View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Texas A&M Repository

PRODUCTIVITY PREDICTION MODEL BASED ON BAYESIAN ANALYSIS AND PRODUCTIVITY CONSOLE

A Dissertation

by

SEOK JUN YUN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2005

Major Subject: Computer Science

PRODUCTIVITY PREDICTION MODEL BASED ON BAYESIAN ANALYSIS AND PRODUCTIVITY CONSOLE

A Dissertation

by

SEOK JUN YUN

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Approved as to style and content by:

Dick B. Simmons (Co-Chair of Committee)	William M. Lively (Co-Chair of Committee)
S. Bart Childs (Member)	Ho-Yeong Kang (Member)
	Valerie E. Taylor (Head of Department)

May 2005

Major Subject: Computer Science

ABSTRACT

Productivity Prediction Model Based on Bayesian Analysis and Productivity Console. (May 2005)

Seok Jun Yun, B.S., Korea Military Academy;

M.S., Naval Postgraduate School

Co–Chairs of Advisory Committee: Dr. Dick B. Simmons
Dr. William M. Lively

Software project management is one of the most critical activities in modern software development projects. Without realistic and objective management, the software development process cannot be managed in an effective way. There are three general problems in project management: effort estimation is not accurate, actual status is difficult to understand, and projects are often geographically dispersed. Estimating software development effort is one of the most challenging problems in project management. Various attempts have been made to solve the problem; so far, however, it remains a complex problem. The error rate of a renowned effort estimation model can be higher than 30% of the actual productivity. Therefore, inaccurate estimation results in poor planning and defies effective control of time and budgets in project management. In this research, we have built a productivity prediction model which uses productivity data from an ongoing project to reevaluate the initial productivity estimate and provides managers a better productivity estimate for project management. The actual status of the software project is not easy to understand due to problems inherent in software project attributes. The project attributes are dispersed across the various CASE (Computer-Aided Software Engineering) tools and are difficult to measure because they are not hard material like building blocks. In this research, we have created a productivity console which incorporates an expert

system to measure project attributes objectively and provides graphical charts to visualize project status. The productivity console uses project attributes gathered in KB (Knowledge Base) of PAMPA II (Project Attributes Monitoring and Prediction Associate) that works with CASE tools and collects project attributes from the databases of the tools. The productivity console and PAMPA II work on a network, so geographically dispersed projects can be managed via the Internet without difficulty.

To my families for their love, patience and encouragement.

ACKNOWLEDGMENTS

This dissertation would not have been successful without the help and support of many people. I would like to particularly express my deep appreciation and gratitude to my advisor, Dr. Dick Simmons, for his guidance, invaluable comments, and the countless hours he spent on me during my graduate studies at Texas A&M University. This research work would not have been possible without his guidance and support. His knowledge and experience have enriched both my academic and work experience. I would like to thank research committee co-chair, Dr. William Lively for his guidance in the area of software engineering and for his interest in my research. I would also like to thank research committee member as well as graduate advisor, Dr. Bart Childs, for giving me an opportunity to extend my knowledge in computer science. Special thanks go to Dr. Ho-Yeong Kang for his friendly guidance and suggestions in the area of statistics and project management. Thanks to the Ministry of National Defense and the Army of the Republic of Korea (R.O.K.) for their economic support and for giving me this great opportunity to pursue a Ph.D. degree at Texas A&M University. I wish to thank my parents and in-laws for their continued support and encouragement. Finally, to my wife Youngah and my daughter, Chaerin, and my son, Seojin, I express sincere thanks for their assistance in helping me achieve my goals by their support, patience and sacrifice over the years.

TABLE OF CONTENTS

CHAPTER			Page
I	INTRODUCTION		1
	I.1	Motivation	1
	I.2	Research Objective	9
	I.3	Organization of the Dissertation	12
II	LITERATURE SURVEY		13
	II.1	Introduction	13
	II.2	Software Project Management	13
	II.3	Definition of Productivity	30
	II.4	Effort Estimation Model	34
	II.5	COCOMO II	47
	II.6	Commercial Off the Shelf (COTS) Components	57
III	BAYESIAN ANALYSIS		59
	III.1	Statistical Analysis	59
	III.2	Bayesian Analysis	62
	III.3	Model Assumption	66
	III.4	Multi-Parameter Estimation	66
IV	DEVELOPMENT OF PAMPA II		68
	IV.1	Overview of PAMPA II	68
	IV.2	Framework of PAMPA II	69
	IV.3	System Architecture of PAMPA II	74
	IV.4	Subsystems of PAMPA II	77
V	PRODUCTIVITY PREDICTION MODEL		83
	V.1	Model Building Based on Bayesian Analysis	83
	V.2	Productivity Console	87
	V.3	Rules and Facts	89
	V.4	Project Attributes Gathering from CASE Tools	93
	V.5	Visual Interface	98
VI	EXPERIMENTAL RESULTS		104

CHAPTER Page

VI.1 Project Description 104
VI.2 Preliminary Productivity Estimation 107
VI.3 Data Collection and Experimental Results 112

VII CONCLUSIONS AND FUTURE EXTENSION 116

VII.1 Conclusions 116
VII.2 Future Extension 118

REFERENCES 121

APPENDIX A 134

VITA 162

LIST OF TABLES

TABLE Page

1 User function types 50

2 FP complexity levels 52

3 UFP complexity weights 53

4 Default UFP to SLOC conversion ratios 54

5 Effort required for COTS based development 58

6 Subsystems of PAMPA II 77

7 Initial facts 90

8 Earned value 97

9 PAMPA II schema 1 102

10 PAMPA II schema 2 103

11 ACAP cost driver 109

12 PCAP cost driver 109

13 PCON cost driver 109

14 APEX cost driver 110

15 PLEX cost driver 110

16 LTEX cost driver 110

17 Posterior distribution on 2-27-04 114

LIST OF FIGURES

FIGURE Page

1 Uncertainty estimate 6

2 Progress curve 7

3 Project tracking and control model 14

4 Simmons' project triangle 26

5 Considering people, process, and product together 28

6 16 critical software practices for performance-based management . . . 29

7 Project management style 31

8 Current productivity model 33

9 Statistical analysis 59

10 Overview of PAMPA II system architecture 69

11 Project 70

12 Plan 71

13 Work breakdown structure 72

14 Software product 74

15 Knowledge base framework and relationship 75

16 Three-tier architecture 76

17 Outline of the system 78

18 Data transformation module 81

19 Productivity prediction model 84

FIGURE Page

20 Productivity console and PAMPA II 88
21 Expert system diagram 89
22 Forms of the IF-THEN rule 91
23 Detailed KB schema on plan 96
24 Productivity console shows a project level view 100
25 Productivity console shows a team level view 101
26 Format of the weekly status report 107
27 Productivity estimates for the project 111
28 A sample of planned activities and effort/cost 112

CHAPTER I

INTRODUCTION

> The process of controlling a software engineering project may well be the most talked about and least understood of all the project managers' functions. Lehman [73]

I.1 Motivation

A critical problem facing software development in today's competitive environment is project management. Project management is the primary key to software project success or failure. Without realistic and objective management, the software development process cannot be managed in an effective way.

A software project is a planned process of activities creating artifacts that occur within a specified time and have the goal of delivering to customers a satisfactory software product on time and within budget. Kemerer and Patrick [64], however, provided ample anecdotal evidence that in general these goals are not being met. They quoted that average budget overrun of 36% in 72 medium-scale software projects [56], and cancellation of 25% in 500 software projects due to cost overruns [37].

The problem of project cancellation and cost overrun mainly depends on unpredictable feature of the software project management. This unpredictability is the basis of what has been referred to for the past 30 years as the "software crisis" [93]. Simmons pointed out that many software projects fail because of the manager's inability to visualize what is being created in time to influence project outcome [98].

Managing and overseeing large software projects is extremely difficult. In 1989,

---

This dissertation follows the style of *IEEE Transactions on Software Engineering*.

the record showed that software development was plagued with cost overruns, late deliveries, poor reliability and user dissatisfaction [2]. Even today, software projects are still late, over budget, and unpredictable [89]. Sometimes, the entire project fails before ever delivering a software product. The Chaos study, published by the Standish Group, found that 26% of all software projects fail (down from 40% in 1997), but 46% experience cost and schedule overruns or significantly reduced functionality (up from 33% in 1997) [111]. Several attempts have been made to overcome these problems, but few have been successful [86].

