiring at Day 212

Figure 30 reveals that the same pattern results when requirements creep is added to another simulation trial.

4.2.3 Small Project Conclusions

A software project of this size with only ten experienced developers present at its start does not face much risk of falling behind the default completion date as a result of the addition of more manpower. This is true as long as new developers are not added too close to the default completion time. Both simulations showed results supporting this claim with very similar evidence. In both simulations, hiring of new developers late in the project narrowed the range of difference in benefit gained by hiring 2,5,7, or 10 new employees, demonstrating how hiring late in the project diminishes the positive effect of any amount of hiring on the project's completion time. With requirements creep present, the importance of hiring earlier was even more evident.

4.3 Medium Project Analysis

The experiment was conducted with the following initial parameters that represent a project that is the size of an average commercial software development effort:

- 1000 FP
- 10 experienced developers

The development rate for experienced developers is initially set to 0.12 FP/day. Communication overhead for a group of ten developers is a constant 6 percent of developer time. The net effective development rate for an individual developer is (0.12 FP/day) * (1 – 0.06) = 0.1128 FP/developer day. Both simulation models were executed with the communication overhead as the only negative factor impacting the development rate. Since no new developers were hired, there was no effort required for training and no increase in communication overhead. When

measuring the default behavior of both simulations, the system dynamics simulation finished at 890 days (approximately 3.2 years) while the Easel simulation finished at 885 days. The resulting estimate of 3.2 years agrees with Jones's minimum estimate of twelve months development time for a project of this size (Jones, 106). Table 5 shows the results of each experiment trial.

Number of New Hires	Hire Date	SD Result	Easel Result
5	222	705	716
5	445	770	774
5	668	830	830
10	222	640	642
10	445	725	732
10	668	810	811
15	222	630	639
15	445	720	729
15	668	805	816
20	222	683	696
20	445	755	773
20	668	828	832

Table 5 – Medium Project Simulation Results

The results of both experiments again disprove Brooks's law. However, by doubling the number of developers early during development time, the default completion time of 3.2 years was not cut in half, as observed with the small project. The development time was reduced to approximately 2.8 years. Due to the communication overhead involved, hiring double the amount of experienced developers was not effective at reducing the trial completion time by half the default completion time. Yet doubling the number of developers at a later time results in a completion time almost two months sooner than no hiring at all.

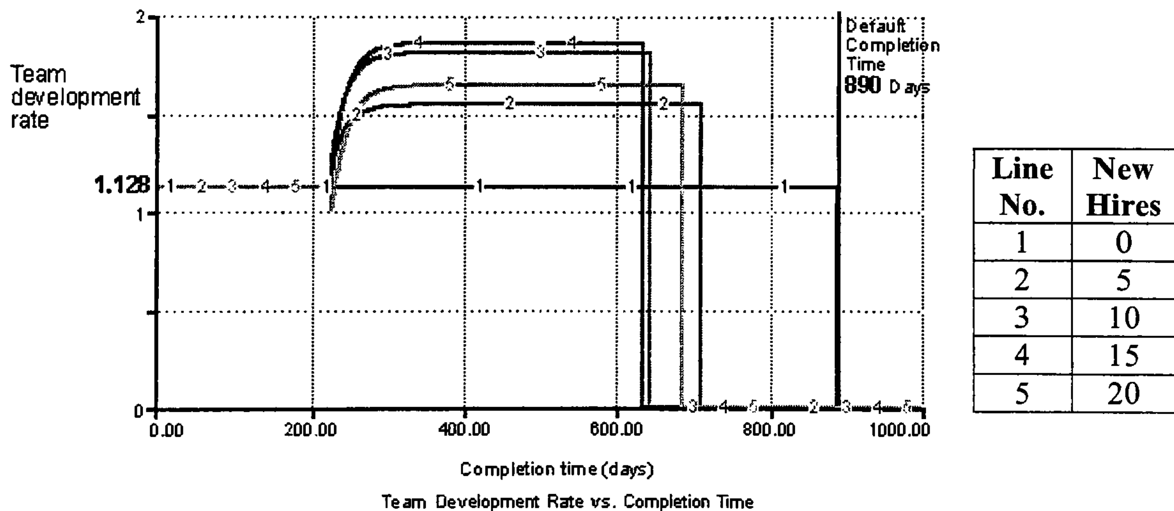
Hiring ten or fifteen new developers results in almost equal completion dates, while hiring twenty new developers is equivalent to hiring five new developers. The Easel results show that when hiring ten or fifteen new developers, the resulting completion dates are anywhere

from five to ten days apart. Evidence of Brooks's law is found when hiring twenty new developers results in a longer completion time than hiring ten or fifteen developers. As with previous results, hiring developers early will result in a shorter completion time.

The hiring of new developers at different hiring times and their impacts on the completion time and software development rate are discussed in section 4.3.1.

4.3.1 Hiring New Developers Midschedule

In the first trial new developers are added early during development. With the default completion time of 890 days, the 222nd day marks the first quarter of the project. Figure 31 shows the software development rate for five different trials, each with a different number of developers hired at each trial.



Line No.	New Hires
1	0
2	5
3	10
4	15
5	20

Figure 31 – System Dynamics – Hiring New Developers at Day 222

Hiring two developers early resulted in a completion time approximately 200 days shorter than the default completion time. By hiring five or seven developers, the completion time is reduced by 65 to 75 days. When twenty new developers are hired, the completion time is 200 days shorter than the default time, yet hiring twenty developers results in a longer completion time than hiring ten or fifteen new developers. For the first time, the training and communication

overhead involved had a negative effect on the overall software development rate. This is shown in Figure 31 by line five's drop below line one (the default team development rate of 1.129 FP/day) at day 222. Figure 32 depicts the communication and training overhead involved. With a medium project, the effects of communication and training overhead were large, unlike the results from a small project where the effects of the communication overhead were minimal.

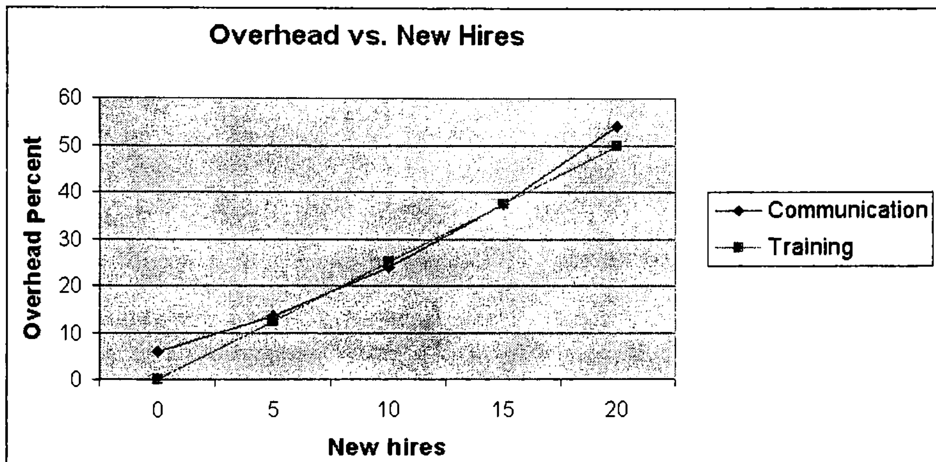
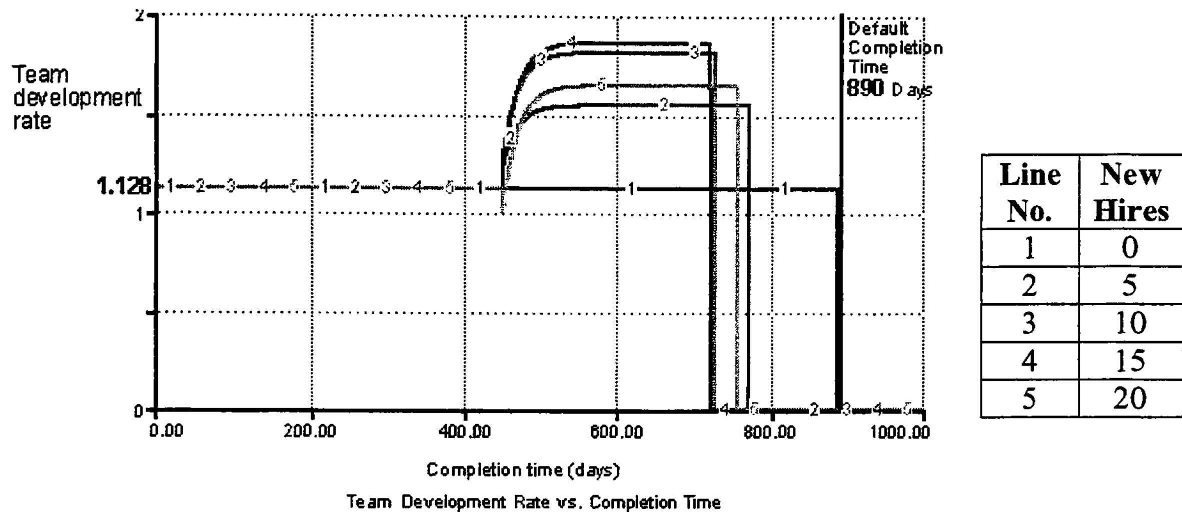


Figure 32 – Overhead for Both Simulations

As mentioned earlier, the default development rate for experienced developers is 0.1128 FP/day. The net effective development rate after hiring 10 new developers is 0.09 FP/day. The net effective development rate after hiring 15 new developers is 0.075 FP/day. Following Madachy (2003), the team development rate is calculated as Team Development Rate = Developer Rate * Number of Total Developers. When hiring 10 new developers, Team Development Rate = 0.09*20 = 1.8. When hiring 15 new developers, Team Development Rate = 0.075 * 25 = 1.875.

Figure 31 shows the similar team development rate, where a team with 10 new developers (line 4) has a development rate of 1.8 and a team of 15 new developers (line 5) has a development rate of 1.1875 FP/day.

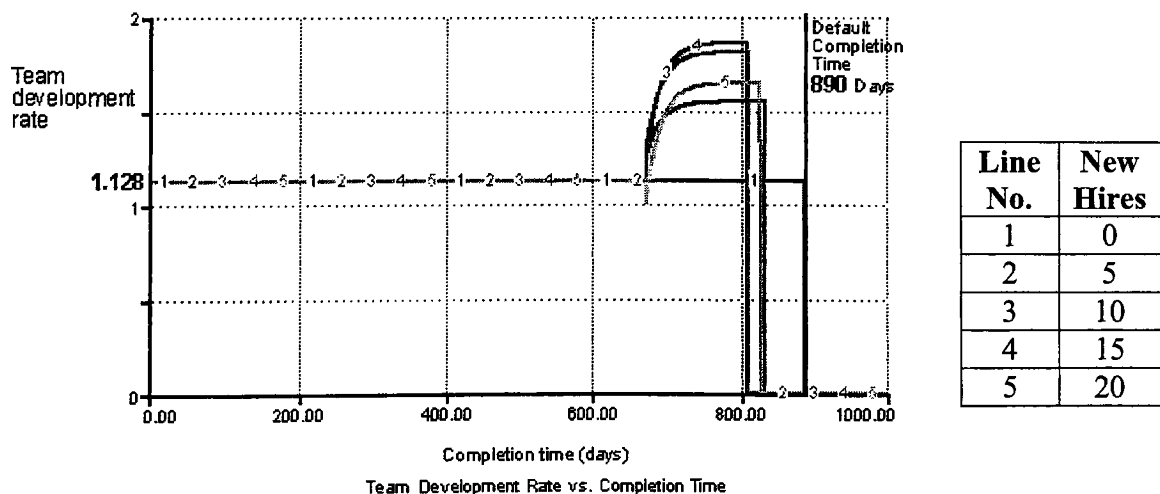
The same effect on the software development rate occurs when hiring at the midpoint of a default project schedule, day 446, and at the three-quarters point, day 668.



Line No.	New Hires
1	0
2	5
3	10
4	15
5	20

Figure 33 – System Dynamics – Hiring New Developers at Day 445

The difference in completion time when hiring five developers and twenty developers is anywhere from two to ten days. Figures 33 and 34 illustrate this difference.



Line No.	New Hires
1	0
2	5
3	10
4	15
5	20

Figure 34 – System Dynamics – Hiring New Developers at Day 668

The Easel simulation also provides some insight into why the last trial resulted in a later completion date. The output from the first three trials reveals the communication and training overhead was not great enough to produce an initial drop in productivity. Instead, the effort created by new developers produced a rise in productivity. This can be seen in the first three trials.

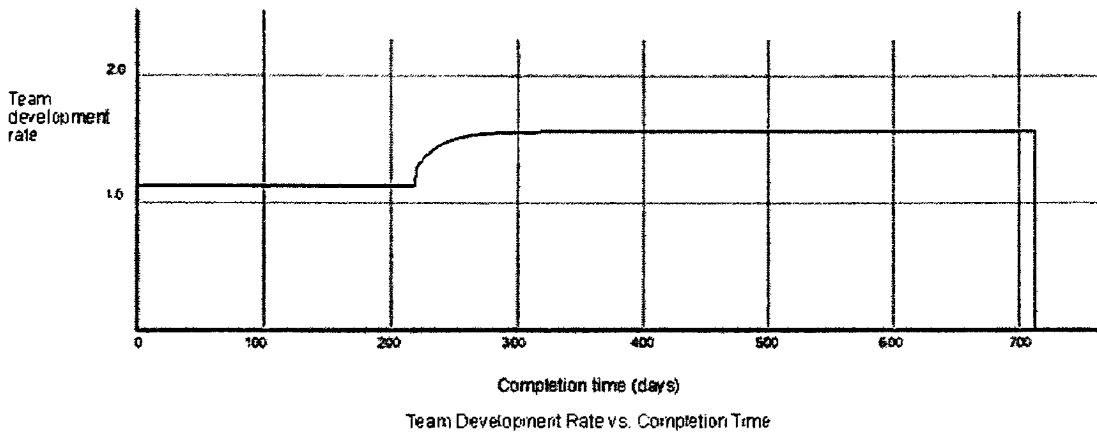


Figure 35 – Hiring 5 New Developers at Day 222

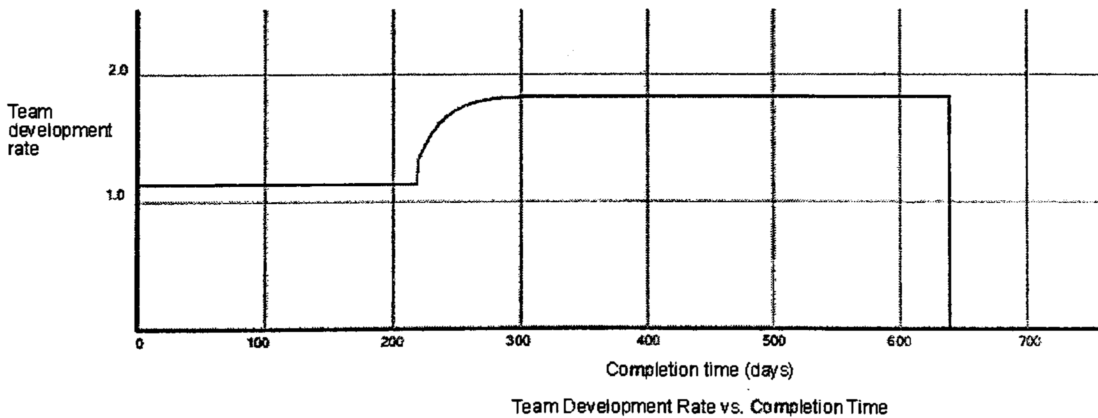


Figure 36 – Easel – Hiring 10 New Developers at Day 222

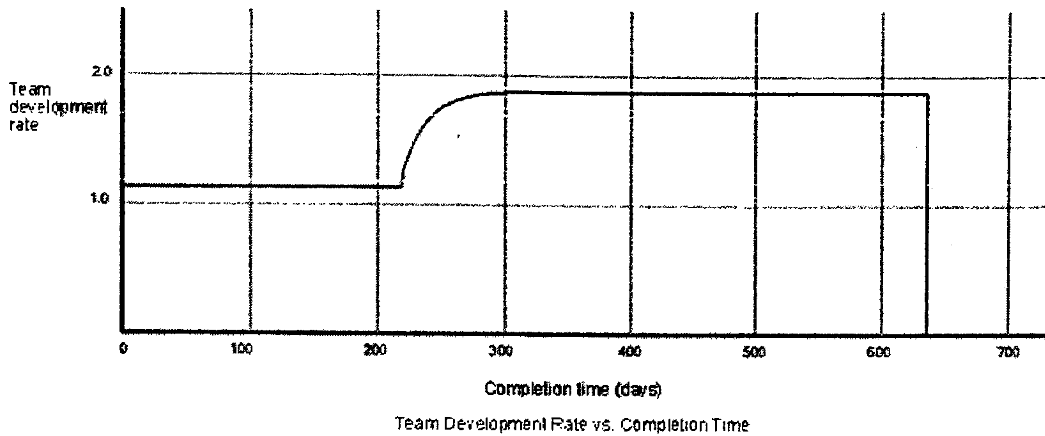


Figure 37 – Easel – Hiring 15 New Developers at Day 222

When hiring 5, 10, or 15 developers, the software development rate increased. When 20 new developers were hired, the software development rate dropped slightly at day 222 because the communication and training overhead created was larger than the effort added by the new developers.