Managers would like to deliver products on time, within the budget, and with few defects. However, it is impossible for them to accurately steer the project in the right direction because there are no accurate ways for the managers to measure where the product is at any given time. Managers have no decent project attributes that can tell them when the project is going astray. Thus, managers have no way to know when to initiate corrective action until it is too late [86]. For example, in 1995, the Denver airport was delayed because of the software that controls the automatic baggage system [17]. The delay caused by the software problem cost Denver $1.1 million a day in interest and operating costs. In 1996, after spending $7 billion, the U.S. Federal Aviation Administration (FAA) ended up with a project that was "out of control" [99]. In these cases, the true nature and pervasive extent of underlying software project problems remained invisible to project managers until it was too late. Because of planning, management, and visualization problems software systems cost far more to build and take much longer to construct than the office buildings occupied by the companies that have commissioned the software [57]. In addition to the previous examples, Standish illustrated [105]:

- In 1995, only 16% of software projects were expected to finish on time and on

budget

- Projects completed by the largest US organizations have only 42% of originally proposed functions
- An estimated 53% of projects will cost nearly 190% of their original estimates
- In large companies, only 9% of projects will be completed on time and on budget.
- Cancelled projects cost the US $81 billion in 1995
- Average Management Information System (MIS) are one year late, 100% over budge

Software projects have the potential to suffer from numerous problems including: missed deadlines, inaccurate budgets, unmet specifications, product defects, unforeseen project risks, changing requirements, poor resource planning, and poor management. These risk factors have the potential to turn any software project into a disaster. There are three general problems existing in the software project management environment:

1. Effort estimation is not accurate.

   Many factors must be dealt with when constructing an accurate software effort estimate and developing a realistic project development plan. Current methods for effort estimation are inadequate for developing an accurate plan. Jones [58] stated that one of the reasons why a software project fails is management's failure to use accurate effort estimate due to fault of existing cost estimation models. Even though many effort estimation models were suggested to date, none of them succeeded to predict development effort accurately. The error rate

can be higher than 30% of the actual productivity [28]. Managers use wrong software effort estimates and brute force plans to manage software development projects.

2. Actual status is difficult to understand.

   Measuring the status of a software project involves collecting, validating, and presenting true accurate status of software metrics and project data in a timely manner. However, the project attributes are dispersed across the various CASE tools and are difficult to measure because they are not hard stuff like building blocks. They often do not know the actual status of the software product, when problems happen, and how to refine the plan to solve problems. To date, no system of standard check points for software projects exists that functions to point out clear and unambiguous indicators of possible failure or success [58]. This leads to the subjective "90% completion" assertions by managers or developers.

3. Projects are often geographically dispersed.

   In modern software development environments where organizations are separated and dispersed across countries and continents and where a software project can be made up of a number of different initiatives, controlling software projects becomes very difficult. Existing project management software does not come close to supporting wide area collaboration and project management.

The main tasks of software project management include planning, estimating, tracking, and decision making. If the project's progress continues to match the plan, the project is in good shape. If there are some mismatches between the progress and plan, then corrective action must be taken.

The software project management depends on the plan. Planning information

includes schedule and resource estimates, which provide the standards used for assessing the significance of what is happening. Indeed, it has been stated that the degree of control over a project can be no greater than the extent to which adequate plans have been made for the project [84]. Therefore, accurate planning information is a key to a successful software project. However, planning information is often not accurate in the beginning of a project due to incorrect information about project environment, resource estimates, customer requirement, etc. In the very early stages, one may not know the specific nature of the product to be developed to better than a factor of 4 [21].

In a software development project, managers use an effort estimation model to estimate initial productivity. However, there is anecdotal evidence that effort estimation models have high error rates [63][68][76]. As a result, it is recommended to calibrate the model to an organization's own actual data to increase the model's accuracy [28]. But it is usually hardly successful to calibrate the model because:

- Most managers don't have knowledge and experience in calibration
- Organizations often have no enough history data available
- The suggested method to calibrate is academic oriented but not practical

With the problems listed above, a project plan can be fraught with inaccuracies and managers can suffer from poor control over the development process, which result in budget overrun or project failure.

Figure 1 shows the uncertainty estimate according to the project life cycle. The uncertainty decreases as the life cycle proceeds, because product decisions are made, and the nature of the product and its consequent features are better known. The uncertainty of a project nature adds serious difficulty to providing of accurate planning

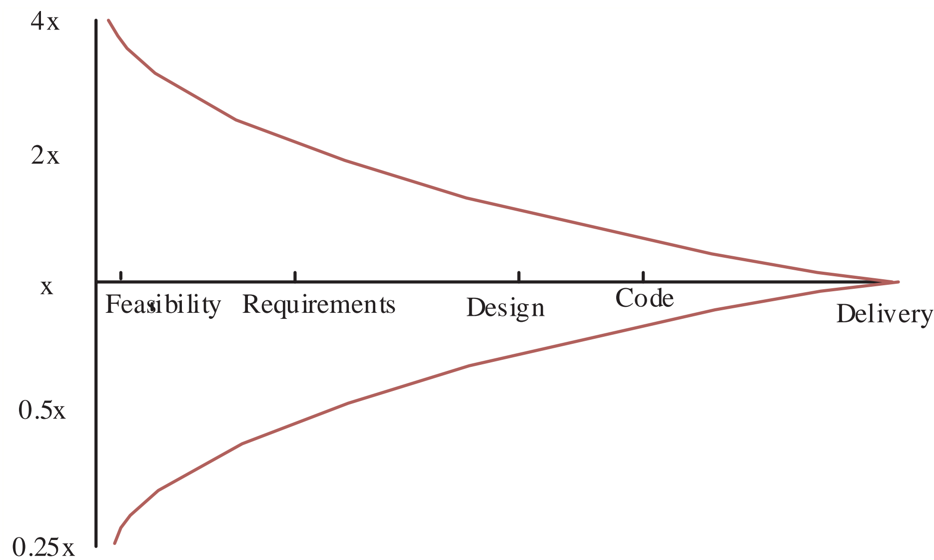


Fig. 1. Uncertainty estimate

information. Therefore, the reevaluation of the planning information as the project evolves gives more accurate management of the project.

To manage a software project, managers should know the status of every task of the process. In short, project management is a management activity aimed at ensuring that the progress of a project conforms to its plan. However, most assessment depends on manual procedures. Inaccurate status information resulting in developer's subjective objection could lead to faulty decision making and cause project delays. For example, a manger asks a developer the following questions regarding the progress of an activity, "How are you doing with your coding?" and the developer replies, "Well, I am almost done." or "I have my coding 90% done already." Those kinds of verbal communications might not be correctly reflecting the actual status of the software project. Manual procedures are fraught with inaccuracies and subjective interpretations of what should be accurate quantitative measurements. An incorrect decision based on faulty assessment can result in project failure.

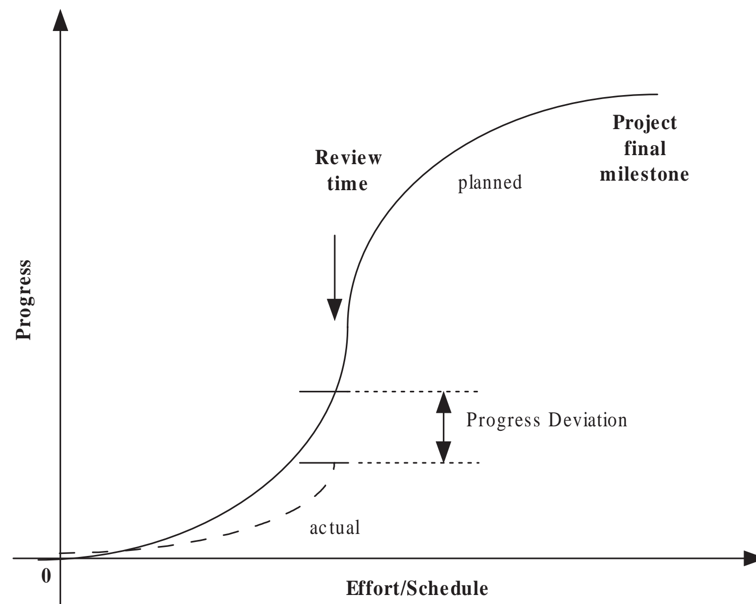


Fig. 2. Progress curve

Figure 2 shows a plot of progress in a project development cycle. The plot of progress is an S-shaped curve [70]. A project starts slowly at the initial stage, because progress is low due to a new development environment, technology, and suffering from slow learning curve, incomplete and ambiguous specification, and frequent changes of user requirements. Once those problems settle down, progress increases rapidly. At the completion time nears, progress slows again when developers start testing and debugging, minor changing of programming module, writing comment and documentation, etc.

More over, many unforeseen factors can affect software project management. Many mangers use the typical *ad hoc* software project management model. When project starts, everyone works hard, and software comes out some time later. If someone asks, the project is always "on the schedule." The manager only has the cloud view of the project status. This, together with the fact that the end software product is hard to visualize, has led to a situation where many software development

projects are carried out in an *ad hoc* fashion and very often fail to meet their success criteria. In Figure 2, actual progress does not coincide with planned progress at a review time so a progress deviation exists. Therefore, it is essential to provide objective assessment of project progress in project management as software development evolves.

Software development is harder to visualize than hardware fabrication. Software is largely invisible. Relative to hardware, it is mostly intellectual. People can touch a computer, a car or a building and can hold a printout of software or a disk or tape containing magnetic images of software, but software is viewed through the minds of an observer. When starting a software project, they have a clear start time, and people typically know when they are finished, but the body of the software product is largely hidden. For software projects, it becomes difficult to meet the deadlines and to deliver the product features as promised and within the budget. The software industry has faced its share of trouble as it has grown. It is possible to minimize these risks by using automated project management tools [53].

A number of process management environments (PMEs) and project management tools (PMTs) have been developed for project management. In general, PMEs provide various features to support process modeling, process automation, and cooperation among workers. PMTs, such as Microsoft Project, AMS Timekeeper, Business Engine, CASCADE, Innate Multi-Project and Timesheets, Micro-Frame Program Manager for Windows, Risk+, Schedule Publisher, and Time Line [29] provide features to support project planning and scheduling. However, they fall short of supporting the project managers in their decision making processes.