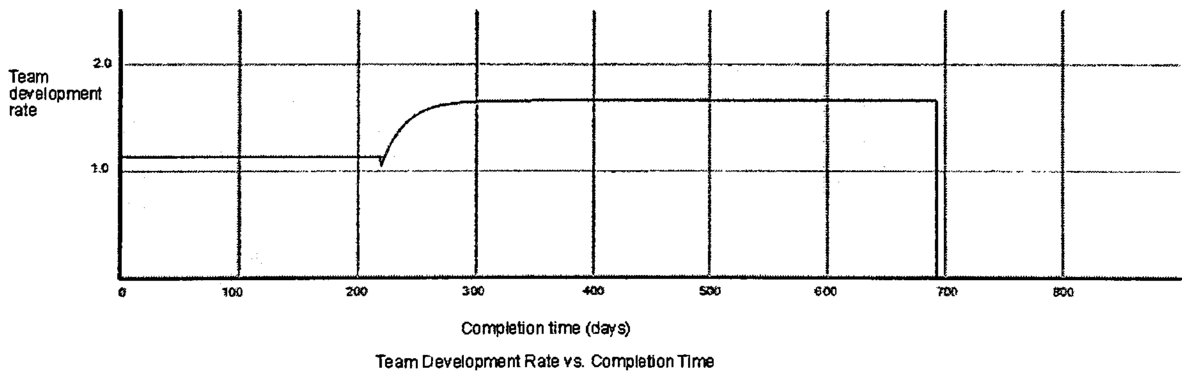


Figure 38 – Easel – Hiring 20 New Developers at Day 22

A discussion of additional Easel trials results is found in Appendix D.

4.3.2 Medium Project with Requirements Creep

Table 6 shows the impact of requirements creep when added to the medium project. Without new manpower, the project completion time was estimated at 2100 days.

Number of New Hires	SD Result (No Creep)	Easel Result (No Creep)	SD Result (with Creep)	Easel Result (with Creep)
5	705	716	1210	1274
10	640	642	990	1026
15	630	639	960	999
20	683	696	1120	1158

Table 6 – Results from Hiring at Day 222 (with and Without Requirements Creep)

With a project of this size, 20 new requirements are added each month, resulting in over 600 new requirements being added before new developers are hired. Since the average developer completes three FP/month, the requirements creep costs between 300 to 500 additional days.

Results produced in the requirements creep trials are similar to the results seen in previous medium project results. The initial hiring of 5 new developers reduces the default completion date by almost fifty percent. The impact of hiring 20 new developers here produces a longer completion time than hiring 10 or 15 new developers. Figure 39 shows the result of this experiment trial.

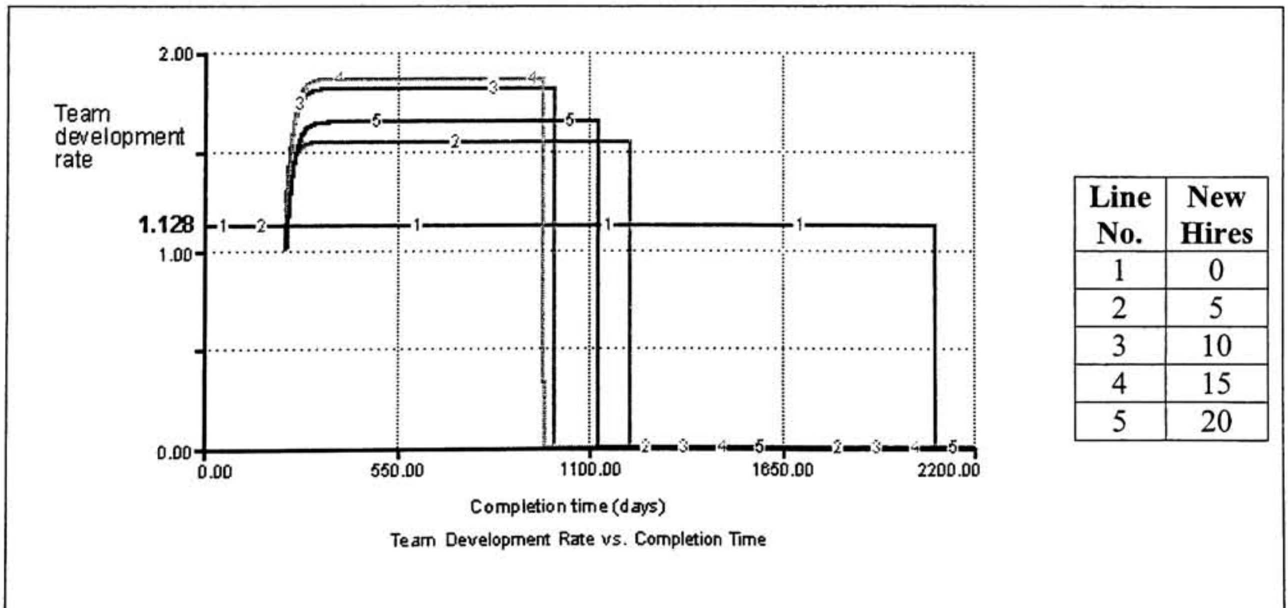


Figure 39 – System Dynamics (with Requirements Creep) – Hiring New Developers at Day 222

4.3.3 Medium Project Conclusions

The results from the medium project data suggest that while Brooks's Law does not hold true in all cases, the communication and training overhead involved at times can outweigh the increase in productivity from hiring new developers. In this experiment, hiring 10 new developers achieves the same effect as hiring 15 new developers. The increase in productivity when hiring new developers was found to be too small to outweigh the communication and training overhead from hiring 20 new developers. For the first time, there was a drop in productivity in both simulations that lead to a longer completion date than found in other trials. Requirements creep here provided similar results to those seen in the small project simulations; the pattern of the software development rate was the same while the completion date was later.

4.4 Large Project Analysis

The experiment was conducted with the following initial parameters that represent a project the size of a large system development effort:

- 10000 function points
- 100 experienced developers
- Developers broken into teams of 5, 10, 15, or 20

Communication overhead was calculated using intragroup communication. Intergroup communication, the overhead generated by the communication of the smaller groups, was not truly represented by either simulation and only a slight penalty was added for each new group, as discussed in chapter 3. Both simulations models were first executed with the communication overhead as the only negative factor impacting the development rate. Since no new developers were hired, there was no effort required for training and no increase in communication overhead.

All default completion dates agree with Jones's estimate of three to fives years development time for a project of this size (Jones, 108).

Table 7 shows the result of each experiment trial where new developers are hired. New developers are hired midproject on day 222.

Simulation Information	Number of New Hires	SD Result	Easel Result
Team size – 5	25	731	732
Default Easel completion date – 848	50	647	648
Default SD completion date – 847	100	547	549
Estimated 3 years	150	485	491
	200	442	452
Team size – 10	25	759	764
Default Easel completion date – 888	50	672	678
Default SD completion date - 890	100	562	565
Estimated 3.2 years	150	500	506
	200	452	463
Team size – 15	25	811	811
Default Easel completion date – 953	50	720	725
Default SD completion date – 955	100	597	599
Estimated 3.4 years	150	527	536
	200	480	492
Team size – 20	25	899	907
Default Easel completion date – 1098	50	793	800
Default SD completion date – 1098	100	667	676
Estimated 3.9 years	150	577	585
	200	525	539

Table 7 – Large Project Simulation Results

The behavior found in the medium system dynamics simulation is also found in the large simulation. Figure 40 shows the completion date and team size for all simulations. The system dynamics and Easel simulation completion dates were again nearly identical.

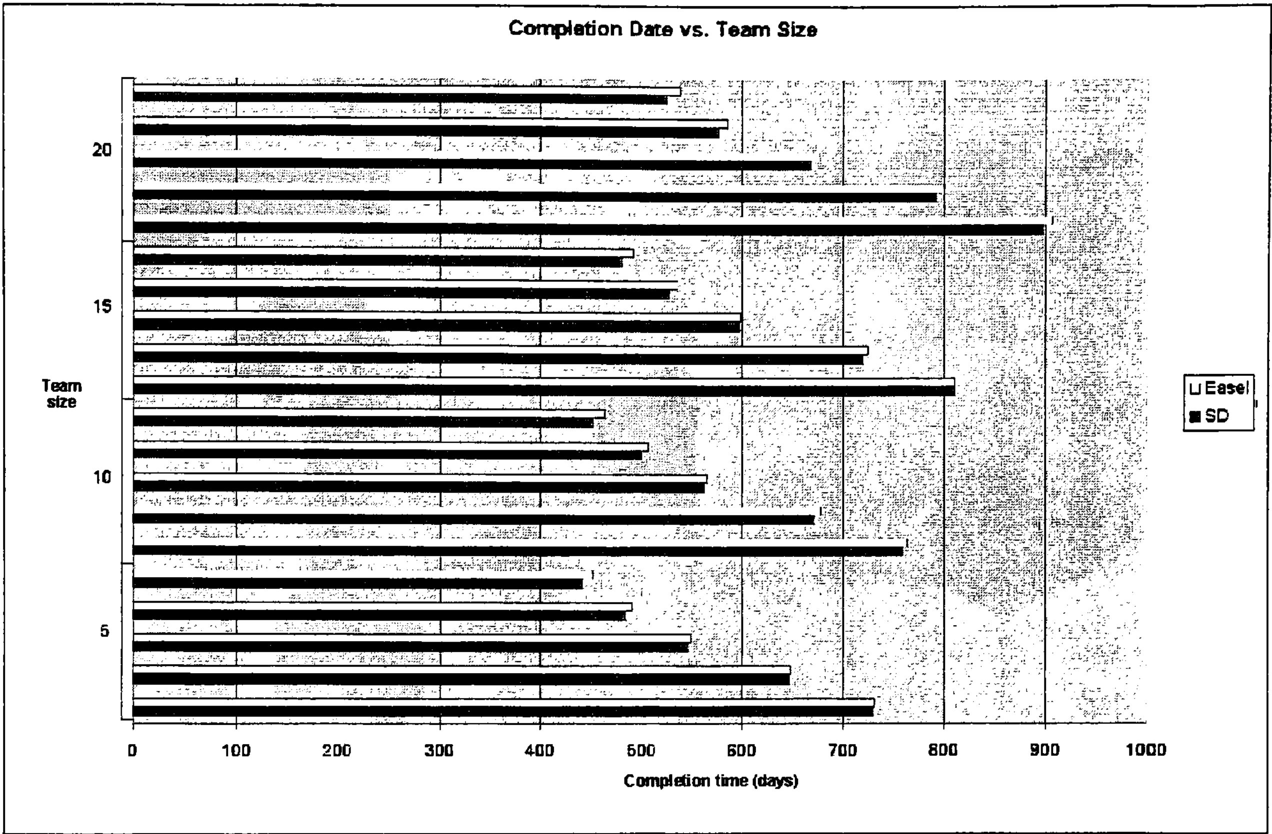
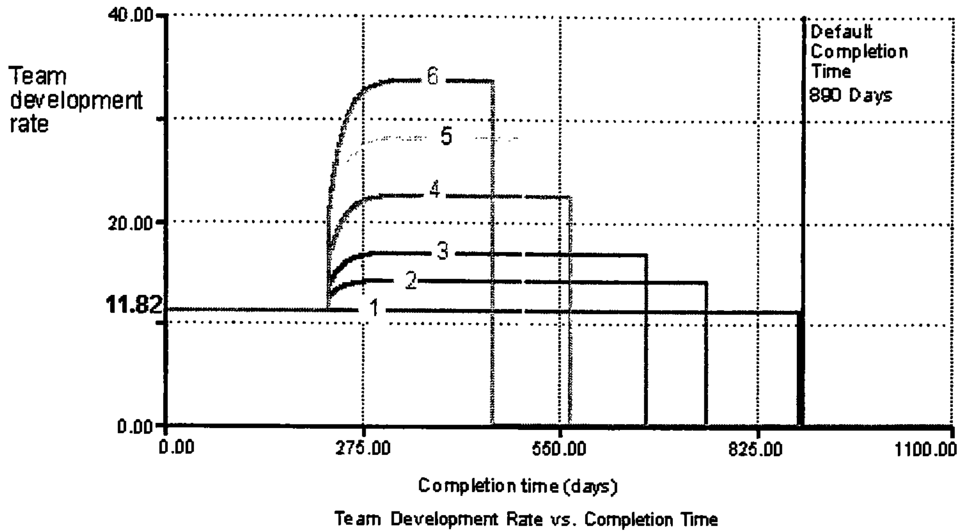


Figure 40 – Results from the Large Project Simulations

In all cases, smaller teams produced the shortest schedules. The system dynamics output for the large project simulation using a team size of 10 is shown below in Figure 41. The pattern shown was the pattern previously seen in the small project simulations. Hiring 50 new developers reduces the completion date by 200 days. This pattern is similar since the communication overhead involved is the same when dealing with a small team project.



Line No.	New Hires
1	0
2	25
3	50
4	100
5	150
6	200

Figure 41 – System Dynamics Results for Team Size of 10

Figure 42 shows that the team completion date increases exponentially as the team size increases. This is due to the communication overhead which increases exponentially (communication overhead is the number of employees squared multiplied by 0.06). Later completion dates are the result of larger team sizes.

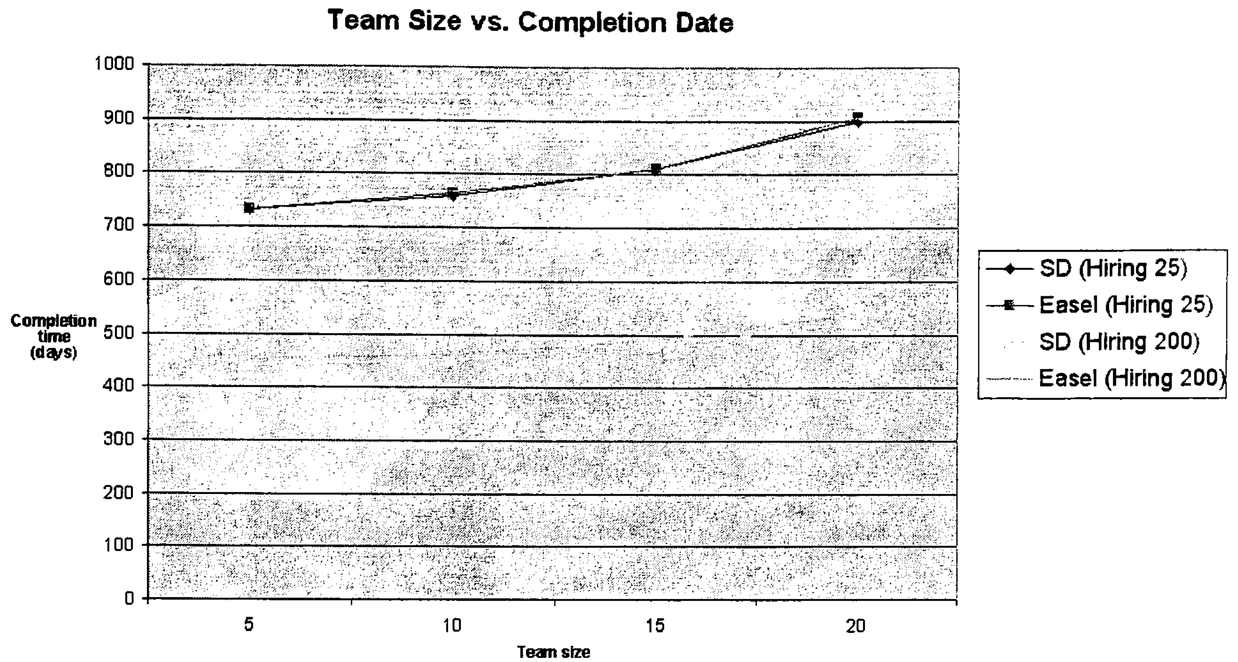


Figure 42 – Large Project – Hiring at Day 222

When comparing trials for each team size, the number of developers hired also played an important role. As shown above in Figure 42, the curve for the completion date increases as the team size grows larger. The number of new hires reduces this rate of change. This is to be expected for two reasons. The first reason is that more developers can complete the work in less time. The second reason is that the communication overhead never increases; new groups are formed for the new developers.

The penalty added for new team leader mentoring was small and distributed over a maximum of 2 percent of the total project time. Requiring at most one-fifth of a team leader's time over four weeks, combined with training and communication overhead, mentoring is too small of a factor to significantly affect the completion date. Therefore adding a new group to the overall system did not have a significant negative effect; it delayed the project from anywhere between zero and five days.

Both simulations supported the idea that smaller teams are more effective due to the reduced amount of communication overhead. All the smaller teams completed projects at an accelerated schedule, regardless of the number of new hires and new teams.

4.4.1 Large Project Conclusions

Initially, the default behavior of both simulations fell into the estimated project duration of three to more than five years. Results showed that smaller teams completed the function points in less time. Further investigation revealed that the rate of completion time change was determined exponentially, which is due to the exponential nature of the communication overhead involved. The number of new hires also played a factor and the exponential rate of change for the completion date was slowed by the increasing number of new hires. The penalty implemented for adding new teams was negligible, and realistically a penalty for introducing additional teams would have a more noticeable effect. Overall, results between simulations showed a variance of only a few days. The simulations demonstrated that smaller teams are more productive than larger teams because the communication overhead is less.

4.5 Summary

Although it may prove more costly for managers of a late project to hire new developers, hiring these developers does not always make the project later. Of the various trials for each project size tested, no trial produced a completion date later than the default completion date. Evidence supporting Brooks's Law first appeared in section 4.3. For a medium project with 10 experienced developers, hiring 10 developers midproject produced the same result as hiring two new developers. With the hiring of 10 more developers, the communication overhead was large enough to negatively affect the development rate. In order to determine what number of new

developers would cause the largest decrease in team development rate, further trials were conducted. As Figure 43 reveals, adding 10 new developers is most detrimental to the team development rate.

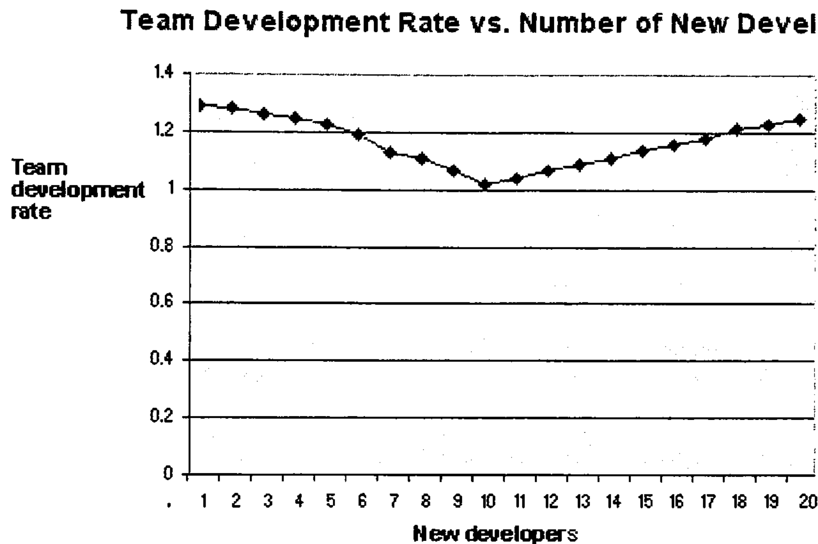


Figure 43 – Team Development Rate for a Medium Project

The medium project results showed a decrease in completion time improvement when hiring was done later, supporting the idea that if additions to the team are planned, it is beneficial to add developers at an earlier date. As hiring is done later, the difference between each completion time narrows and demonstrates how hiring late in the project, regardless of the number of new developers, may not result in schedule reductions. The large project further clarified the importance of keeping team sizes small.

Brooks's Law held true when the degree of communication and training overhead was great enough to have a negative effect on the team development rate. When project team sizes were kept less than or equal to 30 developers and new developers were introduced before or at three quarters of the default completion time, hiring did not cause a late project to be completed at a later time.

CHAPTER 5

FINDINGS, CONCLUSIONS, AND FUTURE WORK

In this thesis, software project dynamics were simulated using both the System Dynamics and Easel simulation techniques. Brooks's Law was used as the project dynamic example. The goal of this work was to identify the relative advantages and disadvantages of both techniques for the study of software project dynamics.

Small, medium, and large-scale projects were simulated using both simulation techniques. The techniques consistently produced similar results, strongly suggesting that the techniques were used correctly and that the resulting models were logically equivalent.

5.1 Findings

The simulation techniques provided insights into the dynamics of Brooks's Law. Although it may prove more costly for managers of a late project to hire new developers, hiring these developers does not always make the project later. Of the various trials for each project size tested, no trial produced a completion date later than the default completion date. Hiring of new developers late showed that the difference between each completion times narrows, demonstrating how hiring late in the project, regardless of the number of new developers, may not result in schedule reductions. The results from the medium project data suggest that while Brooks's Law does not hold true in all cases, the communication and training overhead involved at times can outweigh the increase in productivity from hiring new developers. This was evident when hiring two new developers resulted in the same completion time as hiring twenty new

developers. The large project simulation data suggested that smaller teams are more productive than larger teams because the communications overhead is less.

The remainder of this section presents a comparison of the two techniques based on background knowledge required, debugging difficulty, model maintainability, scalability, object representation, and timing control.

5.1.1 Knowledge Required

Experience with object-oriented languages, such as Java and C++, is necessary to create Easel simulations. As in any object-oriented language, every entity is an object with attributes and procedures. Programmers will also find the syntax similar to an object-oriented language. For example, after the properties and behaviors of an actor are established, an actor is declared by first creating a name for the actor. To complete the declaration, the name is followed by a colon and the object type. An instance of the actor is created by using the keyword "new" followed by the simulation name and a parameter, as shown in Figure 44.

<table>
<tr>
<td>

```
/** Defining the object **/
public class Developer
{
  int devID = 1;   //property

  public Developer()   //behavior
  {
    while(true)
     System.out.println("Working");
  }
}

/** Instantiating the Class **/
Developer d = new Developer(s);
```

</td>
<td>

```
####### Defining the Actor ###########
developer(s:sm): actor type is
        devID: int := 1;  # Property

# Behavior
        for every true do
                outln("Working");

####### Creating the Actor ######
d:developer := new developer(s);
```

</td>
</tr>
<tr>
<td>Syntax for creating an object in Java</td>
<td>Syntax for creating an actor in Easel</td>
</tr>
</table>

Figure 44 - Comparison of Programming Languages

Dot-notation is used to reference properties of an actor and to call procedures. Experience with multi-threaded programming is needed to programmatically manipulate the discrete behavior of the threaded actors. Parallel execution of actor behaviors requires the programmer to call

methods that cause an actor to enter a wait mode for a designated period of time. This is essential for replicating any timed behavior. Basic knowledge of data structures (i.e. stacks, queues, and vectors) is needed for larger-scale simulations because these mechanisms are needed to efficiently manage a large numbers of objects. Beginning Easel programmers must adjust to a few unique qualities of using this property-based language, such as setting up the simulation environment, graphical environment, and depicting neighbor relationships. The reference material provided with the Easel distribution describes all of these new aspects in detail.

The use of system dynamics simulation development environment does not require programming skills in order to successfully create and run a simulation. The drag-and-drop simulation development environments (i.e. iThink & Vensim) facilitate the creation of simulations without writing code and equations, therefore reducing the amount of mathematical knowledge required to an algebra level. These simulation development environments emphasize the importance of using flow diagram symbols when creating a model.

The details of the simulation are represented through a set of equations. Since differential equations are the foundation for system dynamics models, knowledge of differential equations may be helpful in developing a model. The number of requirements (uncompleted function points) in the system dynamics Brooks's Law model was determined using equation 5.1. The number of requirements at time t is determined by subtracting completed requirements (the software_development_rate multiplied by the change in time) from the number of previous uncompleted requirements (requirements(t-dt)).

$$\text{requirements}(t) = \text{requirements}(t - dt) + (- \text{software_development_rate}) * dt$$

Equation 6 - Modeling Number of Requirements

Experience with equations like the one above will aid those interested in understanding dynamic systems.

5.1.2 Debugging

When developing the underlying code or equations, both simulation development environments verify syntax. Easel uses the traditional compiler method; it notifies the user of a complier error using a caret to point to the error in the code and provides a small error description. The descriptions provided are helpful. The iThink environment does not allow the creation of the equation unless the model and the syntax are correct. It also provides helpful descriptions of the syntax error through a dialog box.

Since any variable can be graphed using the iThink environment, it was at times easier to debug problems with this software. While printing a variable value at different time intervals was effortlessly accomplished with the Easel environment, no automatic graphing feature was present. To provide the same visual output for the Easel simulation, new procedures were created to print the visual display. Once these procedures were in place, the graphical output can be duplicated.

A major difference between the simulation development environments is the method for debugging logic errors. The Easel simulation allowed for debugging at an individual level, while the debugging of a system dynamics simulation took place at the system level. It was easier to observe and debug the properties of one developer than debug a single entity that represents all the developers. Debugging at the system level could also take place within the Easel simulation if needed.

5.1.3 Maintainability

The graphical model in the iThink environment facilitated code maintenance. Changes to the Easel model required searching through text to identify a type (object) whose properties need changing. It may be more time consuming to add new properties and behaviors to existing objects in a system dynamics simulation. In Easel, properties may be declared as global attributes. New levels, converters, and/or relationships must be established in system dynamics to achieve the same effect. Figure 45 shows the addition of team size in both simulations. In the system dynamics example, notice that every entity that requires information about the team size must have a line connecting it to that level.

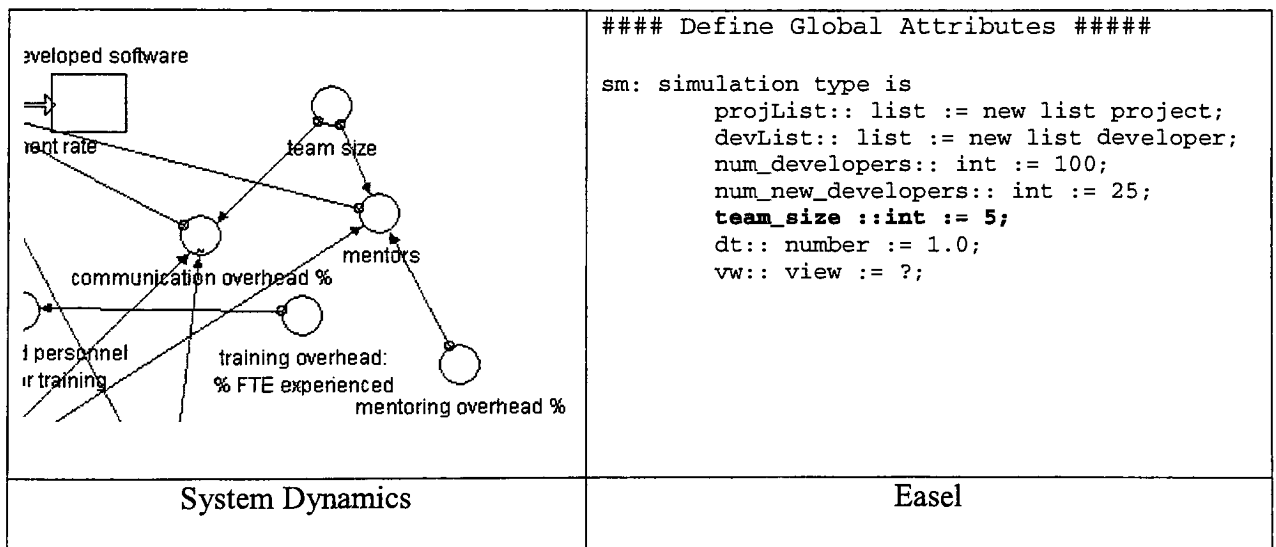


Figure 45 - Adding a Team Size Feature

5.1.4 Scalability

Both simulation techniques can work on a macro and micro scale, although system dynamics models are often created on a macro scale. In system dynamics models of large scale systems, entities are often grouped together as one level. For example, in the Brooks's Law

model, all experienced developers were represented by one level (experience personnel), as shown in Figure 46.

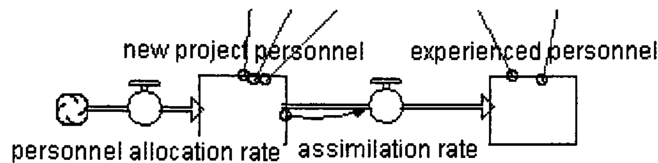


Figure 46 - Personnel Levels

This model does not focus on an individual experienced developer. If each experienced developer was represented by its own symbol in the model, the model and equations involved would become increasingly complicated as the number of experienced developers increased. This is why a level may be used to represent a large number of entities. The end result being that every entity represented in this group will have the same behavior. With Easel, each individual is represented separately, as its own threaded object. Since individuals are represented separately, their attributes and behavior may differ from other individuals.

Both simulation techniques can represent thousands of interacting objects. If a large number of objects are needed and each object behavior may differ, the Easel simulation technique may be a more suitable choice.

5.1.5 Object Representation

In large scale system dynamic simulations, objects are represented by a single level, with converters or flows manipulating that level; all common objects (e.g., experienced developers) are grouped together as one single entity. For example, the assimilation rate for new developers was set using a flow between new developers and experienced developers, as shown in Figure 46. The number of new developers was changed as a function of time where the assimilation

rate was the same for every new developer. This level acts like a black box, leaving much of the object implementation detail hidden.

When simulating large systems, system dynamics does not allow the representation of objects in a programming sense, and it is easier to translate real world objects into Easel types (objects). All developers, modules, and projects were represented as Easel types within the Brooks's Law simulation. Since Easel is a property-based language, types can be given several attributes that represent the characteristics of one type. There is no practical constraint on how detailed an entity can be modeled. The code in Figure 47 shows the properties of the Easel Brooks's Law simulation developer. Each developer has properties that make it unique.

```
developer(s:sm): actor type is
# developers implement modules
        devID: int := ?;
        mod::module := ?;
        num_mods::int := 0;
        devT::number := 0.0;
        modList::list := new list module; # List of uncompleted modules
        modList1:: list := new list module; # List of completed modules
        dev_state:: dev_states:= free;
        communication_overhead::number :=0.0;
        productivity_rate:: number := ?;
        tm:number := 0.0;
```

Figure 47 – Easel Brooks's Law Code: Developer Properties

In large scale system dynamics simulations, the developers would share properties. For example, each new developer is assigned a productivity rate. In system dynamics, this rate is the same for all new developers. In Easel, the rate can be observed as the rate for that developer, instead of the rate for all developers. The effect on one object can then be observed directly, whereas in the system dynamics simulation the effect on a group of developers is observed.

In addition, Easel types can also create other types, which is not possible in system dynamics where the model is fixed at execution time. The ability to represent each entity

separately and change the model during execution time makes Easel a powerful simulation technique.

Relationships between objects can be created in both simulations. The iThink environment uses graphical depictions, such as arrows or flows, to show relationships between levels and converters. Easel does not graphically display these relationships, but the construction of the simulation reveals how the objects relate. One aspect that is unique to Easel relationships is the ability to depict topographical relationship between objects. These types act as neighbors that can see information relating to other objects. For example, a factor like communication overhead might be reduced depending on distance separating developers.

The Easel simulation also provided the means by which to represent two other actors: the client and manager. The client was responsible for implementing the requirements creep and the manager was responsible for distributing modules and monitoring project progress. This proved to be a more realistic representation of a software development system and would be easier to translate the Easel model into real world situations.

5.1.6 Timing Control

Both the Easel and system dynamics simulations have a clock that simulates real time. In the discrete Easel simulation model, each actor has its own thread. All actor threads run concurrently. Scheduling methods are available for placing delays, wait intervals, and other controls on an actor's thread. In the Brooks's Law simulation, after the amount of time required to complete a function point is calculated, an actor enters a wait state for that period of time. This wait period represents a period of development. During that time, if any changes are made to the productivity rate of that actor, these changes do not apply until the actor begins working on

a new function point. Programmers are responsible for implementing wait intervals that represent periods of productivity.

Time in the system dynamics simulations is continuous. The simulation technique uses a clock that can also be manipulated by the use of built-in functions. In the Brooks's Law system dynamics simulation, any changes to the productivity rate are implemented with the next tic of the clock. Instead of using threads, actors and other entities are represented as equation variables that change as a function of time. Although clock manipulation methods are available, they do not need to be used to represent a period of productivity.

5.2 Conclusions

Both techniques may be used to simulate subsets of the software development process. The implementation of each technique provided insight as to conceptually where one technique applies more than another and how the techniques differ.

When beginning to build a simulation of a process, one starts with developing a model of the system. The model focus is limited to the problem area and any contributing factors. With system dynamics, the first step is to discover and document the feedback loops involved which later provide the basis for a CLD. The CLD is then transformed to a flow diagram followed by a mathematic representation of the model. This is the recommended procedure for developing a system dynamics simulation. Currently, there is no preferred method for modeling a system when creating an Easel simulation. Existing modeling techniques such as UML may prove helpful, however Easel is not truly an object-oriented language therefore it can be difficult to transform a UML model into an Easel simulation. The modeling standard established for system

dynamics helps to facilitate simulation development, whereas Easel simulation developers may find this step more difficult due the absence of a preferred modeling technique.