Those tools are based on the deterministic optimization techniques and do not issue warnings regarding possible future schedule slippage, analyze the causes of delays, or provide recommendations for remedial action. Those tools are based on manual

gathering and assessing of the project data, which result in subjective assessment of the project status. They do not provide the necessary features to assist the software project managers in objective project management. For example, we need a tool to explicitly describe if the progress of project activities has been accomplished, determine the current productivity of individual, team and project, or discover if resources are adequate. Without the correct information, it becomes impossible to actively monitor project failures and identify appropriate repairs before the project fails. Using existing tools, the managers do not know whether the project is going according to the plan. Further aspects, which are not addressed by today's management systems to support the software project manager, are the distributed and cross-platform nature of system development [32].

I.2 Research Objective

Most important aspect of a software development project is to estimate development productivity, i.e., estimating development productivity is central to the project management. Productivity is a major attribute for project management to keep track of project status. Productivity of a project can be estimated with an effort estimation model, for example, COCOMO II. However, no current effort estimation model can provide an accurate estimate. As a result, the initial productivity estimate is not accurate enough to tell the true effort of the project, which contributes to the uncertainty of a software development project. This leads to the first of two research questions:

Question 1. Can the reevaluation of the initial productivity estimate reduce the uncertainty caused by the inaccurate initial estimate?

We have researched various statistical tools: regression analysis, logistic regression analysis, stepwise ANOVA, robust regression analysis, Bayesian analysis, etc. Of all these tools, we chose Bayesian analysis as a candidate for our research, because it provides a mechanism of feedback to improve an inference of parameter as well as a system of using prior information available for the parameter as a starting point. In this research, we are more interested in the reevaluation of the initial productivity estimate obtained from an effort estimation model. So we devised the second research question:

Question 2. Can Bayesian analysis be a good tool in the reevaluation of the initial productivity estimate?

Compared with probabilistic modeling, the purpose of a statistical analysis is fundamentally an inversion purpose, since it aims at retrieving the causes summarized by observations. In other words, when observing a random phenomenon directed by parameter θ, statistical methods allow to deduce from these observations an inference about θ, while probabilistic modeling characterizes the behavior of the future observations conditional on θ [91]. Bayesian analysis relies on the probabilistic distribution of parameter θ, therefore, it provides prediction capability. Besides, these two questions lead to the main hypothesis of this research.

Hypothesis. Productivity prediction based on Bayesian analysis reduces the uncertainty by providing a better productivity estimate in a software development project.

To prove the hypothesis, we proposed four objectives of this research. First, the overall goal of this dissertation is to build a productivity prediction model based on

Bayesian analysis that, we believe, provides improvement in the inference of parameters. Reevaluation of the initial productivity estimate as the project evolves gives managers more command of the project management.

Second, we will create a system based on the productivity prediction model. It will be a prototype of integrated three-tier system capable of working on the Internet environment and PAMPA II system.

Third, we will gather real time data to validate the productivity prediction model. The required knowledge objects of project attributes are represented in PAMPA II knowledge base. PAMPA II system is used to gather project attributes in a quantitative and objective procedure that remove inaccuracies and inconsistencies from the monitoring, measuring, analyzing, and reporting assessment.

Fourth, a productivity console will be created to visualize the project status on the Internet environment. The productivity console provides a view of progress, current productivity, productivity and resource balance of a project via graphical charts. The productivity console can be a navigator for managing a software project to reach the desired destination. The graphical charts are working on an Internet web browser, which help managers keep track of project status remotely.

PAMPA II was recently developed to describe plans based on an incremental evolutionary project life cycle [100]. Knowledge can be acquired from software development experts and CASE tool databases to create a knowledge base. Metrics gathered from CASE tool databases can drive a visualization toolkit to assist managers in directing a software project [97][98][49]. The expanded tool is used with a Software Project Planning Associate (SPPA) that can track work breakdown structures compliance to plans [113]. The results of this research have been published in conferences [115][114].

I.3 Organization of the Dissertation

Following this introductory chapter, this dissertation is presented in six additional chapters. Chapter II presents relevant background research. This includes literature on the subject of software project management. Bayesian analysis is described in Chapter III. Chapter IV describes the features and subsystems of PAMPA II. Chapter V discusses the productivity prediction model. Chapter VI explains project experiment results used to test the research model. Chapter VII presents the conclusions and discusses future extension of this research.

CHAPTER II

LITERATURE SURVEY

II.1 Introduction

Software project management is a key process leading to a successful software project [19]. The goal of the software project is to produce software to customer satisfaction that is on time and within budget. For many managers of large complex systems, management is the biggest challenge in the development process. Many methods and techniques have been studied, and various commercial support tools have been introduced to assist managers in resolving the typical software project management problems. This chapter describes the software project management, definition of productivity, existing effort estimation models and the popular effort estimation model, COCOMO II, and effort estimation technique of commercial-off-the shelf (COTS) system.

II.2 Software Project Management

A software project is a planned process of creating artifact activities that occur within a specified time and have the goal of delivering to customers a satisfactory software product on time and within budget. Measuring the status of a software project involves collecting, validating, and presenting true accurate status of software metrics and project data in a timely manner. The main activities of software project managers include planning, estimating, tracking, and decision making. If the project's progress continues to match the plan, the project is in good shape. If there are some mismatches between the progress and plan, then corrective action must be taken.

Fundamentally, control is any process that guides activity toward some prede-

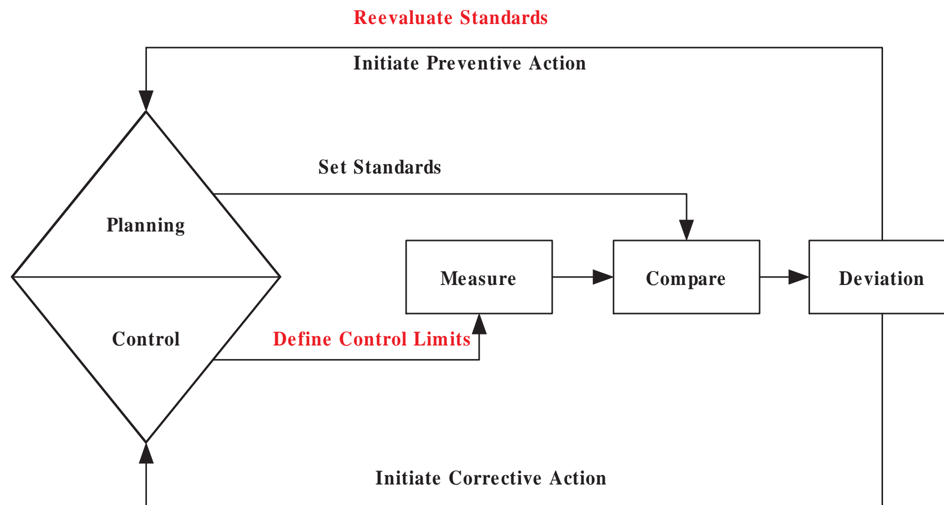


Fig. 3. Project tracking and control model

termined goal. The essence of the concept is in determining whether the activity is achieving the desired results. Figure 3 depicts a standard model of project control [9]. A control system is shown to have four basic elements:

- a measuring device which detects what is happening
- a mechanism for comparing what is actually happening with some standard or expectation of what should be happening
- a procedure for altering behavior if the need for doing so is indicated
- a means of transmitting feedback information (reevaluation of standards) to the planning device.

Two information inputs are vital to effective project control. The first is planning information, such as resource and schedule estimates, which provide the standards used for assessing the significance of what is happening. The second is accurate and timely status information.

Software project management is a key process leading to a successful software project. In simple terms, software project management can be defined as deciding what to do, how to do it and who does it, setting objectives, breaking work into tasks, establishing schedules and budgets, allocating resources, setting standards, and selecting future courses of action [90]. Software project planning is also concerned with identifying the activities, milestones, and deliverables produced by a project [103]. A plan must be drawn up to guide the developers toward the project goal. The software development plan is one of the formal documents for project management. In the plan, the manager describe, in detail, how the project will be developed, what resources will be required, and how these resources will be used. A plan is also a tool for communicating and building commitment. By defining what needs to be done, when, and in what order, the plan offers a unique opportunity for people to see where they fit into the big picture.

According to the Software Engineering Process Office (SEPO) of the United States Navy, the Software Development Plan (SDP) is the most critical planning document for a software development project. The SDP address cost, size and schedule, project risks, project tracking (metrics), methodologies, and technologies to be employed. It is a dynamic document that guides the software project manager and staff members through the software development process.

Boehm [20] defines a plan and summarizes the elements of a good project plan with *WWWWWHH* planning: who, what, where, when, why, how and how much, as follows:

Objectives Why is the system being developed?

Milestones and Schedules What will be done? When?

Responsibilities Who is responsible for a function? Where are they organization-

ally located?

Approach How will the job be done technically and managerially?

Resources How much of each resources is needed?

The first major step in planning is to choose a development process that will fit the product and people [80]. The second major step is to derive tasks and a way to execute them according to the process model chosen. The planning process must encompass both the product that is being produced and the accompanying processes that are required to support the product [82].

According to Phillips, a plan requires three items [81]:

Task list Tasks are the building blocks of a plan. Each task has input and produces output. The task is not completed until a review approves the output.