As a programming language, Easel provides some flexibility when simulating decision making. System dynamics was not designed to be a programming language and therefore lacks the proper control structures needed to simulate these actions. For example, to dynamically determine the optimal number of developers to hire, an entity in the simulation would be required to observe variables and execute appropriate statements after considering both the final completion date predicted and costs involved. System dynamics was not created to handle such situations since the model must be fixed at execution time. To reach the same conclusion, the person running the simulation would have to execute it several times with different values for input parameters. Easel does not require that the model be fixed at simulation time, allowing dynamic changes to be made.

The view level depicted by each simulation is a key difference between the techniques. With system dynamics, common entities are often grouped together. By grouping common entities together as one entity, they share attributes and behaviors creating a high level view of the system. Entities in Easel are represented independently and since every actor in Easel has its own properties and behaviors, this provides a detailed and low level view of the system.

Consequently, this low level view makes it easier to validate and map the behaviors of actors to those of real world entities. With the Brooks's Law simulations, concepts like repartitioning of work would demonstrate this. To simulate this action using system dynamics, a number representing a period of time may be used to depict the time delay required for repartitioning. In Easel, removing all the function points from a developer's list, and then redistributing them would represent this action. A certain period of time would also be allotted

to this process. More work may be required to simulate this in Easel, but it would provide a more realistic and detailed representation of the real world process.

Depending on the needs of the user, one simulation technique may be more suitable than another. If the focus lies on the individual level and the need to simulate dynamic decision making is present, Easel would be more suitable choice. However if simulation simplicity is preferred, the abstract view created by the system dynamics technique may be the ideal. As one might expect, the established modeling standard and availability of commercial simulation development environments for system dynamics makes it more attractive for a wide range of users. Yet Easel offers flexibility and a low level depiction which is often difficult to produce with system dynamics.

5.3 Recommendations for Future Work

The experiments conducted provide many opportunities for further work to distinguish the differences between system dynamics and Easel. A few possibilities are presented below.

- Decision making of managers when determining when to hire new developers, and how many to hire, could be explored. Many of the simulations showed little difference in the completion dates where various amounts of new developers were hired. This decision processes would be easier to implement using Easel due to its model flexibility during execution.
- In Easel, longer function point development periods may depict shortcomings of discrete simulation modeling.
- The Easel model could also be modified to show the sequential constraint of modules or function points. It would be easier to implement this in Easel considering the modules

are represented as independent entities. Actors in Easel could communicate information about modules currently finished, and the anticipated completion date of other modules. In addition, communication overhead could be adjusted for neighbor relationships and proximities in the Easel simulation.

- The Easel simulation currently accounts for the redistribution of modules after new developers have been hired, but no time is given to this task. To show the extensibility of system dynamics models, this is an aspect, or delay, that could be added.
- Validation of the system dynamics simulation may be attempted. It may prove difficult to translate equations to real world behaviors.

APPENDIX A

SYSTEM DYNAMICS SIMULATION EQUATIONS

Model Equations without Enhancements

Madachy's (2003) basic Brook's Law model equations are provided below.

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

DOCUMENT:  This level represents software function points that have been
implemented.

INFLOWS:
software_development_rate = nominal_productivity*(1-
communication_overhead_%/100.)*(.8*new_project_personnel+1.2*(experienced_per
sonnel-experienced_personnel_needed_for_training))
DOCUMENT:  The development rate represents productivity adjusted for
communication overhead, weighting factors for the varying mix of personnel,
and the effective number of experienced personnel.

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

DOCUMENT:  The number of experienced personnel.

INFLOWS:
assimilation_rate = new_project_personnel/20
DOCUMENT:  The average assimilation time for new personnel is 20 days.

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

DOCUMENT:  The number of new project personnel.

INFLOWS:
personnel_allocation_rate = pulse(10,100,999)
OUTFLOWS:
assimilation_rate = new_project_personnel/20
DOCUMENT:  The average assimilation time for new personnel is 20 days.

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

DOCUMENT:  The project size is 500 function points.  This level represents
the number left to be implemented.

OUTFLOWS:
```

```
software_development_rate = nominal_productivity*(1-
communication_overhead_%/100.)*(.8*new_project_personnel+1.2*(experienced_per
sonnel-experienced_personnel_needed_for_training))
DOCUMENT:  The development rate represents productivity adjusted for
communication overhead, weighting factors for the varying mix of personnel,
and the effective number of experienced personnel.

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

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

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

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

Enhancements

Requirements Creep Simulation Equations

For the requirements creep simulation, the following code was added.

```
requirements_creep_rate = PULSE(Intitial_Requirements*0.02,30.5,30.5)
```

The equation for determining requirements was altered as follows:

```
requirements(t) = requirements(t - dt) + (requirements_creep_rate -
software_development_rate) * dt
```

Large Project Simulation Equations

A team size variable was added to Madachy's original model.

```
team_size = 5
```

The following code was modified to simulate the development staff when divided into teams.

```
communication_overhead % =
IF(MOD(INT(experienced_personnel+new_project_personnel), team_size)!=0)
THEN
((INT(experienced_personnel+new_project_personnel)/team_size)*(team_size*team
_size*0.06)+(MOD(INT(experienced_personnel+new_project_personnel),team_size)*
MOD(INT(experienced_personnel+new_project_personnel),team_size)*0.06))/((INT(
experienced_personnel+new_project_personnel)/team_size)+1)
ELSE(team_size*team_size*0.06)
```

To simulation mentoring overhead, the following equation was added.

```
mentoring_overhead_% = 20

mentors = ((new_project_personnel/team_size)*mentoring_overhead_%/100)
```

The software development rate then had to be altered to include this overhead.

```
software_development_rate = nominal_productivity*(1-

communication_overhead_%/100.) *(.8*new_project_personnel+1.2*

(experienced_personnel-experienced_personnel_needed_for_training - mentors))
```

APPENDIX B

EASEL CODE

Small and Medium Project Code

```
#################################################################
# Based on a simulation written by Alan Christie, January 2002
# Copyright 2002, Carnegie Mellon University
#
# Modified by Alicia Strupp, February 2004
# Purpose:  To simulate a software development organization in
#           order to observe Brooks' Law
# ################################################################

mod_states: type is enum(unassigned, assigned, completed);
prj_states: type is enum(unallocated, allocated, mods_completed, closed);
dev_states: type is enum(occupied, free);

# ######################## Define Global Attributes #######################

sm: simulation type is
        projList:: list := new list project;    # list of projects
        devList:: list := new list developer;   # list of developers
        num_developers: int :=10;               # number of experienced
developers
        num_new_developers:: int :=20;
        dt:: number := 1.0;                     # dt = 1 day
        vw::view := ?;
        experienced_rate :: number := 1.2;      # experienced developer
productivity
        inexperienced_rate :: number := 0.8;    # unexperienced developer
productivity
        man_day_fraction :: number := 0.06;     # used for communication
overhead

# ####################### Define Project and Modules Types ################

project: type is
        modList:: list := new list module;            # list of modules for a
project
        modListCompleted:: list := new list module;   # list of completed
modules
        prj_state::prj_states := unallocated;
        proj_ID::int := ?;
        prj_modules:: int := 1000;                    # number of modules
        prj_creepPerMonth:number := (prj_modules *0.02);    # number of
creeping requirements per month
        hire_date::int :=668;                         # date of hiring

module: type is
```

```
# a project consist of multiple modules
        mod_state::mod_states := unassigned;          # module state
        mod_ID::int := 0;
        dev_time:: number := 10.0;    # for fixed dev time - .1 function point
per day
        startT:: number:=?;           # time module is started
        endT:: number:=?;             # time module is completed
        proj:: project := ?;          # project the module belongs to

# ############## Define Actors: Developer, Manager, & Client ##############

developer(s:sm): actor type is
# developers implement modules
        dev_ID:: int := ?;
        mod:: module := ?;            # Module currently working on
        num_mods:: int := 0;          # Number of module assigned
        devT:: number := 0.0;         # Developer time
        modList:: list := new list module;      # List of modules waiting to
be completed
        modList1:: list := new list module;     # List of modules completed
        dev_state:: dev_states:= free;
        communication_overhead:: number :=0.0;
        productivity_rate:: number := ?;
        tm:: number := 0.0;

        # Complete modules waiting to be completed
        for every true do
                if (length modList) > 0 & dev_state = free then # work on
next module
                        dev_state := occupied;
                        mod := pop modList;     # get module
                        push(modList1, mod);
                        mod.startT := devT;     # set start time
                        communication_overhead := (s.man_day_fraction
*(s.num_developers)*(s.num_developers));
                        mod.endT :=
devT+mod.dev_time/((productivity_rate)*(1-communication_overhead/100)); # get
end time
                        devT := max(mod.endT, tm);
                        num_mods:=num_mods-1;
                        outln("mod.dev_time: ", mod.dev_time, "new: ",
mod.dev_time/((productivity_rate)*(1-communication_overhead/100)));
                        outln("dev ID: ", dev_ID, "   mod ID: ", mod.mod_ID,
"   productivity rate: ", productivity_rate);
                        outln("start time: ", mod.startT, " end time:
",mod.endT, "   mods remaining ", num_mods);

outln("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
                        wait mod.dev_time/((productivity_rate)*(1-
communication_overhead/100)); # wait until completed
                        mod.mod_state:= completed;
                        dev_state := free;
                else
                        tm:= tm+s.dt;
                        wait s.dt;

manager(s:sm): actor type is
```

```
# managers assign modules
        prj:project:=?;
        hire:int:=0;
        numOfmodules:int:=?;
        prj_size:int:=?;
        # Assign modules and check status of modules
        for every true do
                for prj: every s.projList do
                        if prj.prj_state != closed then # observe progress
                                if prj.prj_state = unallocated then
                                        assign_mods_to_developers(s, prj); #
get modules distributed
                                        prj.prj_state := allocated;
                                else if prj.prj_state = allocated &
all_mods_completed(s, prj) then
                                        prj.prj_state:= closed;           #
mods_completed
                                        outln("all mods completed for project
", prj.proj_ID, " Time: ", s.skdr.clock);

outln("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
                wait s.dt;
                prj_size := length(s.projList);
                prj1:project := s.projList[0];

                # Hire more people at day 100 if needed
                if (prj_size!=0) then
                        if (s.skdr.clock > prj1.hire_date & hire = 0 &
s.num_new_developers!= 0) then
                                new_developer_training(s, prj1);

client(s:sm): actor type is
# responsible for distributing modules and requirements creept
        i::int := 0;
        p::project := new project;
        p.proj_ID := i;
        push(s.projList, p);
        for j:each 1..p.prj_modules do # create all the modules
                m::module := new module;
                m.mod_ID := j;
                m.proj := p;
                push(p.modList, m);

        for every true do   # Add creeping requirements every 30.5 days
                if (p.prj_state != closed) then
                        if((mod(s.skdr.clock, 30.5)) < 1) then
                                assign_newmods_to_developers(s, p); #
allocate new modules
                wait 1.0;

# ####################### Procedure for adding new developers #############

addDevelopers(s:sm, p:project, numOfnewbies:int): action is
# add specified number of developers to the project
        total_mods:int := 0;
        new_id:int :=length(s.devList);
        mod_list_len:int := 0;
```

```
        mod:module:=?;
        s.num_developers := s.num_developers + s.num_new_developers; # add to
the number of developers
        for j:each 1..numOfnewbies do
                d:developer := new (s, developer(s)); # create new actor
                new_id  := new_id + 1;
                d.dev_ID := new_id;                     # assign ID
                d.devT := p.hire_date;          # time to bring in recruits
                d.productivity_rate := s.inexperienced_rate;        # set
productivity rate
                d.communication_overhead := (s.man_day_fraction
*(s.num_developers)*(s.num_developers));
                push(s.devList, d); # add to simulation list of developers


# ####################### Procedure for reassigning modules ###############

reassignModules(s:sm, p:project): action is
        newModList:: list := new list module;
        # clear developer module lists and send modules for reassignment
        for d: every s.devList do
                for m: every d.modList do
                        push(newModList, m);
                d.num_mods :=0;
                emptyModList:: list := new list module;
                d.modList := emptyModList;  # erase list
        p.modList := newModList;  # initialize project list with incomplete
modules
        assign_mods_to_developers(s,p);

# ####################### Procedure for training of new developers #########
new_developer_training(s:sm, p:project): action is
        trainerList:: list := new list developer; # set up a list of trainers
        id:: number := ?;
        found:: boolean:=true;
        training_percent :: number := 0.25;
        numOfTrainers:: int := trunc((s.num_new_developers) *
training_percent); # get number of trainers required

        # Get a trainers
        for dev:each s.devList do
               push (trainerList, dev);
               dev.productivity_rate:=s.experienced_rate-
(s.num_new_developers/(s.num_developers)*training_percent*s.experienced_rate)
;

        addDevelopers(s, p, s.num_new_developers); # add new developers
        reassignModules(s, p);      # reassign modules
        assimilation (s, trainerList);    # finish assimilation

assimilation(s:sm, trainerList:list): action is
        # Assimilation of new employees
        rate:: number := 20;
        last_rate:: number :=0.0;
        assim:: boolean:= true;
        percent:: number :=0.0;
        training_percent :: number := 0.25;
```

```
        min:: number := 0.00001;
        training_prod:: number := (s.num_new_developers/(s.num_developers-
s.num_new_developers)*training_percent*s.experienced_rate);

        # slowly increases developer productivity rate
        for every assim do
                rate := rate-(rate/20);
                percent := (1-rate/20)-(last_rate);
                for d: every s.devList do
                        if (d.dev_ID > (s.num_developers -
s.num_new_developers) & d.dev_ID < s.num_developers+1) then
                                d.productivity_rate:=d.productivity_rate +
((s.experienced_rate-s.inexperienced_rate) *percent);
                last_rate := last_rate + percent;
                if rate < min then
                        assim:= false;
                for t: every trainerList do
                        t.productivity_rate:=t.productivity_rate +
(training_prod*percent);
                wait 1.0;

# ####################### Procedure for checking project status ############

all_mods_completed(s:sm, prj: project): boolean is
        for mod: every prj.modList do
                if mod.mod_state != completed then return false;
        for d: every s.devList do
                if d.dev_state != free | (length d.modList) > 0 then return
false;
        return true;

# ####################### Procedure for assigning modules ##############

assign_mods_to_developers(s:sm, prj:project): action is
        min_mods::int:= ?;
        devList1::list := new list developer;
        d::developer:=?;
        i::int := ?; j::int := ?;
        for mod: every prj.modList do
                min_mods:=1000;
                for dev: every s.devList do     # smallest number of modules
assigned
                        if dev.num_mods < min_mods then
                                min_mods := dev.num_mods;
                                d:= dev;
                i:= 0;
                devList1 := list'[d];
                d := pop devList1;
                for dev1: every s.devList do    # find developer with least
modules
                        if dev1.num_mods = min_mods then
                                push(devList1, dev1);
                                i:= i+1;
                j:= rand(1,i);
                d:= devList1[j-1];
                d.num_mods:=d.num_mods+1; # increase module count
                push(d.modList, mod);  # add module to developer list
```

```
                mod.mod_state := assigned;

# ####################### Procedure for assigning new modules #############

assign_newmods_to_developers(s:sm, prj:project): action is
        min_mods::int:= ?;
        devList1::list := new list developer;
        d::developer:=?;
        i::int := ?; j::int := ?;
        # Assign modules fairly
        for mod:each 1..prj.prj_creepPerMonth do
                m::module := new module;
                prj.prj_modules:= prj.prj_modules +1;
                m.mod_ID := prj.prj_modules;
                m.proj := prj;
                push(prj.modList, m);
                min_mods:=1000;

                for dev: every s.devList do
                        if dev.num_mods < min_mods then
                                min_mods := dev.num_mods;
                                d:= dev;
                i:= 0;
                devList1 := list'[d];
                d := pop devList1;
                for dev1: every s.devList do
                        if dev1.num_mods = min_mods then
                                push(devList1, dev1);
                                i:= i+1;
                j:= rand(1,i);
                d:= devList1[j-1];
                d.num_mods:=d.num_mods+1;
                push(d.modList, m);
                m.mod_state := assigned;


# ####################### Procedures for graphical view ####################

graph1(s:sm): action is
# Development Time vs. Effort
        y0: number:= 100.0;
        dy::number:= 0.0;
        x0:: number:= 100.0;
        dt:: number:=1.0;
        first_flag::boolean := true;
        working::boolean := true;
        productivity:: number := 0.0;
        prj:: project := s.projList[0];
        wait 5.0;
        # depicts a line showing team development rate
        for every working do
                productivity := 0.0;
                for dev: every s.devList do
                        productivity := productivity + (dev.productivity_rate
- dev.productivity_rate*((s.man_day_fraction
*(s.num_developers)*(s.num_developers))/100));
```

```
                if first_flag then y0:= 100 + (200- productivity*10);
                first_flag := false;
                depict(s.vw, paint(polyline(2.0, x0, y0 , x0 + dt, 100 + (200
 - productivity*10)), (blue)));
                x0 := x0 + dt;
                y0 := 100 + (200 - productivity*10);
                wait 1.0;

                if prj.prj_state = closed then # stop drawing
                        working := false;
                        depict(s.vw, paint(polyline(2.0, x0, y0 , x0, 300),
 (blue)));
                dy := productivity;

 displayGraphs(): actor type is
        sim.vw := new view(sim, "*** SW process schedule ***", ivory, nil);
        null make_window(sim.vw, 1);
        displayEffort();  # display effort(productivity graph)

 displayEffort(): action is
        # x axis
        dept1: cee  := paint(polyline(3.0, 100, 300, 1000, 300), (black));
        depict(sim.vw, dept1);
        # y axis
        dept2: cee := paint(polyline(3.0, 100, 300, 100, 50), (black));
        depict(sim.vw, dept2);
        # horizontal lines
        dept3: cee := paint(polyline(1.0, 100, 200, 1000, 200), (gray));
        depict(sim.vw, dept3);
        dept4: cee := paint(polyline(1.0, 100, 100, 1000, 100), (gray));
        depict(sim.vw, dept4);
        # vertical lines
        dept5: cee := paint(polyline(3.0, 300, 300, 300, 50), (gray));
        depict(sim.vw, dept5);
        dept6: cee := paint(polyline(3.0, 400, 300, 400, 50), (gray));
        depict(sim.vw, dept6);
        dept7: cee := paint(polyline(3.0, 500, 300, 500, 50), (gray));
        depict(sim.vw, dept7);
        dept8: cee := paint(polyline(3.0, 600, 300, 600, 50), (gray));
        depict(sim.vw, dept8);
        dept9: cee := paint(polyline(3.0, 200, 300, 200, 50), (gray));
        depict(sim.vw, dept9);
        dept10: cee := paint(polyline(3.0, 700, 300, 700, 50), (gray));
        depict(sim.vw, dept10);
        dept11: cee := paint(polyline(3.0, 800, 300, 800, 50), (gray));
        depict(sim.vw, dept11);
        dept12: cee := paint(polyline(3.0, 900, 300, 900, 50), (gray));
        depict(sim.vw, dept12);
        graph1(sim);


# ####################### Procedure for beginning simulation #############


SEprocess(): action is
# initializes new manager, developers, projects and associated modules
        s::sm := new sm;
```

```
        for i:each 1.. s.num_developers do
                d:developer := new (s, developer(s));
                d.dev_ID := i;
                d.productivity_rate := s.experienced_rate;
                push(s.devList, d);
        null new (s, client(s));        # create client
        null new(s, manager(s));        # create manager
        null new(s, displayGraphs());   # used to display graph output
        wait s;

# ####################### Call to start simulation#########################

SEprocess();
```

Large Project Code

```
# ############################################################
# Based on a simulation written by Alan Christie, January 2002
# Copyright 2002, Carnegie Mellon University
#
# Modified by Alicia Strupp, August 2004
# Purpose:  To simulate a software development organization in
#           order to observe Brooks's Law
# #############################################################

mod_states: type is enum(unassigned, assigned, completed);
prj_states: type is enum(unallocated, allocated, mods_completed, closed);
dev_states: type is enum(occupied, free);
dev_rank: type is enum(dev, teamLead);  # enhancement - developers have a
rank

# ######################## Define Global Attributes ######################

sm: simulation type is
        projList::list := new list project;     # list of projects
        devList::list := new list developer;    # list of developers
        num_developers::int := 100;             # number of experienced
developers
        num_new_developers::int := 25;          # number of inexperienced
developers
        team_size::int := 5;              # team size
        dt::number := 1.0;
        vw::view := ?;
        experienced_rate::number := 1.2;       # experienced developer
productivity
        inexperienced_rate::number := 0.8;      # unexperienced developer
productivity
        man_day_fraction::number := 0.06;      # used for communication
overhead

# ####################### Define Project and Modules Types ###############

project: type is
        modList::list := new list module; # modules per project
        modListCompleted::list := new list module;    # modules completed
        prj_state::prj_states := unallocated;
```

```
        proj_ID::int := ?;
        prj_modules::int := 10000;          # number of function points
        hire_date::int := 222;                    # date to hire new hires

module: type is
# a project consist of multiple modules
        mod_state::mod_states := unassigned;
        mod_ID::int := 0;
        dev_time::number := 10.0;                 # for fixed dev time- .1 FP
per day
        startT::number := ?;
        endT::number := ?;
        proj::project := ?;

# ####################### Define Actors: Developer, Manager, & Client
##########

developer(s:sm): actor type is
# developers implement modules
        devID::int := ?;
        mod::module := ?;      # modules currently developing
        num_mods::int := 0;    # number of modules assigned
        devT::number := 0.0;   # development time
        modList::list := new list module;              # List of modules
waiting to be completed
        modList1::list := new list module;             # List of modules
completed
        dev_state::dev_states := free;
        communication_overhead::number :=0.0;
        productivity_rate::number := ?;
        tm::number := 0.0;
        team_ID::int := ?;
        rank::dev_rank := dev;

        # Complete modules waiting to be completed
        for every true do
                if (length modList) > 0 & dev_state = free then   # if free
and work needs to be done - start developing
                        dev_state := occupied;
                        mod := pop modList;             # get module
                        push(modList1, mod);
                        mod.startT := devT;
                        # determine overhead involved
                        if (rem(s.num_developers, s.team_size) = 0) then
                                communication_overhead := (s.man_day_fraction
*(s.team_size)*(s.team_size));
                        else
                                communication_overhead :=
((s.man_day_fraction  *(s.team_size)*(s.team_size))*(floor
(s.num_developers/s.team_size))+((rem (s.num_developers, s.team_size))*(rem
(s.num_developers, s.team_size))*s.man_day_fraction ))/((floor
(s.num_developers/s.team_size))+1);
                        mod.endT :=
devT+mod.dev_time/((productivity_rate)*(1-communication_overhead/100));
                        devT := max(mod.endT, tm);
                        num_mods:=num_mods-1;
```

```
                        outln("mod.dev_time: ", mod.dev_time, "new: ",
mod.dev_time/((productivity_rate)*(1-communication_overhead/100)));
                        outln("dev ID: ", devID, "   mod ID: ", mod.mod_ID, "
productivity rate: ", productivity_rate);
                        outln("start time: ", mod.startT, " end time:
",mod.endT, "   mods remaining ", num_mods);

outln("-------------------------------------------------------------");
                        # create module
                        wait mod.dev_time/((productivity_rate)*(1-
communication_overhead/100));
                        mod.mod_state:= completed;
                        dev_state := free;
                else
                        tm:= tm+s.dt;
                        wait s.dt;

manager(s:sm): actor type is
# managers assign modules
        prj::project := ?;
        hire::int := 0;
        prj_size::int := ?;
        # Assign modules and check status of modules
        for every true do
                for prj: every s.projList do
                        if prj.prj_state != closed then  # if a project is
not complete, check status
                                if prj.prj_state = unallocated then
                                        assign_mods_to_developers(s, prj);
                                        prj.prj_state := allocated;
                                else if prj.prj_state = allocated &
all_mods_completed(s, prj) then
                                        prj.prj_state:= closed; # was
mods_completed
                                        outln("all mods completed for project
", prj.proj_ID, " Time: ", s.skdr.clock);

outln("-------------------------------------------------------------");

                wait s.dt;
                prj_size := length(s.projList);
                prj1:project := s.projList[0]; # moved from loop below
                # Hire more people if needed
                if (prj_size!=0) then
                        if (s.skdr.clock > prj1.hire_date & hire = 0 &
s.num_new_developers!= 0) then
                                hire := 1;
                                new_developer_training(s, prj1);

client(s:sm): actor type is
# Responsible for creating modules (requirements)
        i::int := 0;
        p::project := new project;
        p.proj_ID := i;
        push(s.projList, p);
        for j:each 1..p.prj_modules do  # create the modules (requirements)
                m::module := new module;
```

```
            m.mod_ID := j;
            m.proj := p;
            push(p.modList, m);

# ####################### Procedure for adding new developers #############

addDevelopers(s:sm, p:project, numOfnewbies:int): action is
# add specified number of developers to the project
        total_mods::int := 0;
        new_id::int :=length(s.devList);
        mod_list_len:int := 0;
        mod::module:=?;
        newDevList::list := new list developer;
        s.num_developers := s.num_developers + s.num_new_developers;
        for j:each 1..numOfnewbies do  # create each new developer
                d:developer := new (s, developer(s));
                new_id  := new_id + 1;
                d.devID := new_id;
                d.devT := p.hire_date; # for now, time to bring in recruits
                d.productivity_rate := s.inexperienced_rate;
                d.communication_overhead := (s.man_day_fraction
*(s.team_size)*(s.team_size));
                push(s.devList, d);
                push(newDevList, d);

# ####################### Procedure for reassigning modules ###############

reassignModules(s:sm, p:project): action is
        newModList::list := new list module;
        for d: every s.devList do               # get modules not completed
                for m: every d.modList do
                        push(newModList, m);
                d.num_mods :=0;
                emptyModList:: list := new list module;
                d.modList := emptyModList;
        p.modList := newModList;
        assign_mods_to_developers(s,p);   # send uncompleted modules to be
dispersed

# ####################### Procedure for training of new developers #########
new_developer_training(s:sm, p:project): action is
# training of new developers and mentoring of new project team leaders
        trainerList::list := new list developer;
        mentorList::list := new list developer;
        id::number := ?;
        found::boolean := true;
        num_of_mentors::int := ceil((s.num_new_developers / s.team_size));
        counter::int := 0;
        training_percent::number := 0.25;

        # Get a trainers
        for dev:each s.devList do
               push (trainerList, dev);
               dev.productivity_rate:=s.experienced_rate -
(s.num_new_developers/(s.num_developers)*training_percent*s.experienced_rate
);
```

```
        # Get Team Leads for Mentoring
        for dev:each s.devList do
                if ((counter < num_of_mentors) & (dev.rank == teamLead)) then
                        dev.productivity_rate := dev.productivity_rate -
(0.24);
                        push(mentorList, dev);
                        counter := counter + 1;

        addDevelopers(s, p, s.num_new_developers);
        reassignModules(s, p);
        assimilation (s, trainerList, mentorList);

assimilation(s:sm, trainerList:list, mentorList:list): action is
# training period of new developers
        rate::number := 20;
        last_rate::number := 0.0;
        assim::boolean := true;
        percent::number := 0.0;
        min::number := 0.00001;
        days::int := 0;
        training_prod::number := (s.num_new_developers/(s.num_developers-
s.num_new_developers)*0.25*s.experienced_rate );
        for every assim do
                rate := rate-(rate/20);
                percent := (1-rate/20)-(last_rate);
                for d: every s.devList do
                        if (d.devID > (s.num_developers -
s.num_new_developers) & d.devID < s.num_developers+1) then
                                d.productivity_rate:=d.productivity_rate +
((s.experienced_rate-s.inexperienced_rate)*percent);
                last_rate := last_rate + percent;
                if rate < min then
                        assim:= false;
                for t: every trainerList do
                        t.productivity_rate:=t.productivity_rate +
(training_prod*percent);
                # mentoring only for 20 days
                for m: every mentorList do
                        if (days < 20) then
                                m.productivity_rate := m.productivity_rate +
(0.012);
                days := days + 1;
                wait 1.0;


# ####################### Procedure for checking project status ############

all_mods_completed(s:sm, prj: project): boolean is
        for mod: every prj.modList do  # check to see if modules have been
completed
                if mod.mod_state != completed then return false;
        for d: every s.devList do      # check to see if all developers are
free
                if d.dev_state != free | (length d.modList) > 0 then return
false;
        return true;
```

```
# ###################### Procedure for assigning modules #############

assign_mods_to_developers(s:sm, prj:project): action is
        min_mods::int := ?;
        devList1::list := new list developer;
        d::developer := ?;
        i::int := ?; j::int := ?;
        for mod: every prj.modList do
                for dev: every s.devList do
                        if dev.num_mods < ((length
prj.modList)/s.num_developers) then
                                d:= dev;
                d.num_mods:=d.num_mods+1;
                push(d.modList, mod);
                mod.mod_state := assigned;

# ###################### Procedure for assigning new modules #############

assign_newmods_to_developers(s:sm, prj:project): action is
        min_mods::int := ?;
        devList1::list := new list developer;
        d::developer := ?;
        i::int := ?; j::int := ?;
        for mod:each 1..prj.prj_creepPerMonth do
                m::module := new module;
                prj.prj_modules:= prj.prj_modules +1;
                m.mod_ID := prj.prj_modules;
                m.proj := prj;
                push(prj.modList, m);
                min_mods:=1000;

                for dev: every s.devList do
                        if dev.num_mods < min_mods then
                                min_mods := dev.num_mods;
                                d:= dev;
                i:= 0;
                devList1 := list'[d];
                d := pop devList1;
                for dev1: every s.devList do
                        if dev1.num_mods = min_mods then
                                push(devList1, dev1);
                                i:= i+1;
                j:= rand(1,i);
                d:= devList1[j-1];
                d.num_mods:=d.num_mods+1;
                push(d.modList, m);
                m.mod_state := assigned;

# ###################### Procedures for graphical view ####################

graph1(s:sm): action is
        # Development Time vs. Effort
        y0::number := 100.0;
        dy::number := 0.0;
        x0::number := 100.0;
        dt::number :=1.0;
        first_flag::boolean := true;
```

```
        working::boolean := true;
        productivity::number := 0.0;
        prj::project := s.projList[0];
        wait 5.0;
        for every working do
                productivity := 0.0;
                for dev: every s.devList do
                        productivity := productivity + (dev.productivity_rate
- dev.productivity_rate*(dev.communication_overhead/100));
                if first_flag then y0:= 100 + (200- productivity);
                first_flag := false;
                depict(s.vw, paint(polyline(2.0, x0, y0 , x0 + dt, 100 + (200
- productivity)), (blue)));
                x0 := x0 + dt;
                y0 := 100 + (200 - productivity);
                wait 1.0;

                if prj.prj_state = closed then
                        working := false;
                        depict(s.vw, paint(polyline(2.0, x0, y0 , x0, 300),
(blue)));
                dy := productivity;

displayGraphs(): actor type is
        sim.vw := new view(sim, "*** SW process schedule ***", ivory, nil);
        null make_window(sim.vw, 1);
        displayEffort();

displayEffort(): action is
        # x axis
        dept1: cee  := paint(polyline(3.0, 100, 300, 1000, 300), (black));
        depict(sim.vw, dept1);
        # y axis
        dept2: cee := paint(polyline(3.0, 100, 300, 100, 50), (black));
        depict(sim.vw, dept2);
        # horizontal lines
        dept3: cee := paint(polyline(1.0, 100, 200, 1000, 200), (gray));
        depict(sim.vw, dept3);
        dept4: cee := paint(polyline(1.0, 100, 100, 1000, 100), (gray));
        depict(sim.vw, dept4);
        # vertical lines
        dept5: cee := paint(polyline(3.0, 300, 300, 300, 50). (gray));
        depict(sim.vw, dept5);
        dept6: cee := paint(polyline(3.0, 400, 300, 400, 50), (gray));
        depict(sim.vw, dept6);
        dept7: cee := paint(polyline(3.0, 500, 300, 500, 50), (gray));
        depict(sim.vw, dept7);
        dept8: cee := paint(polyline(3.0, 600, 300, 600, 50), (gray));
        depict(sim.vw, dept8);
        dept9: cee := paint(polyline(3.0, 200, 300, 200, 50), (gray));
        depict(sim.vw, dept9);
        dept10: cee := paint(polyline(3.0, 700, 300, 700, 50), (gray));
        depict(sim.vw, dept10);
        dept11: cee := paint(polyline(3.0, 800, 300, 800, 50), (gray));
        depict(sim.vw, dept11);
        dept12: cee := paint(polyline(3.0, 900, 300, 900, 50), (gray));
        depict(sim.vw, dept12);
```

```
        graph1(sim);

# ####################### Procedure for beginning simulation ##############

SEprocess(): action is
# initializes new manager, developers, projects and associated modules
        s::sm := new sm;
        num_of_teams :int := (s.num_developers/s.team_size);
        counter::int := 1;
        team::int := 1;
        # create developers
        for i:each 1.. s.num_developers do
                d:developer := new (s, developer(s));
                d.devID := i;
                d.team_ID := team;
                if (counter == 1) then d.rank := teamLead;  # assign team
leads
                if (counter == s.team_size) then
                        counter := 1;
                        team := team + 1;
                else
                        counter := counter + 1;
                d.productivity_rate := s.experienced_rate ;
                push(s.devList, d);
        null new (s, client(s));
        null new(s, manager(s));
        null new(s, displayGraphs());  # used to display graph output
        wait s;

# ####################### Call to start simulation#########################

SEprocess();
```

APPENDIX C

EASEL SMALL PROJECT RESULTS

Remaining Easel simulation results for the small project trials are provided below.