Resource Each task requires some degree of time, people, and equipment.

Task network A task network shows task precedence and dependency. It lets the manager verify that each task receives its prerequisites from previous tasks and sends its output to another task. If the prerequisites are not present, the manager needs to create tasks to build them. If the outputs go nowhere, the manager can eliminate the task.

Making estimates about the software project before it has even begun is very important to software managers. These estimates can be derived using a variety of methods. When planning a project, the manager will first consider constraints on the project. These constraints include the required delivery date, the staff available, and the budget. The manager will also make estimates about such things as project size and structure. The manager will then define the milestones and deliverables and

construct a schedule. As the project moves toward the goal, the manager assesses progress and adjusts the schedule accordingly. As more project information becomes available, the manager will revise the initial estimates and make them better and more accurate [103]. The foundation of estimation is metrics [101]. Project attributes and software metrics are the important indicators of how the software project is progressing. Humphrey [52], in his paper, "The Personal Software Process," calls for recording the time required by all tasks in minutes to estimate about how long it takes to finish the tasks. Recording software metrics and project attributes for every task and every developer for future use and analysis of the on-going project are the project visualization process.

Sommerville [103] states that effective management of a software project depends on thoroughly planning the progress of the project and presents a method for doing so. The project manager must anticipate problems which might arise and prepare tentative solutions to those problems. A plan, drawn up at the start of a project, should be used as the driver for the project. Project planning is probably the activity that takes most management time. Planning is required for development activities from specification through to delivery of the system.

Sommerville [103] also describes that most plans should include the following plan structure:

Introductions This section briefly describes the objectives of the project and sets forth the constraints (such as budget, development time, and so on) that affect the project management.

Project Organization This section describes the way in which the development team is organized, the people involved, and their roles in the teams.

Risk Analysis This section describes possible project risks, the likelihood of these

risks arising, and the risk reduction strategies that are proposed.

Hardware and Software Resource Requirements This component describes the hardware and the support software required to carry out the software project development. If hardware has to be bought, estimates of the prices and the delivery schedule should be included.

Work Breakdown (Task Plan) This section describes the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity.

Project Schedule This component described the dependencies between activities, the estimated time required to reach each milestone, and the allocation of people to activities.

Monitoring and Reporting Mechanisms This section describes the management reports that should be produced, when these should be produced, and the project monitoring mechanisms.

Conger [31] also describes the following steps to developing a software project development plan:

1. Decide the development life cycle, approach, and methodology.
2. For each phase, list the deliverable products that mark completion of the phase.
3. Decide on information gathering technique(s) and use of Joint Application Development/Design (JAD), prototyping, or other variants to the development life cycle.
4. Decide which products the technical project team members will develop and which the users will develop.

5. Define dependencies and develop Critical Path Method (CPM) chart.
6. Assign times to tasks and compute total project time.
7. Estimate inputs, outputs, interfaces, queries, and files according to function point directions.
8. Use function points rating to estimate project complexity.
9. Compute function points.
10. Look up lines of code per function point (FP) in the language table and compute total lines of code (LOC) for the project.
11. Estimate productivity in LOC/month.
12. Compare FP number of person months to the estimated total time
13. Adjust time estimate, as required, and complete the CPM diagram by adding times.

The Software Engineering Institute's Capability Maturity Model (SEI CMM) provides a well-known benchmark of software process maturity [25][79][102]. The CMM has become a popular vehicle in many domains for assessing the maturity of an organization's software process. The SEI Maturity Questionnaire has a scenario on software project planning for evaluating the completeness of the planning framework as follows:

1. Are estimates (e.g., size, cost, and schedule) documented for use in planning and tracking the software project?
2. Do the plans document the activities to be performed and the commitments made for the software project?

3. Do all affected groups and individuals agree to their commitments related to the software project?

4. Does the project follow a written organizational policy for planning a software project?

5. Are adequate resources provided for planning the software project (e.g., funding and experienced individuals)?

6. Are measurements used to determine the status of the activities for planning the software project (e.g., completion of milestones for the project planning activities as compared to the plan)?

7. Does the project manager review the activities for planning the software project both a periodical and event-driven basis?

Software project planning is a Level 2 Capability Maturity Model (CMM) Key Process Area (KPA) [79]. Satisfying the KPA is a major step toward achieving Level 2 (Repeatable). This KPA requires a written process for planning a software project. It also requires the development of a project Software Development Plan (SDP). The CMM [79] defines 15 activities for the Project Planning KPA. These activities assure the appropriate participants are involved in the process. It also forces the company to document a defined process for developing the plan. The 15 steps in the CMM Planning KPA are as follows:

1. The software engineering group participates on the project proposal team.

2. Software project planning is initiated in the early stages of, and in parallel with, the overall project planning.

3. The software engineering group participates with other affected groups in the overall project planning throughout the project life.

4. Software project commitments made to individuals and groups external to the organization are reviewed with senior management according to a documented procedure.

5. A software life cycle with predefined stages of manageable size is identified or defined.

6. The project's software development plan is developed according to a documented procedure.

7. Software work products that are needed to established and maintain control of the software project are identified.

8. Estimates for the size of the software work products (or changes to the size of software work products) are derived according to a documented procedure.

9. Estimates for the software project's effort and cost are derived according to a documented procedure.

10. Estimates for the project's critical computer resources are derived according to a documented procedure.

11. The project's schedule is derived according to a documented procedure.

12. The software risks associated with the cost, resources, schedule, the technical aspects of the project are identified, assessed, and documented.

13. Plans for the project's software engineering facilities and support tools are prepared.

14. Software planning data are documented.

15. Measurements are made and used to determine the status of the software planning activities

Hughes introduces the Step Wise planning [46] to complement PRINCE, which is the publication of the government standard in Europe for the management of Information Technology (IT) projects. It emphasizes the iterations of planning in an outline first and then in more detail as the time approaches to tackle a part of the project. An overview of the "Step Wise Planning" framework is the following:

Step 0 Select project

Step 1 Establish project scope and objectives

Step 2 Establish project infrastructure

Step 3 Analysis of project characteristics

Step 4 Identify project products and activities

Step 5 Estimate efforts for each activity

Step 6 Identify activity risks

Step 7 Allocate resources

Step 8 Review/publicize plan

Steps 9/10 Execute plan/lower level of planning

From the disciplines above, one can see that, in general, there are five basic important components of a software project plan [51].

Goals and Objectives In the end, the goal for the project is to deliver a quality software product that meets a customer's needs, and to do so on time and within budget. The project's goals and objectives are determined in the requirements negotiation phase. The initial statement of work must be clear, straightforward, and stable because it will be the statement from which the software development company will determine the product's functional goals.

Work Breakdown Structure The WBS was introduced into software project planning in the early 80's [107]. WBS provides a hierarchical view for the whole project, but the precedence relationships among the work packages are not clearly identified in the Work Breakdown Structure. After the requirements have been declared, an estimate of the product size and project effort is required. To produce an effective estimate requires the project to be broken down into its various work elements comprising the project WBS. Project structure and the software process affect the WBS. Once the project structure is defined, the process tasks for each unit of the project will be defined and allocated to the appropriate design group. The design of the WBS should be as detailed as possible, such that each task can be completed by a small team in a fixed time. A well detailed WBS leads to more accurate estimates and a better overall plan [40].

Product Size and 17 Other Dominators This is probably the most critical portion of the planning process. The 17 project dominators are as follows: Development Schedule Constraints, Project Life Cycle process, Volume, Amount of Documentation, Programming Language, Complexity, Type of Application, Work Breakdown Structure, Management Quality, Lead Designer, Individual Developers, Personal Turnover, Communications, Number of People, Software

Reuse, Customer Interface Complexity, and Requirements Volatility Dominators are project attributes that cause effort (and productivity) to vary by an order of magnitude (10 to 1) [99]. A poor size and dominators estimate is the root of many problems in the software industry. Dominators may or may not appear as variables in effort models. For example, we often assume that all projects are properly managed, even though they may not be. The result can be a failed project dominated by poor management. Dominators like management often do not have a 10:1 affect on reducing effort, but they definitely can have over a 10:1 affect on increasing effort [99]. Productivity is improved when managers reduce the effort to produce a product, as effort required to produce a product is inversely related to productivity. Dominators that affect effort prediction are everywhere in the project life cycle and are not independent of each other. Product size is useful for predicting effort. Two units are common for size measurement: lines of code and function points. A line of code is a fixed unit and easier to count, but is language-dependent. Function points are a more subjective and abstract unit, which is subject to bias [99].

Resource Estimates The amount of effort spent on a project is limited by resource constraints [99]. Given an estimate of the amount of code that is needed for the software product, a manager can estimate the resources that are required to design and implement it. Human resource is the most important of these resources as it plays the most important role in determining the cost of implementation. Many tools for cost estimation are available. A software company's historical performance plays the most important role when it comes to estimation of resources. The historical productivity rate can be applied to a new estimate to convert a size estimate into a corresponding estimate of resources.

If a cost model such as COCOMO or SLIM is used, its calibrations must match the software company's historical experience [99].