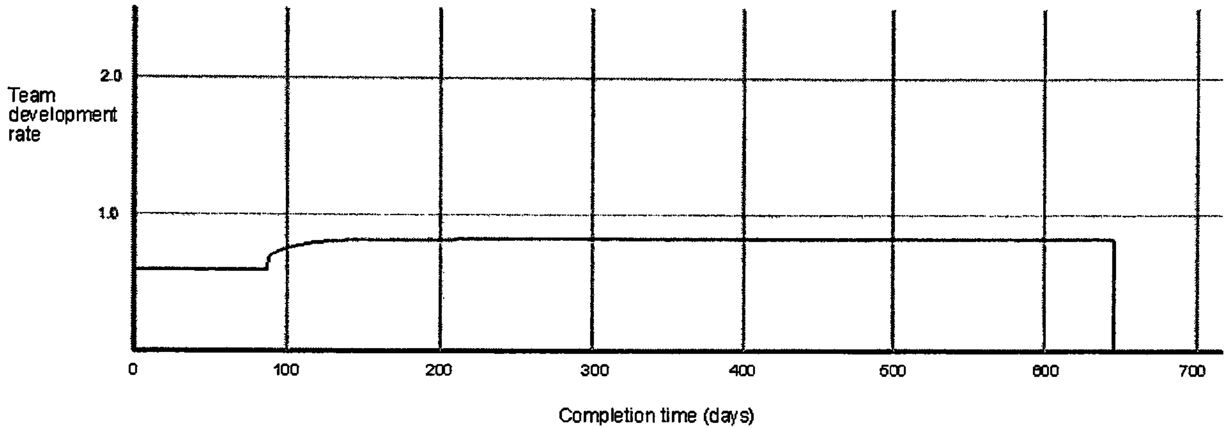


Figure 48 - Easel - Hiring 2 New Developers at Day 90

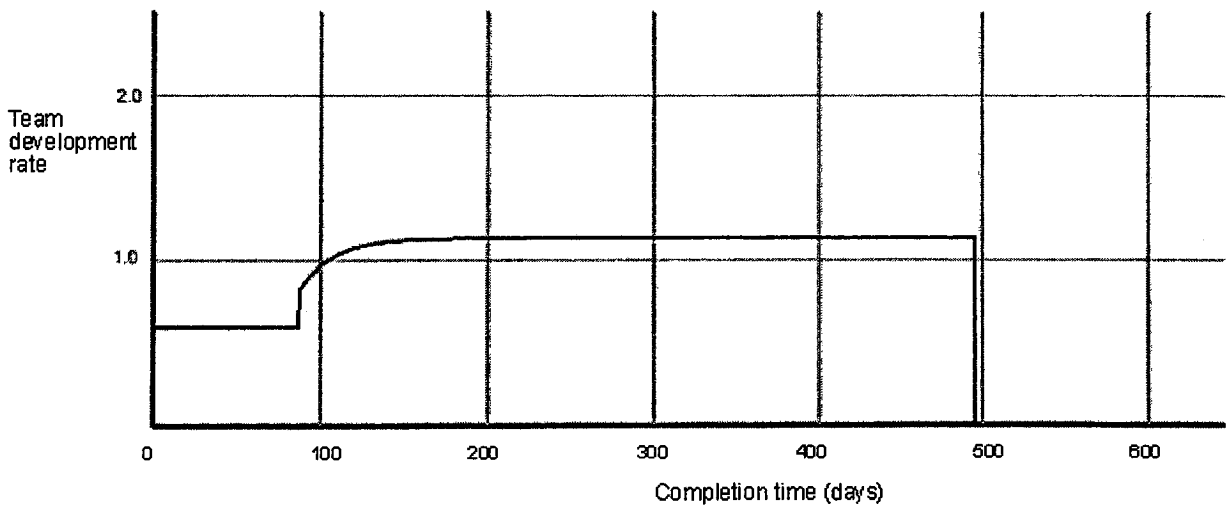


Figure 49 - Easel - Hiring 7 New Developers at Day 90

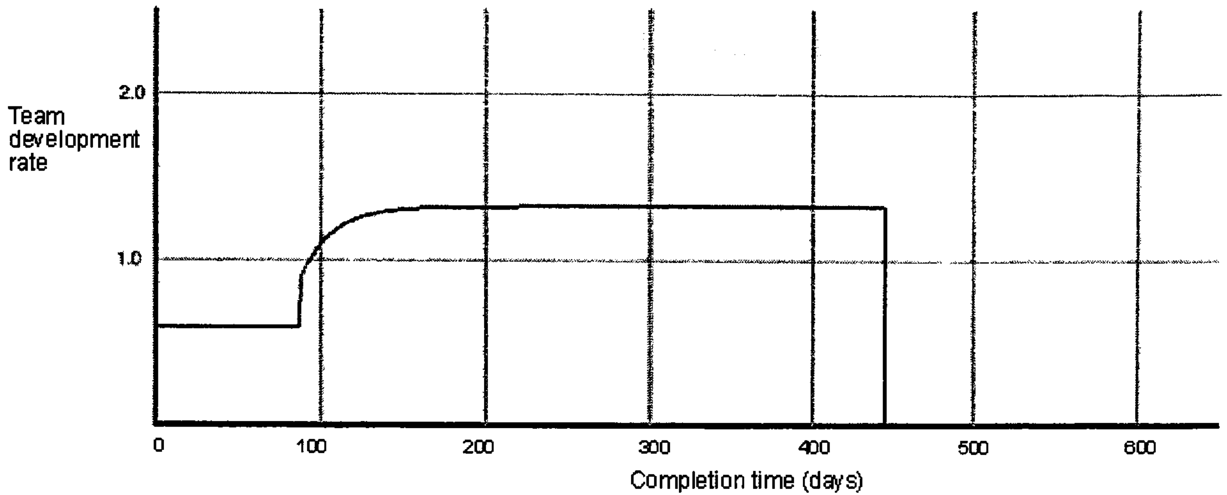


Figure 50 - Easel - Hiring 7 New Developers at Day 90

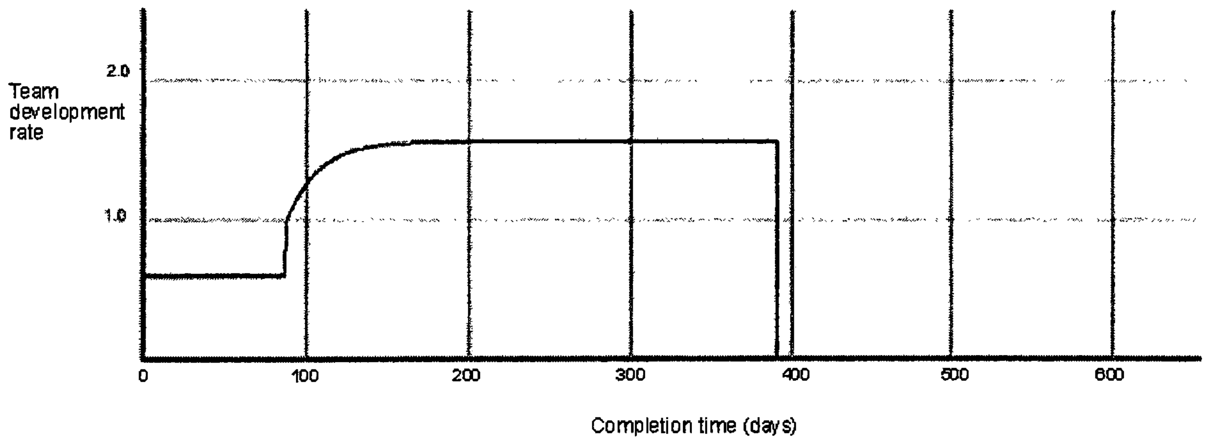


Figure 51 - Easel - Hiring 10 New Developers at Day 90

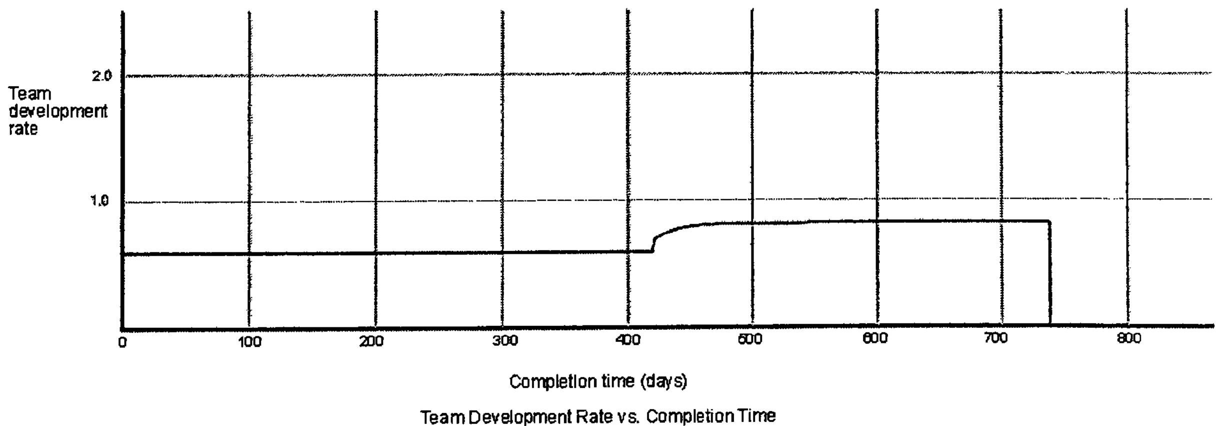


Figure 52 - Easel – Hiring 2 New Developers at Day 423

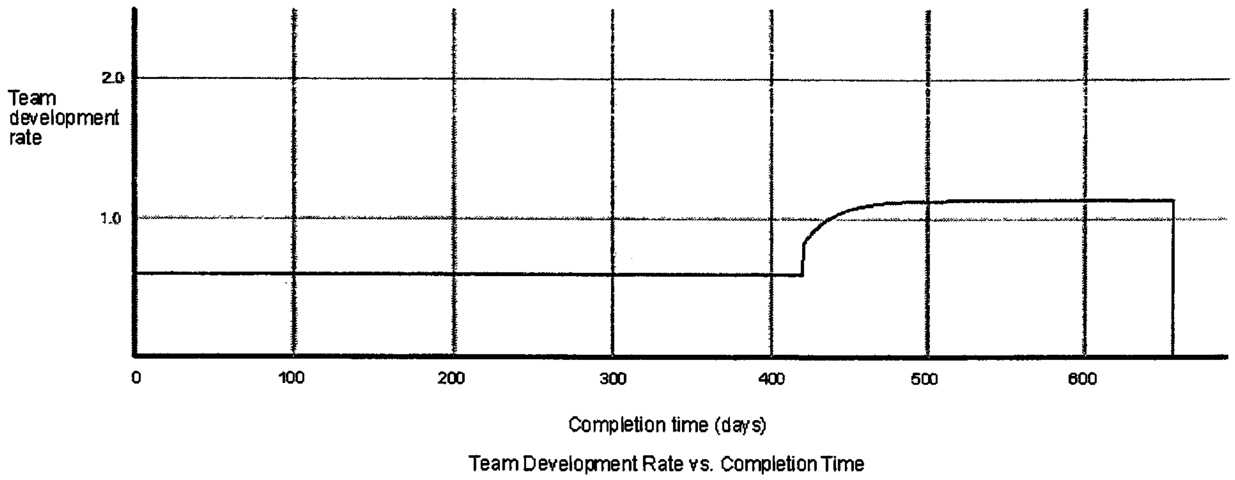


Figure 53 - Easel - Hiring 5 New Developers at Day 423

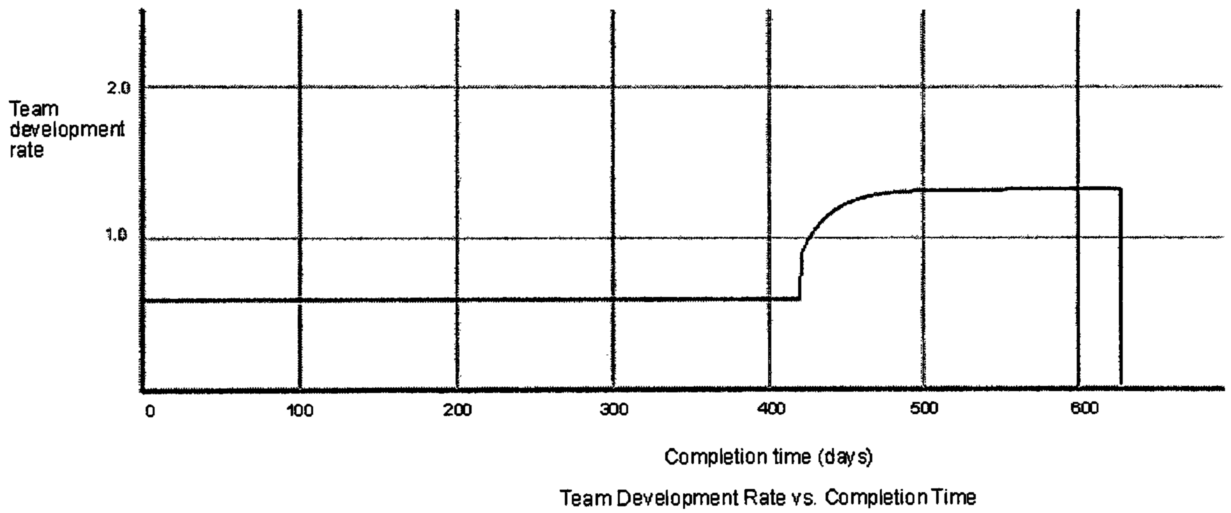


Figure 54 - Easel - Hiring 7 New Developers at Day 423

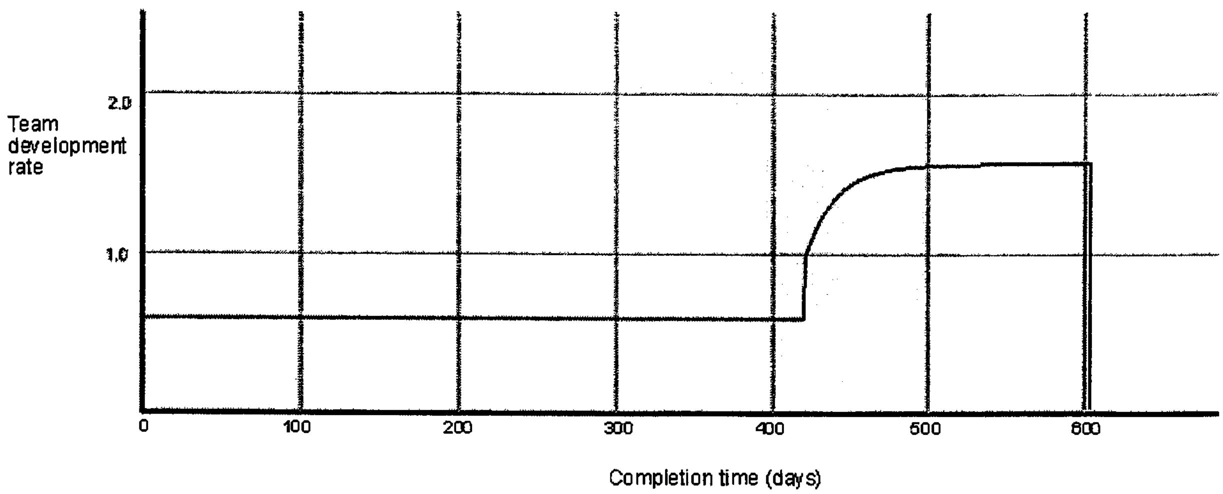


Figure 55 - Easel - Hiring 10 New Developers at Day 423

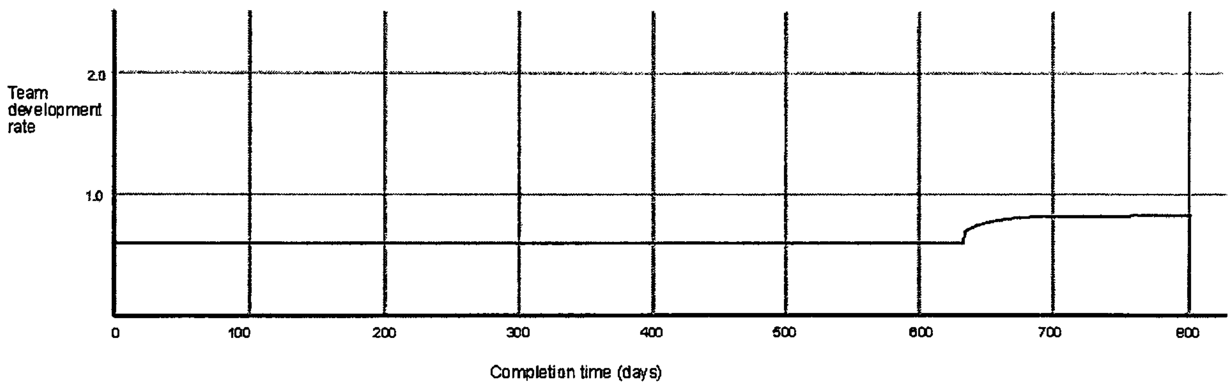


Figure 56 - Easel - Hiring 2 New Developers at Day 635

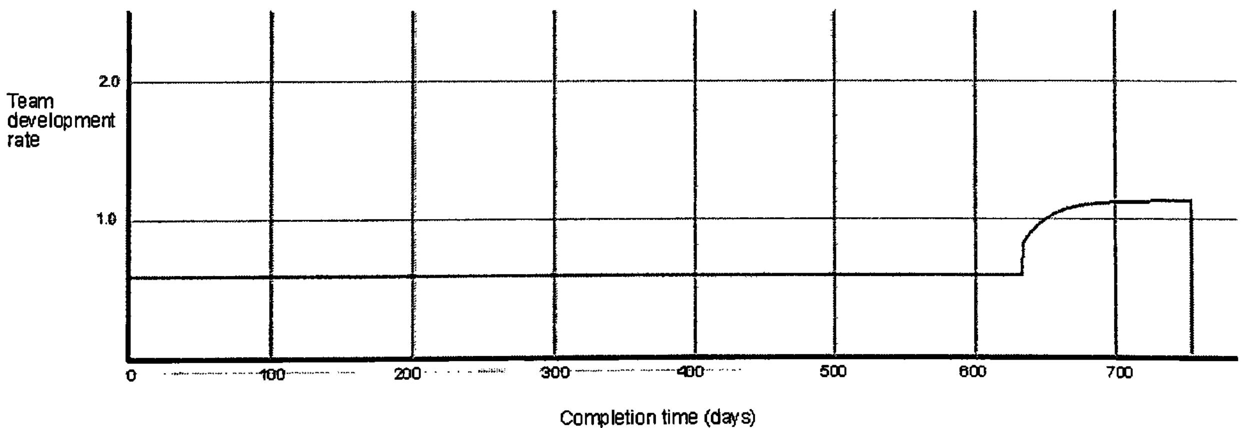


Figure 57 - Easel - Hiring 5 New Developers at Day 635

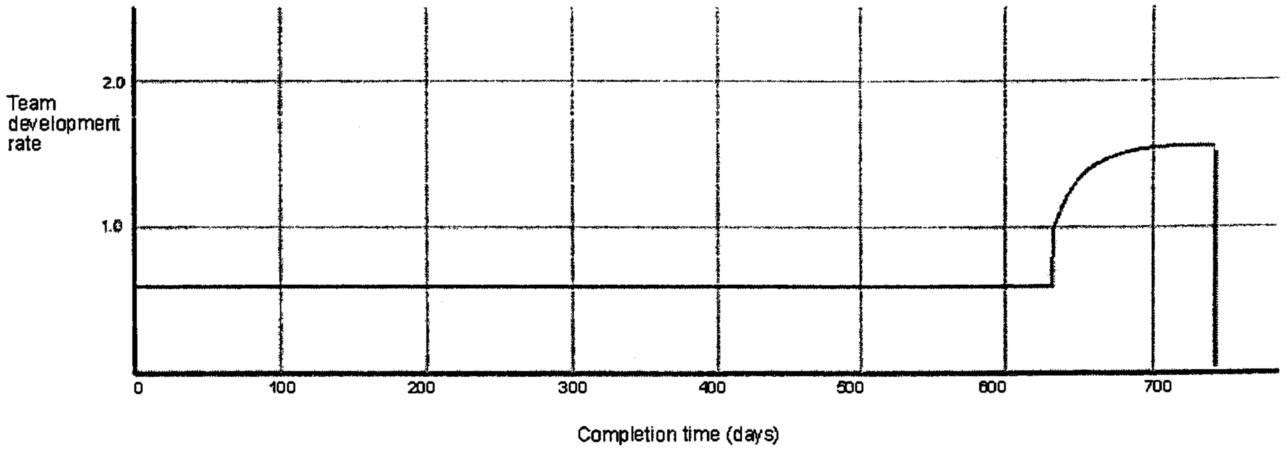


Figure 58 - Easel - Hiring 7 New Developers at Day 635

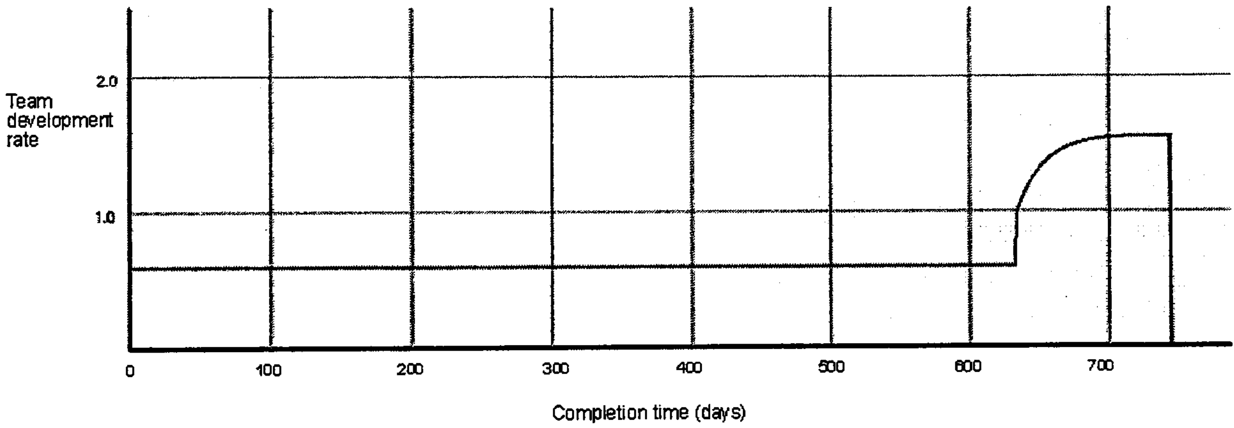


Figure 59 - Easel - Hiring 10 New Developers at Day 635

APPENDIX D

EASEL MEDIUM PROJECT RESULTS

Remaining Easel simulation results for the medium project trials are provided below.