Scheduling The scheduling of a project is dependent on the resource estimates. Two situations arise in scheduling, depending on which side sets the release date for the product [85]. Usually, the project manager will make the decision on the release date based on an appropriate starting date and schedule. However, should the customer require the product by a certain date, the software developer must schedule all tasks to be completed before this date. The latter situation is much more difficult. The software developer may or may not be able to meet the deadline depending on their existing commitments. Overtime and extra staff may be required. The schedule must fulfill the needs of all parties involved in the project before the development can begin. If no such schedule is possible, requirements must be re-negotiated. Three optional approaches are available for scheduling such as Gantt Charts, Milestone Documents, or Project Evaluation Review Technique (PERT) Charts [92].

Software project management is, like many other activities in the software process, a problem-solving issue. It has two principal phases: planning, including creation and scheduling, and on-going project control [106]. These involve what is to be done, a decision regarding how to do it, the control of how it is being done, and an evaluation (or measurement) of what was done [27]. The issue on "what" typically takes the form of a plan. Many tasks have to be performed for a manger to properly manage a software project. In general, these tasks fall into the following categories: planning, organization, staffing, monitoring, controlling, innovating, and representing [15]. Every project, no matter what the industry or work type, including the software development project, is a compromise among three variables: scope, time,

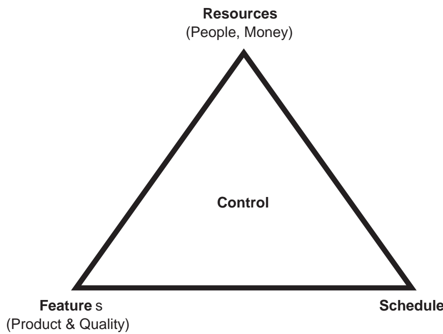


Fig. 4. Simmons' project triangle

and cost.

A project planning process includes the activities of comprising the three vertices defined in the project triangle as shown in Figure 4. Scope is the total amount of work to be conducted, the sum of the activities that will lead, at the end of the project, to the "deliverable" or "product." Cost is sometimes referred to as "budget," the total resource usage required to accomplish the work scope. Time is the total elapsed time, from the concept to completion, that it takes to perform the work scope. The project management deals with those three variables to show the impact of any change across all three.

The role of the project manager is to establish a plan, select the right personnel for task assignment, track and review the results, and modify the plan when appropriate. To succeed at a software project, a manger must compromise those three variables, resource, feature, and schedule to comply with the plan. If any one of the triangle vertices is adjusted, one or both of the other vertices must be modified and the plan has to be tailored for a project to stay on track. If a project is behind schedule, the manager can add resources or decrease features. If a project is ahead of schedule, the manager may decide to decrease resource or add features. If the manager wants to add features, s/he must lengthen the schedule or add additional

resources. If the manager wants to reduce resources, s/he must decrease feature or lengthen the schedule.

According to Dwayne Phillips [81], all undertakings in a software project include the 3Ps: people, process, and product. A successful software project requires keeping these three in harmony to comply with a project plan. People are critical to software development and maintenance. Software development is people-intensive. The best asset on a software project is people who know how to build the product. Process has become the most discussed aspect of the 3Ps in recent years including some of the famous software process improvement methods, the Capability Maturity Model, the ISO 9000 series, and Best Practices. Process is important because it lets people build products. Before starting a software project, the manager first defines a process needed for the project in a plan. Process is repeatable, but the same process does not fit all projects, even though with the similar goals. The objective of software development is to create a product. The product must satisfy the customers and within budget. Without a product, there is no customer, no income, and no software organization. Figure 5 shows how people, process, and product fit together. The axes represent the capabilities of people and process. The distance from the origin of the graph represents how difficult the product is to build. The mission of the manager's job is to keep 3Ps in balance to create a good quality product.

The Microsoft process [81] is based on the three dimensions of quality: reliability; feature set; and schedule. Reliability is how good the product must be before shipping. Feature set is the product's definition (the requirement), and schedule is the ship data. The relative importance of the three variables would be changed with the product. For example, entertainment products must ship before Christmas, but do not need to be as reliable as a spreadsheet.

Rapid Application Development (RAD) [65] employs the best available people

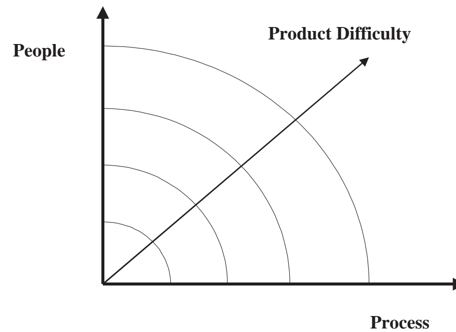


Fig. 5. Considering people, process, and product together

and process to build a product with the features most valuable to the customer in the quickest manner. RAD concentrates on people, process, and product. The people focus is to discover the 20% of what the customer wants that delivers the 80% of what they need. This usually begins with people surveys that aim to shorten the list of wants. From the list, the manager finds the core requirements to build the product of prototype. The product part of RAD emphasizes essentials only to enable rapid delivery: bring customers a product quickly. The process part emphasizes throughput (rapid delivery), but not at the expense of sound engineering. The process is iterative or evolutionary and gives a product to the customer in a series of deliveries. The first delivery has limited functionality, but is delivered rapidly. This keeps the customers involved and gains their confidence and trust.

The software project management discipline is more of a discriminator in success or failure than are technology advance [59][57][93]. The major disciplines necessary for an effective management work flow are: planning, organization, automation, and project control. The challenge is to develop a plan that best balances the available resources to provide optimal win conditions for all stakeholders. The project organization discipline concerns itself with the management of people: organizing them into teams and allocating responsibilities for efficient operations. Automating the

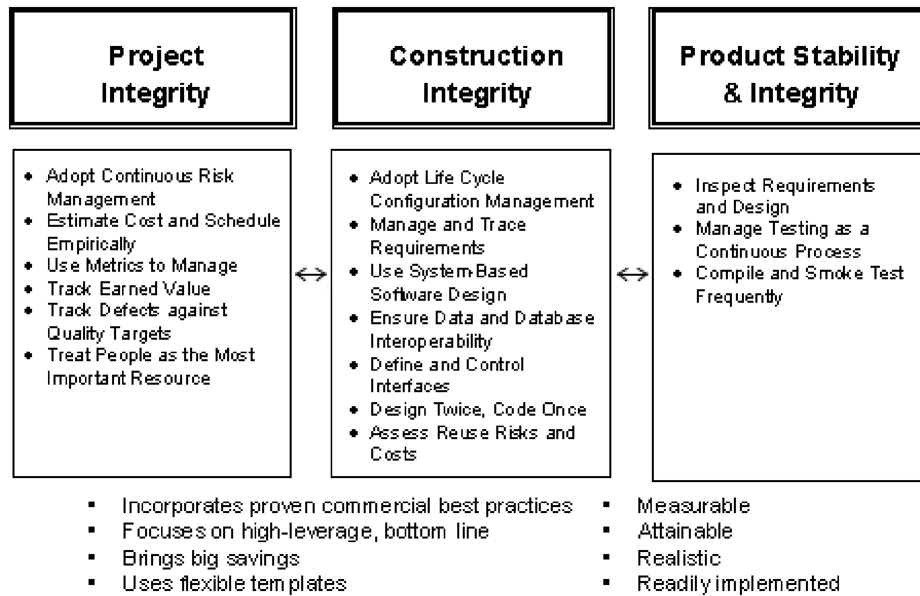


Fig. 6. 16 critical software practices for performance-based management

development process with an electronic repository for the artifacts provides a foundation for objective instrumentation. Project control activities act as the "sense" of the project. They are used to assess the health of the plan, the quality of the artifacts, and the need for changes to any of the management set of artifacts that define the expectations among stakeholders.

The correct and effective planning process can lead to a successful software project. The practice from the Software Program Management Network outlines the 16 Critical Software Practices [75] that serve as the basis for implementing effective management of software projects as shown in Figure 6. The "16-Point Plan and Templates for Critical Software Practices" contain the 16 practices (9 best and 7 sustaining) that are the key to avoiding significant problems for software development projects and must be incorporated in the planning phase (See Appendix A for details). These practices have been gathered from past real-world, large-scale, software

development and maintenance projects. Together, they constitute a set of disciplines that is focused on improving a project's bottom line. These practices can be used as the starting point for structuring and deploying an effective process for managing large-scale software development and maintenance.

Shenhar introduces the concept of the software management style in a holistic way [96] as shown in Figure 7. It assumes that project management is more than just tools or processes, and it directs people's attention to higher levels of awareness which have substantial impact on project performance. The holistic approach includes the following five components: strategy, culture and attitude, organization, process, and tools. The key to software project success is integration of all the styles of the previous techniques and approaches. A holistic framework uses the classical planning concepts and the project management integration knowledge areas such as cost, time, etc., to generate an integrated matrix for the project management. This framework raises an integrated concept of the project management and planning.

II.3 Definition of Productivity

A software **Project** develops a **SoftwareProduct** (**SP**). **SP** status can be observed by tracking **Feature**s, **Artifact**s, known **Defect**s, reported **Problem**s, testing **Activit(y)ies** and approved **Change**s. Examples of **Artifact** are user requirements, design documents and source codes. An important **Artifact** is source code used to create the executable file that is delivered to a customer.

Volume attribute is used to describe physical magnitude, extent or bulk of artifacts [99]. Equivalent source lines of code, function points, and object points are metrics used to measure **Volume**. **Volume** can be used to track the progress of development. Effort attribute is the amount of resource expense required to produce

Fig. 7. Project management style

an **Artifact**. The effort of personnel is the main cost in a software development project. A widely used effort metric is person-month (PM).