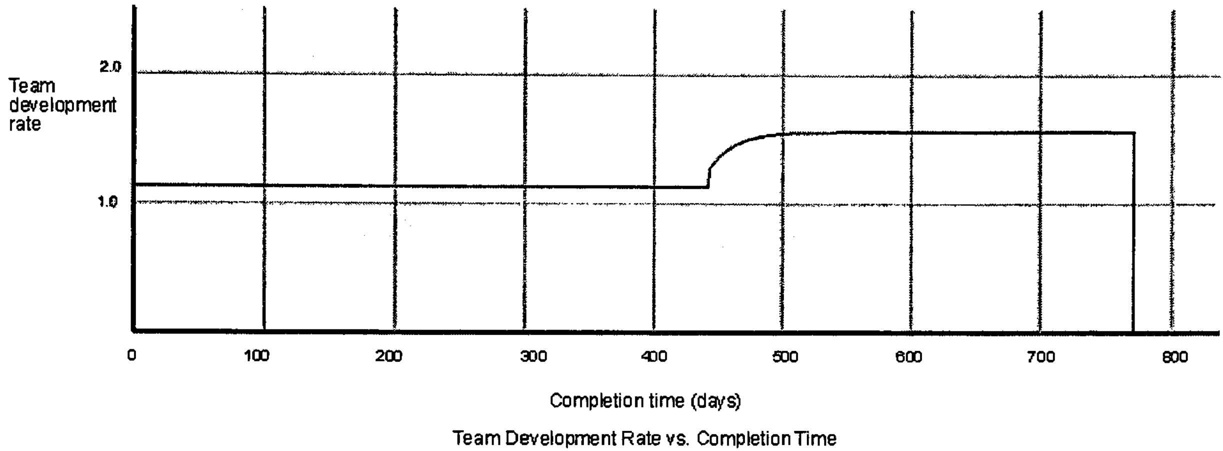


Figure 60- Easel - Hiring 5 New Developers at Day 445

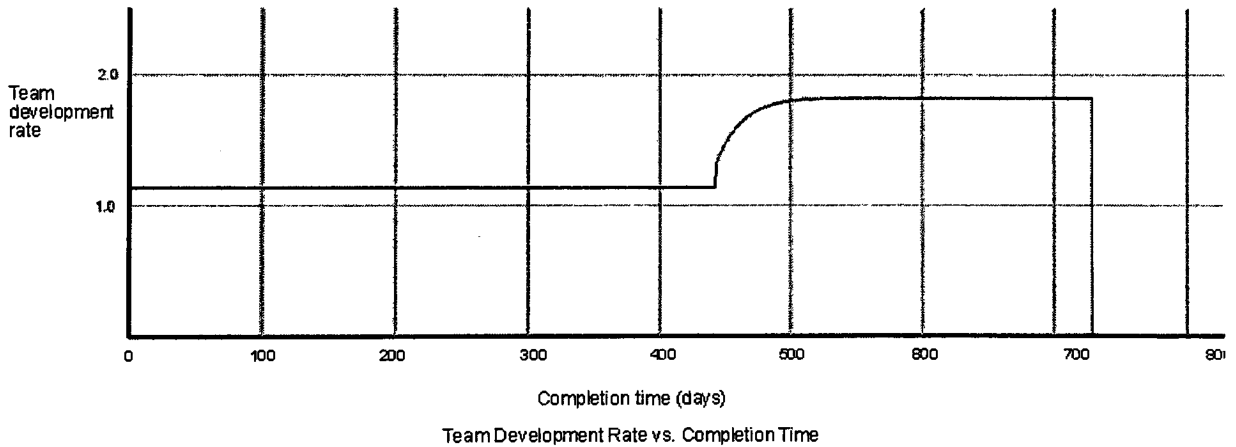


Figure 61 - Easel - Hiring 10 New Developers at Day 445

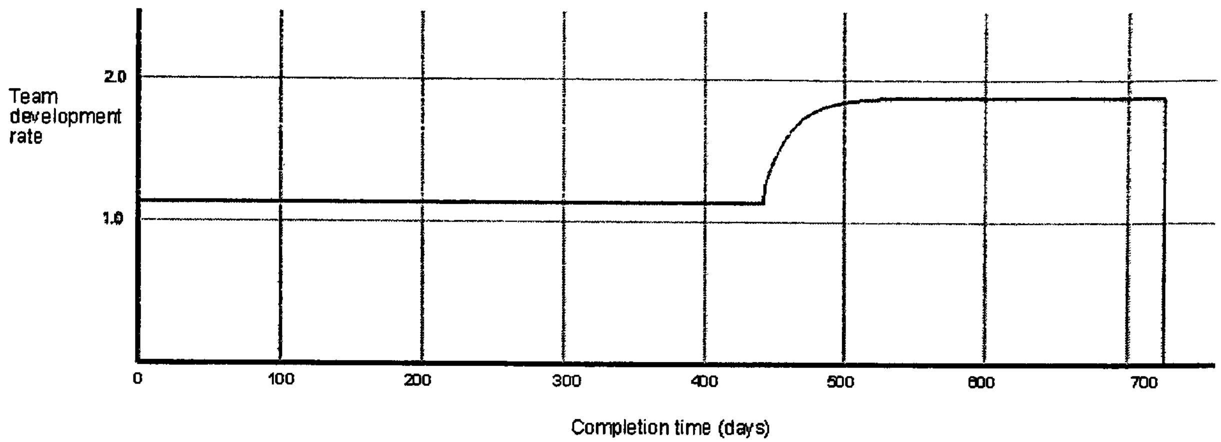


Figure 62 - Easel - Hiring 15 New Developers at Day 445

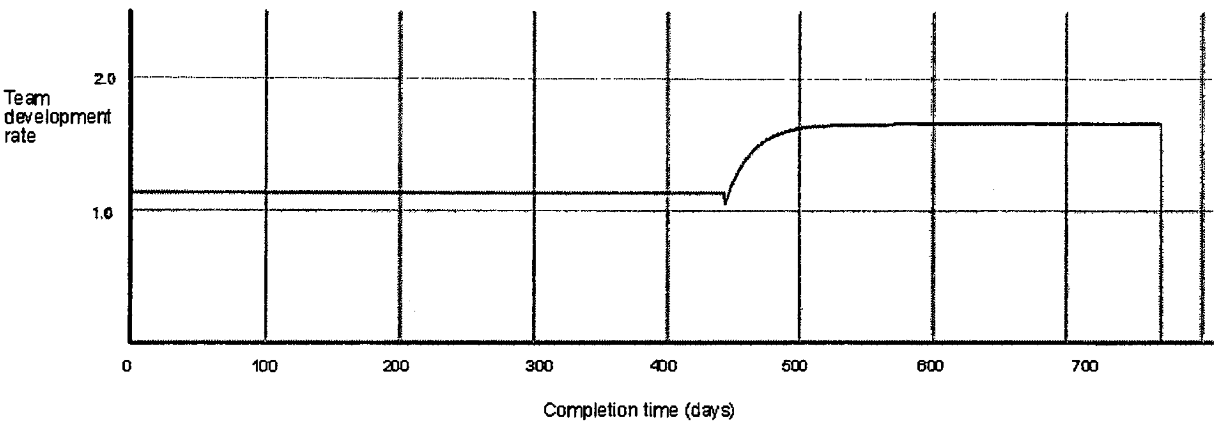


Figure 63 - Easel - Hiring 20 New Developers at Day 445

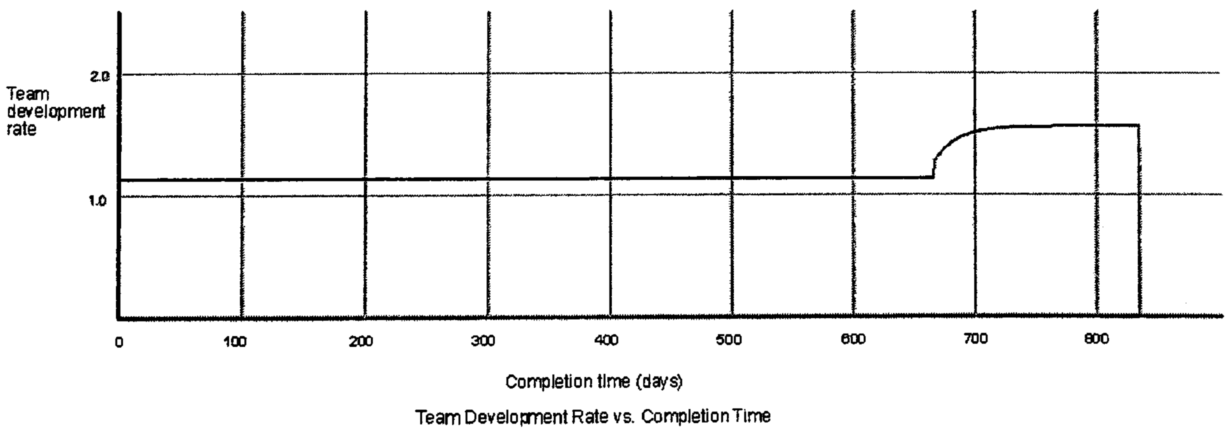


Figure 64 - Easel - Hiring 5 New Developers at Day 668

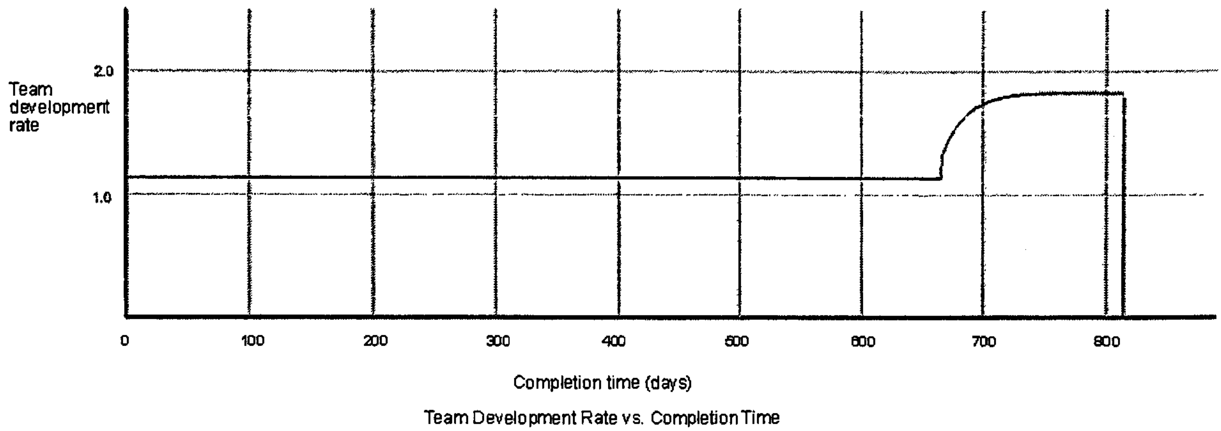


Figure 65 - Easel - Hiring 10 New Developers at Day 668

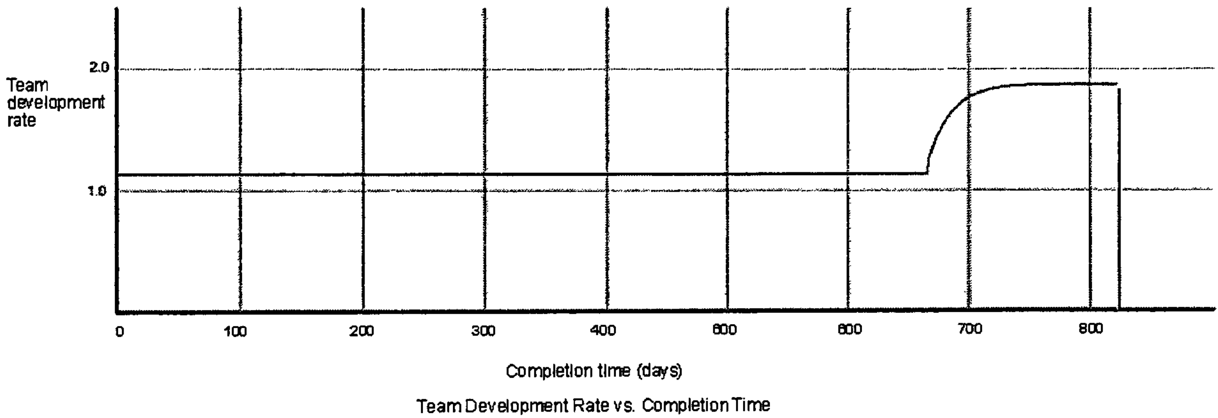


Figure 66 - Easel - Hiring 15 New Developers at Day 668

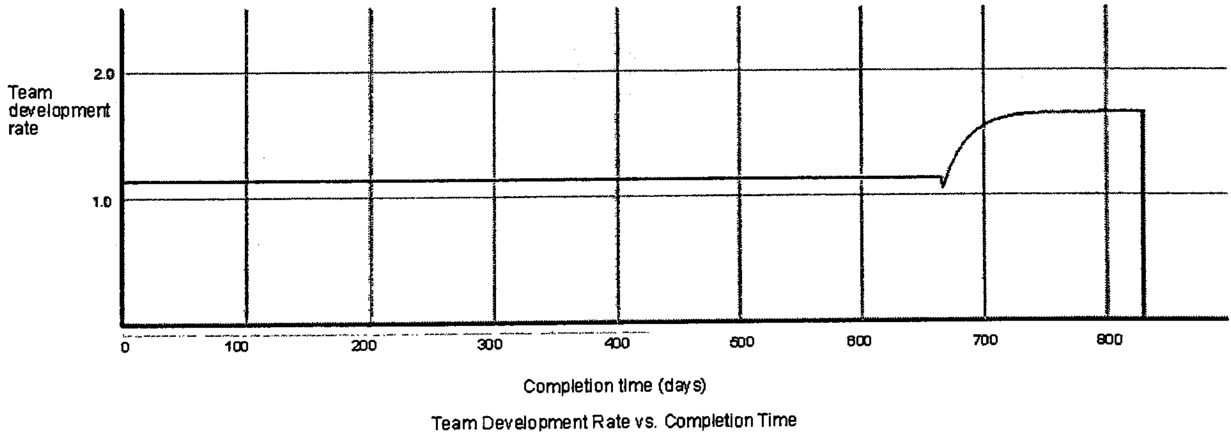


Figure 67 - Easel - Hiring 20 New Developers at Day 668

BIBLIOGRAPHY

Abbot, M., & Stanley, R. (1999). Modeling groundwater recharge and flow in an upland fractured bedrock aquifer. *System Dynamics Review 15*(2), 163-182.

Abdel-Hamid, T.K. (1984). *The dynamics of software development project management: An integrative system dynamics perspective.* Unpublished doctoral dissertation. MIT Sloan School of Management.

Abdel-Hamid, T.K. (2002). Modeling the dynamics of human energy regulation and its implications for obesity treatment. *System Dynamics Review, 18*(4), 431.

Abdel-Hamid, T.K., & Madnick, S. (1983). The dynamics of software project scheduling. *Communications of the ACM, 26*(5), 340-346.

Abdel-Hamid, T.K., & Madnick, S. (1989). Lessons learned from modeling the dynamics of software development. *Communications of the ACM*, 32(12).

Abdel-Hamid, T.K., & Madnick, S. (1990). The elusive silver lining: How we fail to learn from software development failures. *MIT Sloan Management Review, 32*(1), 39-48

Abdel-Hamid, T.K., & Madnick, S. (1991). *Software project dynamics: An integrated approach.* Englewood Cliffs, NJ: Prentice-Hall.

Agha, G. (1986). An overview of actor languages. *ACM SIGPLAN Notices, 2* (10).

Agha, G. (1986). *Actors: A model of concurrent computation in distributed systems.* Cambridge, Mass: MIT Press.

Agha, G. (1988). Foundational issues in concurrent computing. *ACM SIGPLAN Notices*, 24(14).

Akkermans, H.A. (2001). Emergent supply networks: System dynamics simulation of adaptive supply agents. *Proceedings of the 43th Hawaii International Conference on System Sciences (HICSS-34)*, University of Hawaii, January 2001.

Akkermans, H.A., Bogerd, P., Vos, B. (1999). Virtuous and vicious cycles on the road towards international supply chain management. *International Journal of Operations and Production Management*, 19(5), 565-581.

Ambler, S. (1999). Enhancing the unified process. *SDMagazine*. October 1999. Retrieved November 11, 2003 from http://www.sdmagazine.com/documents/s=753/sdm9910b/9910b.htm

Angerhofer, B.J., & Angelides, M.C. (2000). System dynamics modeling in supply chain management: Research review. *Proceedings of the 2000 Winter Simulation Conference*, 342–351.

Belani, R. V., Das, S. M., & Fisher, D. A. (2002). One-to-one modeling and simulation of unbounded systems: Experiences and lessons. *Proceedings of the 2002 Winter Simulation Conference*, San Diego, CA.

Booch, G. (1993). *Object-oriented analysis and design with applications (2nd ed.)*. Addison-Wesley.

Brock, S., Hendricks, D., Linnell, S., & Smith, D. (2003). A balanced approach to IT project management. *Proceedings of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology*, 2-10. Retrieved February 14, 2003, from the ACM Digital Library.

Brooks, F.P. Jr. (1995). The mythical man-month. MA: Addison-Wesley.

Burke, S. (1997). *Radical improvements require radical actions: Simulating a high-*

maturity software organization. Retrieved September 28, 2003, from http://www.sei.cmu.edu/pub/documents/96.reports/pdf/tr024.96.pdf

Bustard, D., Kawalek, P., & Norris, M. (2000). *Systems modeling for business process improvement.* Artech House.

Cano, J. (2003). *Critical reflections on information systems: A systemic approach.* Idea Group Publishing.

Carr, D. (1990). System dynamics models of software developments. *IEEE Computer Society Digital Library.* Retrieved September 3, 2003 from the Communications of the ACM database.

Caulfield, C.W., & Maj, S.P. (2002). A case for system dynamics. *Global Journal of Engineering Education*, 6(1), 25-34.

Christie, A. (1999). *Simulation—An enabling technology in software engineering.* Retrieved September 1, 2003 from http://www.sei.cmu.edu/publications/articles/christie-apr1999/christie-apr1999.html

Christie, A. (2002). *Software development simulation.* Retrieved September 27, 2003, from http://www.cert.org/easel/demos/SWprocess.html

Christie, A.M., & Fisher, D.A. (2000). *Simulating the emergent behavior of complex software-intensive organizations.* Retrieved March 26, 2003, from http://www.cert.org/easel/prosim.pdf

Christie, A., Durkee, D., Fisher, D.A., & Mundie, D.A. (2003). *Easel language reference manual and author's guide.* Retrieved March 25, 2003, from http://www.cert.org/easel

Coyle, R.G. (1977). *Management system dynamics*. New York: Wiley.

Dangerfield, B., & Roberts, C. (1999). Foreword to the special issue on health and health care dynamics. *System Dynamics Review, 15* (3), 197.

Dangerfield, B., & Roberts, C. (1999). Optimisation as a statistical estimation tool: An example in estimating the AIDS treatment-free incubation period distribution. *System Dynamics Review, 15*(3). 273-287.

DeSantis, M. (2001). *Using Easel to study complex systems.* Retrieved January 1, 2004 from http://interactive.sei.cmu.edu/news@sei/features/2001/3q01/pdf/feature-4-3q01.pdf

Dill, M. (1997). *Capital investment cycles: A system dynamics modeling approach to social theory development.* 15th International System Dynamics Conference, Systems Approach to Learning and Education into the 21st Century, Istanbul, Turkey.

Edwards, P. (1997). *The world in a machine: Origins and impacts of early computerized global system and models.* Retrieved December 2, 2003 from http://www.si.umich.edu/~pne/modeling.world.htm

Fisher, D. A. (1999). *Design and implementation of EASEL: A language for simulating highly distributed systems.* Retrieved March 25, 2003, from http://www.cert.org/archive/pdf/design-easel.pdf

Fisher, D. A. (1991). *What is Informalism?* Retrieved January 3, 2004 from http://www.cert.org/easel/Fisher.ppt

Fisher, D. A. (2000). *Easel survivability simulation system.* Retrieved January 3, 2004 from http://www.cert.org/easel/iMPACTS2000.ppt

Fisher, D. A., & Christie, A. M. (2000). *Easel—A new simulation language and its application to software process.* Retrieved January 2, 2004 from http://www.cert.org/easel/ProSim_Presentation.ppt

Forrester, J.W. (1958). Industrial dynamics: A major breakthrough for decision makers. *Harvard Business Review*, 36(4), 37-66.