Software productivity is the rate at which **SP Artifact**s are produced in relation to the time, and resource. Software productivity is usually defined as **Volume** divided by effort. Software productivity Pr is expressed as:

$$Pr_i = \frac{V_i}{E_i}, \tag{2.1}$$

where V_i is the **Volume** of **Artifact** i, and E_i is the amount of effort expended to produce **Artifact** i.

The Internet environment enables development to be distributed across the world.

And when the budget for development is limited, employing cheaper labor can decrease the total cost of development. For example, an entry-level programmer's salary ranges from $167 to $417 per month in India. That programmer's US counterpart typically commands $4,167 to $5,000 per month [45]. Therefore, the salary is an important attribute to account for resource expense in developing an SP in more than two countries. As shown in Figure 8, the current productivity model assumes dollar cost as a main factor to estimate productivity. Labor cost LC is:

$$LC_i = E_j \times S_j, \tag{2.2}$$

where S_j is **Salary** rate of a person j. After taking labor cost into account, we can change productivity as:

$$Pr_i = \frac{V_i}{LC_i}, \tag{2.3}$$

Productivity will be calculated in **Volume** per dollar. Given their performance are same, programmers in India are 10 times more productive than those in US when you use labor cost instead of effort. Nowadays, many software companies outsource their development work to other countries that have cheaper labor. And labor cost gives a manager a view of controlling resource expense in the multisite development environment.

Early research in cost estimation concentrated on determining causes for the wide variation of project productivity. In Boehm's COCOMO II model [21], 17 software factors (cost drivers) that have a significant impact on productivity were identified. In an IBM study by Walston and Felix [110], 29 factors that were significantly correlated with productivity were found. In an analysis of data from the NASA/Goddard Space Flight Center, Bailey and Basili [11] identified 21 productivity parameters. At ITT, Vosburgh et al. [108] found 14 significant productivity factors, with modern

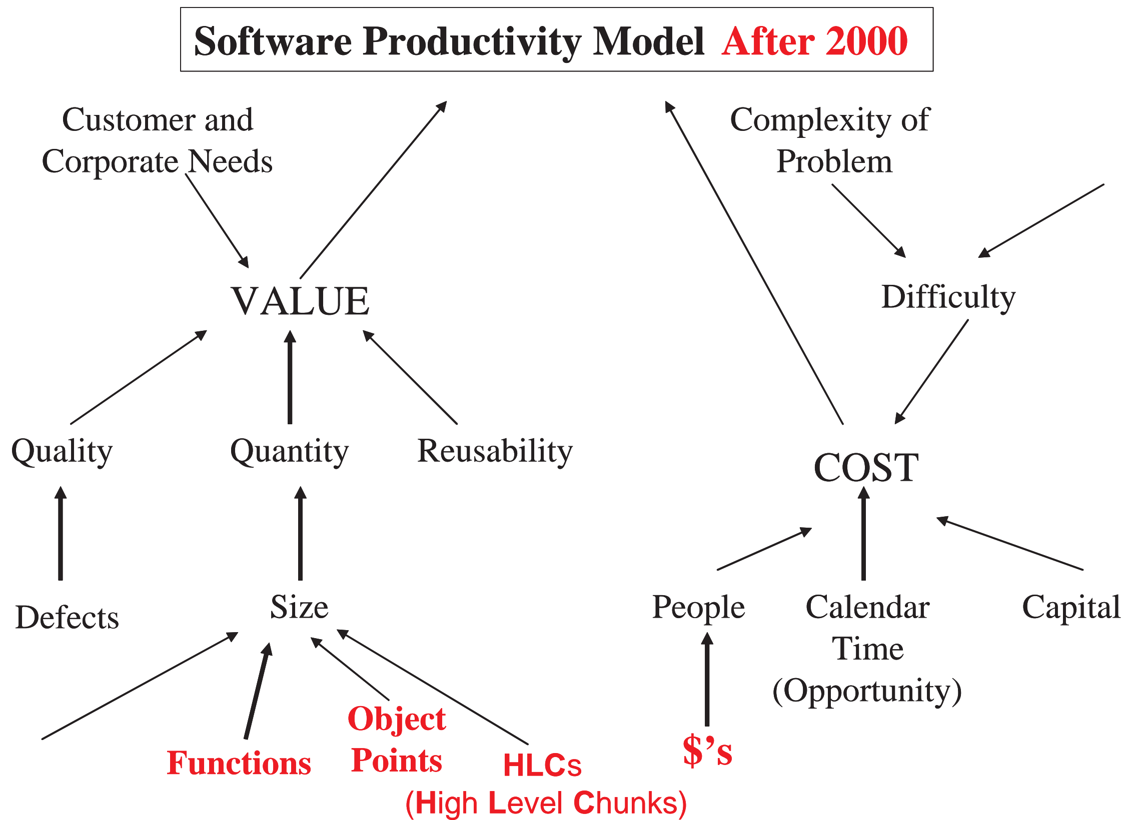


Fig. 8. Current productivity model

programming practice usage and development computer size explaining 24% of the variation in productivity.

Several studies attempt to determine nominal productivity rates depending on the type of software [36][71][87]. The productivity of subsystems that were part of a ballistic defense system was found to be a function of software type, with real-time software having the lowest productivity. Vosburgh et al. [108] identified three different programming environments with business applications having the highest average productivity followed by normal-time and real-time applications. These environments were characterized by the hardware used, resource constraints, application complexity

and programming language.

Aron [7] found that the variation of productivity for a group of IBM projects involving systems programs and business applications was due to differences in system difficulty, characterized by the number of interactions with other system elements, and project duration. Kitchenham [66] found that productivity varied with programming language level and working environment. Productivity has also been found to vary with hardware constraints [21][108], programmer experience [21][108][71][60], team size [33][24][55], duration [7][10], project size [108][33][55][14], and modern programming practices [21][108][60] among other factors.

II.4 Effort Estimation Model

Software cost estimation is as much a relevant area of research now as it was 30 years ago, when difficulties of estimating were discussed in "The Mythical Man Month" [24]. The purposes for which an estimation is required are as follows:

- Exploring the feasibility of developing or purchasing a new system
- Planning how to staff a software development project
- Quoting a price or schedule for a new system
- Exploring the impact of changing the functions of an existing system

However, software cost estimates are typically inaccurate, and there is no evidence that the software engineering community is improving its ability to make accurate estimates. In spite of the research effort in developing software cost estimation models, it is true that most estimates are made informally, or cost models give estimates with significantly greater inaccuracy [48][72]. This suggest that software developers have difficulty in applying existing research on software cost estimation.

Inaccurate estimates of software cost and delivery times have unacceptable consequences. For example, where effort is underestimated, a cost overrun may make a project unprofitable, and overruns in delivery time may result in project failure. An overestimate of effort may also adversely affect the competitiveness of a business, for example, where a decision is made to cancel what would otherwise have been finished in time or where the overestimate leads to subsequent overstaffing when a project is completed.

Over the past three decades there has been considerable activity in the area of effort estimation with five classes of estimation models:

- Empirical parametric models
- Empirical nonparametric models
- Analogical models
- Theoretical models
- Heuristic

Heuristics are rules of thumb, developed through experience, that capture knowledge about relationships between attributes of the empirical model. Heuristics can be used to adjust estimations made by other methods. For example, Cuelenaere et al. [35] describe an expert system that uses rules to assist in calibrating the PRICE SP software cost estimation model.

The most common estimation models are empirical parametric models. Any estimation that relates the attributes of interest to other measurable attributes must be based on an empirical parametrical model. Where effort is estimated based on one or more simple measures, these models have been extended, in some cases, by the use

of cost drivers. Empirical parametric methods analyze data to establish a numerical model of the relationship between measures of the attributes in the empirical model. Statistical regression analysis is one example of empirical parametric models.

The simplest form of an empirical parametric model is a function that relates the effort to develop a system or program to a size measures. In this context, a size measure is a count of some feature of a product of the development process, for example, a count of the number of lines of code in a program. Effort is often measured in person-months. The models are developed by fitting the function to a data set of size and effort value pairs, using regression techniques. Models with linear and exponential relationships between effort and the size measure are most commonly explored. Whatever the exact niceties of the model, the general form tends to be:

$$E = a \times V^b, \tag{2.4}$$

where E is effort, V is **Volume** typically measured as lines of code (LOC) or function points, a is a productivity parameter and b is an economies or diseconomies of scale parameter. This model has been investigated by Walston et al. [110], Bailey et al. [11] and Boehm [18]. COCOMO II represents an approach that could be regarded as "off the shelf." Here the estimators hope that the equations contained in the cost model adequately represent their satisfactorily accounted for in terms of cost drivers or parameters built into the model.

Another empirical parametrical approach is to calibrate a model by estimating values for the parameters (a and b in the case of (2.4)). However, the most straightforward method is to assume a linear model, that is set b to unity, and then use regression analysis to estimate the slope (parameter a) and possibly introduce an

intercept so the model becomes:

$$E = a_1 + a_2 \times V, \tag{2.5}$$

so that a_1 represents fixed development costs (for example regression testing will consume a fixed amount of effort irrespective of the size the software) and a_2 represents productivity.

As seen in above, the development of an empirical parametric model is an exercise in curve fitting. So there are some pitfalls inherent in the development of these models. Courtney et al. [34] report that researchers who set out to discover empirical relationships by trying different combinations of measures and functional forms before choosing the one with the highest correlation tend to make a good model with small data sets.

Models based on empirical parametric approach give higher error rate in explaining the variation in effort, whether the functional form is linear or nonlinear. Conte et al. [33] give an example of a linear model with a correlation coefficient, R^2, of 82% and mean absolute relative error of 37%. Miyazaki et al. [76] give an example of a calibrated COCOMO model with a lower mean absolute relative error of 20%.

When a model has many input parameters, each with a range of possible values, the range of estimates generated by the model increases. Although such a wide variation in input values would not occur in practice, Conte et al. [33] report that a variation in effort of up to 800% possible in Intermediate COCOMO when the range from highest to lowest values for each cost driver is combined. The range of possible values for an estimate increases further when the uncertainty in input values is combined with the uncertainty associated with the model.

Furthermore, when empirical models are applied outside of the organization or environment on whose data are based, the estimations made by the model are likely

to be inaccurate, unless the model is re-calibrated using local data [63][68][55]. Even more genetic models such as COCOMO fail to make accurate estimations without calibration. Boehm and Miyazaki et al. describe procedures how to calibrate models [18][76]. However, models that include a large number of cost drivers are difficult to calibrate. The immediate difficulty is that the data set required for calibration may be much larger than is typically available within a single organization.

Briand et al. [23] describe the optimized set reduction (OSR) technique which uses the empirical nonparametric approach. It is a pattern recognition model for analyzing data sets based on decision trees. They compare the accuracy of the OSR technique to a COCOMO model calibrated for the combined COCOMO and Kemerer data sets and a stepwise regression model. The OSR technique has a lower mean absolute relative error than both the two parametric models, with the COCOMO model performing least favorably.

One advantage of OSR is that it can be applied with incomplete input data. It is possible to make an estimate for a project where only a subset of the cost driver values are known. Another advantage is that nominal or ordinal cost driver values can be used as inputs without being mapped to numeric multiplier values.

Srinivasan and Fisher [104] describe two further nonparametric methods for generating effort models. The first method uses a learning algorithm to derive a decision tree. The second method uses back-propagation to train an artificial neural network. These methods were also tested on the COCOMO and Kemerer [63] data sets. The effort estimates from the artificial neural network had a lower mean absolute relative error than the decision tree. Differences in the sampling techniques mean that the results presented by Srinivasan and Fisher [104] are not directly comparable with those of Briand et al. [23], although the same data sets are used. It appears likely that the accuracy of both the artificial neural network and the decision tree is comparable with

that of OSR and the stepwise regression model. However, Srinivasan and Fisher [104] indicate that the computatinal cost of training the artificial neural network is high in comparison to the cost of deriving the decision tree.

While most research into project effort estimation has adopted approach described above, there has been limited exploration of artificial intelligence methods. Karunanithi et al. [67] studied the use of neural nets for predicting software reliability, and conclude that both feed forward and Jordan networks with a cascade correlation learning algorithm, out of perform traditional statistical models. Wittig et al. [112] described the use of back propagation learning algorithms on a multilayer perception in order to predict development effort.

To be applied confidently, each of the techniques just described require a large number of data points because of the large number of independent variables and value ranges covered by the models. Both set of authors comment on the small size of the COCOMO data set (63 projects) for applying their techniques and on the desirability of all projects in the data set coming from the same environment. However, although the COCOMO data set may be small, it is significantly larger than many organizations could hope to collect. Even though, there is a large enough data set available within a single organization, it is hard to believe that all projects come from the same environment.

Decision tree, artificial neural network, and OSR techniques can still be applied where the number of independent variables is reduced to complement the size of the available data set, for example, lines of code as the single independent variable. However, it is unclear whether these techniques are superior to simple regression techniques under those circumstances.

Another study by Samson et al. [94] used an Albus multilayer perception in order to estimate software development effort. The work compares linear regression with

a neural net approach using the COCOMO data set. There have been a number of attempts to use regression and decision trees to estimate aspects of software engineering. Srinivasan et al. [104] described the use of a regression tree to estimate effort. They found the result were less good than using either a statistical model derived from function points or a neural net.

Analogical estimation methods use measures of the attributes from the empirical model to characterize the current case, for which the estimation is to be made. Known values of measures for the current case are used to search a data set for analogous cases. The estimation is made by interpolating from one or more analogous cases to the current cases. An advantage of these approaches to estimation is that they can succeed where no statistically significant relationships can be found in the data. Case-based reasoning is a form of analogical reasoning that employs five basic processes [109]:

- Construction of a representation of the target problem
- Retrieval of a suitable case to act as source analog
- Transfer of the solution from the source case to target
- Mapping the differences between source and target cases
- Adjusting the initial solution to take account of these differences

ESTOR is a case-based reasoning model to estimate development effort [77]. In ESTOR, the cases are software projects, and each is represented by the values of a set of measures. The measures used by ESTOR are function point components and Intermediate COCOMO model inputs. ESTOR retrieves one case to act as a source analog based on the values of the function point components of the project

for which the estimate is sought. A vector distance calculation is used to find the nearest neighbor. The initial solution or effort estimate for the project is the effort value for the analogous project. The differences between the analog and new project are determined by comparing the values of their measures. The effort value for the analog is adjusted to take account of these differences by applying a set of rules. The rules used by ESTOR are derived from verbal protocols of an expert whose estimates were accurate for the data set used. The rules adjust the effort value by a multiplier if particular preconditions on the target and source project values are met. The data set used to develop ESTOR is a subset of 10 projects from the Kemerer [63] data set. ESTOR was tested on all 15 projects of this data set, with a reported mean absolute relative error of 53%.

Atkison and Shepperd [8] describes a method for estimating development effort for a software project by analogy, which represents projects by their function point components [6]. Analogous projects are neighbors of the new project, identified by calculating the vector distance from the new project to other projects in the data set. Effort for the new project is estimated from a weighted mean of the effort values of its neighbors.

Shepperd et al. describe the tool ANGEL, which also supports estimation by analogy. ANGEL is based on a generalization of the approach of Atkison and Shepperd [8]. In ANGEL, the user can specify the measures on which the search for analogous projects is based. ANGEL can also automatically determine an optimal subset of measures for a particular data set. ANGEL can be requested to search for one, two, or three analogous projects and calculates an unweighted mean of their effort values to estimate effort for the new project.

Both ANGEL and ESTOR represent projects by values of readily available measures, and use a vector distance calculation to search for analogs. ESTOR uses only

one analog on which to base its estimate, whereas ANGEL may retrieve and use the effort values from several analogs. The main difference is that ESTOR adjusts the effort value of the analogous case by applying rules, whereas ANGEL will either use the effort value directly where one analog only is retrieved, or calculate a mean of the effort values for analogs.

ANGEL performed as well as or better than linear and stepwise regression models for effort. The regression models were based on the measures in the data set that displayed the highest correlations with effort. On the Kemerer [63] data set, the reported mean absolute relative error for ANGEL is 62%, which compares with more than 100% for the regression models and 53% for ESTOR. Although ESTOR appears to perform better than ANGEL on this data set, the adjustment rules for ESTOR were developed based on 10 of the 15 projects in the set, and these rules may not be as successful when applied to projects from difference data sets.

Abdel-Hamid and Madnick [95][4][3] have developed a theoretical model of software development project. Dynamic feedback relationships among staff management, software production, planning, and control are modelled via a simulation language. Simulations of project management scenarios can be run to investigate the effects of management policies and decisions. The model works from an initial estimate for overall effort and then explores how the actual effort is influenced by the model's assumptions about the interactions and feedback between project and decisions.

As estimates for new projects are based on past projects, they suggest that their model can be used to explore what the minimum effort for a completed project would have been, if it had been estimated correctly at the outset. Future estimates can then be based on the corrected effort for the project. Therefore, the model's assumption should be examined to see whether they are valid when the model is applied in a new environment, because the model relies on assumptions about management

policies that may be inaccurate in a new environment and hence invalidate the existing model. The model also relies on a number of parameters that have to be determined specifically for each environment in which it is applied.

Their overall contribution is to demonstrate how both underestimates and overestimates of project effort can lead to lower average productivity and increased overall effort. However, the published material includes only a small number of example projects from similar environment. This makes it hard to asses how accurate the model would be for projects from a wider range of environments.

In addition to those formal approaches described above, expert judgment is also recognized as an estimation method [18][48]. Experts may employ one or more of the other methods in making estimations, either informally or formally. It is likely that expert judgment is employed to make estimations whenever an expert is available. Expert judgment is not included in the framework for selecting estimation methods, as this method cannot easily be characterized, and it assumed that it is selected whenever experts are available.

It is hard to assess which method described above best suits for a software development project on hand. Of the methods described from developing models, empirical parametric method is some of the easiest to apply. The popular COCOMO II [21] model is based on this method. Empirical nonparametric methods such as an artificial intelligent neural network are hard to set up, because it involves more work than preparing a model based on a statistical regression [41]. Analogy based estimation is also straightforward to apply, provided only a small data set needs to be searched for analogs, and the number of variables to consider is no more than half a dozen. However, specific tools are needed to support to build the model based on analogy when the number of cases and variables increase [109] [41]. Moreover, how similar a new project development project is to historical projects also influences the selection

of method. If the new project differs from all historical projects, in a way that is recognized, then ideally an estimate should take this difference into account.