Forrester, J.W. (1962). *Industrial dynamics*. Cambridge, MA: Productivity Press.

Forrester, J.W. (1969). *Urban dynamics*. Cambridge, MA: Productivity Press.

Forrester, J.W. (1971). *World dynamics*. Cambridge, MA: Productivity Press.

Forrester, J.W. (1995). The beginning of system dynamics. *The McKinsey Quarterly, 1995* (4), 6 -16.

Forrester, J.W., Legasto, A., & Lyneis, J. (1980). *System dynamics*. Amsterdam, NY: North-Holland Pub. Co.

Frolund, S. (1996). *Coordinating distributed objects: An actor-based approach to synchronization*. Cambridge, Mass: MIT Press.

Gharajedaghi, J. (1999). *Systems thinking : Managing chaos and complexity : A platform for designing business architecture*. Boston: Butterworth–Heinemann.

Haines, S. (2000). *The systems thinking approach to strategic planning and management*. Boca Raton, FA: St. Lucie Press.

Herbsleb, J., Carleton, A., Rozum, J., Siegel, J., & Zubrow, D. (1994). *Benefits of CMM-based process improvement: Initial results*. Retrieved June 4, 2004 from http://www.sei.cmu.edu/publications/documents/94.reports/94tr013/94tr013title.html

Hewitt, C. E. (1977). *Viewing control structures as patterns of passing messages*. Retrieved January 1, 2004 from ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-410.pdf.

Houston, D. (1996) *System dynamics modeling and simulation of software development: A tutorial*. Retrieved July 18, 2003 from http://www.eas.asu.edu/~sdm/papers.html

Hsia, P, Hsu, C., & Kung, D.C. (1999). Brooks' law revisited: A system dynamics approach. *Twenty-Third Annual International Computer Software and Applications Conference*, Phoenix, Arizona.

Huber, N. (2003). *Hitting targets: The state of UK IT project management.* Accessed February 17, 2002, from http://www.computerweekly.com/articles/article.asp?liArticleID=126138&liFlavourID=1

Jones, C. (1998) *Estimating software costs.* New York: McGraw-Hill.

Kahen, G., Lehman, M., & Ramil, J.F. (2000). *System dynamics modelling for the management of long term software evolution processes.* Retrieved December 12, 2003, from http://www.doc.ic.ac.uk/research/technicalreports/2000/DTR00-16.pdf

Kellner, M.I., Madachy, R. J., & Raffo., D.M. (1999). Software process modeling and simulation: Why? What? How? *Journal of Systems and Software, 46* (2).

King, J. & Diaz, M. (2002). *How CMM impacts quality, productivity, rework, and the bottom line.* Retrieved July 10, 2004, from http://www.stsc.hill.af.mil/crosstalk/2002/03/diaz.html

Kirkwood, C. (1998). *System dynamics methods: A quick introduction.* Retrieved October 6, 2003, from http://www.public.asu.edu/~kirkwood/sysdyn/SDIntro/SDIntro.htm.

Legasto, A., Forrester, J.W., & Lyneis, J.M. (1980). *System dynamics.* New York: North-Holland Pub. Co.

Lyneis, J. (2000). System dynamics for marketing forecasting and structural analysis. *System Dynamics Review, 16*(1), 3.

Lyneis, J., Cooper, K., & Els, S. (2001). Strategic management of complex projects: A case study using system dynamics. *System Dynamics Review, 17*(3), 237.

Madachy, R. J. (2003). Software process dynamics. *IEEE Computer Society Press.* Retrieved August 3, 2003, from http://www-rcf.usc.edu/~madachy/spd/

Madachy, R. J. (1996). System dynamics modeling of an inspection-based process, *International Conference on Software Engineering, Proceedings of the 18th International Conference on Software Engineering,* (376-386).

McConnell, S. (1996). *Rapid development: Taming wild software schedules.* Microsoft Press.

McCray, G., & Clark Jr., T. (1999). Using system dynamic to anticipate the organizational impacts of outsourcing. *System Dynamics Review, 15*(4), 345.

Meadows, D., & Club of Rome. (1972). *The limits to growth: A report for the Club of Rome's project on the predicament of mankind.* Universe Books.

Morecroft, J. D. W. (1983). System dynamics: Portraying bounded rationality. *Omega, 11*(2), 131-142.

Morecroft, J. D. W. (1984). Strategy support models. *Strategic Management Journal, 5* (3).

Paulk, M. C. (1995). *The rational planning of (software) projects.* Retrieved September 13, 2003, from http://www.sei.cmu.edu/cmm/Misc/rational.planning.pdf

Pearce, S. (2003). *Government IT projects.* Retrieved February 14, 2004, from http://www.parliament.uk/post/pr200.pdf

Pestel, E. (1972). *The limits to growth (Abstract).* Retrieved February 28, 2004, from http://www.clubofrome.org/docs/limits.rtf.

Powell, S., Schwaninger, M., & Trimble, C. (2001). Measurement and control of business processes. *System Dynamics Review, 17*(1), 63.

Pugh, J. (1984). Actors—The state is set. *ACM SIGPLAN Notices, 19*(3).

Raynus, J. (1999). *Software process improvement with CMM.* Artech House.

Richardson, G. P. (1996). Problems for the future of system dynamics. *System Dynamics Review 12*(2): 141-157.

Ritchie-Dunham, J. L., & Galvan, F. M. (1999). Evaluating epidemic intervention policies with systems thinking: A case study of dengue fever in Mexico. *System Dynamics Review, 15*(2), 119-137.

Roberts, N., Anderson, D., Deal, R., Garet, M., & Shaffer, W. (1983). *Introduction to computer simulation: The system dynamics approach.* Reading, Mass.: Addison–Wesley.

Roberts, E.B. (1978). *Managerial applications of system dynamics*. Cambridge, MA: Productivity Press.

Rus, I. (1997). *Modeling the impact on project cost and schedule of software engineering practices for achieving and assessing software quality factors.* Unpublished doctoral dissertation proposal, Arizona State University.

Russel, G.W. (1991) Experience with inspection in ultralarge-scale developments. *IEEE Software 8,* 25–31.

Sastry, M.A., & Sterman, J.D. (1992) *Desert island dynamics: An annotated survey of the essential system dynamics literature.* Retrieved December 2, 2003, from http://web.mit.edu/jsterman/www/DID.html

SDEP: Road maps. Retrieved August 1, 2003 from http://sysdyn.clexchange.org/road-maps/home.html

Simon, H.A. (1981). The architecture of complexity. *The Sciences of the Artificial.* MIT Press, 98-114.

Software Engineering Institute at Carnegie Mellon University. (1995). *The capability maturity model: Guidelines for improving the software process.* Addison-Wesley.

Standish Group, The. (2001). *Extreme chaos.* Retrieved February 17, 2003, from http://www.standishgroup.com/sample_research/index.php

Sterman, J.D. (1989). Deterministic chaos in an experimental economic system. *Journal of Economic Behavior and Organization*, 12, 1-28.

Sterman, J.D., & Sweeney, L.B. (2002). Cloudy skies: Assessing public understanding of global warming. *System Dynamics Review, 18*(2), 207.

Tvedt, J. D. (1996). *An extensible model for evaluating the impact of process improvements on software development cycle time.* Unpublished doctoral dissertation. Arizona State University.