From the viewpoint of an organization's management and from the viewpoint of a customer, the most interesting software cost estimation measures are total effort and total duration, and once development is under way, the totals to complete. Individual developers are less likely to be interested in total effort estimates. They may want to track their own productivity, however, to make effort estimates for their own activities. For example, in some organizations, developers are expected to sign up to meet a target duration for a particular activity. Estimates based on group productivity figures generally will not be satisfactory, because of the significant variations commonly found between individual developers [37].

Estimates of total effort are clearly useful prior to or at the start of system development. However, this is the time relative to system development activities when there is the least information available on which to base an estimation. Especially, models that estimate total effort based on lines of code cannot give an accurate effort estimate due to lack of detailed information at this time.

Models for estimating total effort that are based on measures available early in the system development life cycle are clearly desirable. Models based on function points offer some improvement over lines of code, as it appears that function points can be estimated more consistently from specification and design descriptions than lines of code [74]. However, considerably more experience and effort is involved in counting function points than lines of code, so data pairs of total effort and function points are likely to be harder to obtain.

As initial software cost estimates are made based on limited information, re-estimating is desirable when additional information is available. Once system development is under way, the interest shifts from total effort to total effort to complete

development. For this measure, the re-estimate of total effort needs to take into account the actual progress that has been made so far, as well as the effort so far. For example, an initial estimate of the total effort to develop a system may be based on a rough, preliminary estimate of function points. Thus a new estimate may be calculated from a re-estimate of function points which is made after a high level design is complete. However, the model assumes the same average productivity for system development for both estimates. If the productivity of the development team is substantially different from that assumed by the model, the new total effort estimate will not incorporate this knowledge, and the estimate of total effort to complete the development also will not. Issues such as those complicate the process of re-estimation and indicate that measures that reflect actual progress will be important for accurate software cost estimation, once a project is under way. Estimation models need to incorporate these measures to estimate the total effort or time to complete successfully.

The environment for system development contributes factors such as targets and constraints. When a project starts there is often a target for delivery date and constraints on how many staff can ultimately be assigned to work on the project and on the availability of these staff. Estimates of these are needed to plan system development or check whether it is feasible to deliver within the desired time. Total effort, duration, and staffing are closely related and interdependent, but there may be independent constraints on all three. This makes the problem of estimating any one or two of them complex.

Existing parametric models such as COCOMO [18] and Putnam [88] have not proved widely successful in explaining the relationships among effort, duration, and staffing across a range of organizational settings. The dynamic model of Abdel-Hamid and Madnick [3] appears able to explain interrelationships among staffing, duration,

and overall cost in a qualitative way, but the model is not easy to apply, because it requires a specialized simulation tool.

Historical data are arguably the most important elements of an organization's experience base. The availability of historical data is critical in model development, as the measures that can be estimated are dictated by the measures for that data values already collected. Experience in developing and applying measures and models must also be cultivated within organizations, if the benefits of collecting local data are to be realized. The simplest models to develop and apply are empirical parametric models, with few variables, and analogical models. Models that are more difficult to develop and apply may be models based on a large number of variables such as Abdel-Hamid and Madnick [3].

We can discuss some limitations and difficulties found in software cost estimation approaches described so far. First, the lack of measurement within the software development environment is constraining accurate effort estimation. Second, most models for estimating total system development effort are in the focus of research area. Therefore, the models lack practicality in estimating effort for system development activities needed for planning and monitoring progress under process. Third, making accurate estimates using existing cost estimation models is difficult. Uncertainty is introduced because the model explains only part of the variation in effort. Developing models that are better at explaining this variation, and hence more accurate, is a great challenge. Forth, Uncertainty is also introduced where the values of input parameters cannot be measured. There are arguably too few models that are suitable for early life-cycle estimation [109]. Finally, an obvious difficulty with the cost estimation models is that the accuracy of models is not improving [109]. One of the reasons is that most practitioners take an informal approach to estimation that does not incorporate the feedback to improve the model in use. This highlights the need for a

software cost estimation process that incorporates feedback.

II.5 COCOMO II

COCOMO II is one of the popular software engineering cost models, which is based on the multiple regression approach. COCOMO II is a recent update of the COCOMO model published in 1981 [18]. COCOMO II provides two models, the Post-Architecture and Early Design models. The Post-Architecture is a detailed model that is used once the project is ready to develop and sustain a fielded system. The system should have a life-cycle architecture package, which provides detailed information on cost driver inputs, and model that is used to explore architectural alternatives or incremental development strategies. This level of detail is consistent with the general level of information available and the general level of estimation accuracy needed.

Both the Post-architecture and Early Design models use the same functional form to estimate the amount of effort and calendar time it will take to develop a software project. These nominal-schedule (NS) formulas exclude the Cost Driver for Required Development Schedule (SCED). The amount of effort in person-months, PM_{NS}, is estimated by the formula:

$$PM_{NS} = A \times Size^{E} \times \prod_{i=1}^{n} EM_i, \tag{2.6}$$

$$E = B + 0.01 \times \sum_{j=1}^{5} SF_j$$

E is the scaling exponent for the effort equation, and F scaling exponent for schedule.

The amount of calendar time, $TDEV_{NS}$, it will take to develop the product is estimated by the formula:

$$TDEV_{NS} = [C \times (PM_{NS})^{F}] \times \frac{SCED\%}{100}, \tag{2.7}$$

$$F = D + 0.2 \times [E - B],$$

where the values of A, B, C, and D are 2.94, 0.91, 3.67 and 0.28, respectively.

A good size estimate is very important for an effort estimation. Projects are generally composed of new code, code reused from other sources - with or without modifications - and automatically translated code. Size attributes are used to describe physical magnitude, extent or bulk. A size attribute can represent relative or proportionate dimensions. Software size attributes are classified as volume, structure, and rework. Volume attributes can be used to predict the amount of effort required to produce a software product, Defects remaining in a software product, and time required to create a software product.

There is no single volume attribute that should be applied by itself to measure the bulk of a software product. They should be used in combination to provide information related to controlling software projects and improving the software development process. The SLOC volume attribute is probably still the most widely used attribute because it is;

- relatively easy to define and discuss unambiguously,
- easy to objectively measure,
- conceptually familiar to software developers,
- used directly or indirectly by most cost estimation models and rules of thumb for productivity estimation, and
- is available directly from many organization's project databases.

However, Jones suggested several problems with the SLOC attribute as follows [60]:

- It does not accurately support cross-language comparisons for productivity or quality for the more than 500 programming languages in current use.
- There is no national or international standard for a source line of code.
- Paradoxically, as the level of language gets higher, the most powerful and advanced languages appear to be less productive than the lower level languages.

Even with these deficiencies, SLOC is still gathered by most metric programs.

Simmons et al. [99] introduced the Chunk metric. The intent is to measure software at the cognitive level at which software is developed. Chunks can be applied to objects, scripts, spreadsheets, graphic icons, application generators, etc.

Object points are similar to function points. They have the same advantages and disadvantages, but can be estimated and counted earlier than function points. Function points are based on functional requirements and can be estimated and counted much earlier than lines of code. Function points let organizations normalize data such as cost, effort, duration, and defects. Even though function points are a popular measure, they also have problems:

- They are based on a subjective measure which have resulted in a 30% variance within an organization and more than 30% across organizations [68].
- Function points behave well when used within a specific organization, but they do not work well for cross-company bench marking.

There are several sources for estimating new lines of code. The best source is historical data. For instance, there may be data that will convert function points, components, or anything available early in the project to estimate lines of code. Lacking historical data, expert opinion can be used to derive estimates of likely, lowest-likely, and highest-likely size.

Table 1. User function types

Function Point	**Description**
External Input (EI)	Count each unique user data or user control input type that enters the external boundary of the software system being measured
External Output (EO)	Count each unique user data or control output type that leaves the external boundary of the software system being measured
Internal Logical File (ILF)	Count each major logical group of user data or control information in the software system as a logical internal file type. Include each logical file (e.g., each logical group of data) that is generated, used, or maintained by the software system
External Interface File (EIF)	Files passed or shared between software systems should be counted as external interface file types within each system
External Inquiry (EQ)	Count each unique input-output combination, where input causes and generates an immediate output, as an external inquiry type

Code size is expressed in thousands of source lines of code (KSLOC). A source line of code is generally meant to exclude nondelivered support software such as test drivers. Defining a line of code is difficult because of conceptual differences involved in accounting for executable statements and data declarations in different languages. Difficulties arise when trying to define consistent measures across different programming languages. In COCOMO II, the logical source statement has been chosen as the standard line of code. The Software Engineering Institute (SEI) definition checklist

for a logical source statement is used in defining the line of code measure. The SEI has developed this checklist as part of a system of definition checklists, report forms and supplemental forms to support measurement definitions [78] [44].

The function points cost estimation approach is based on the amount of functionality in a software project and a set of individual project factors [14] [1]. Function points are useful estimators since they are based on information that is available early in the project life cycle. Function points measure a software project by quantifying the information processing functionality associated with major external data or control input, output, or file types. Five user function types should be identified as defined in Table 1.

Each instance of these function types is then classified by complexity level. The complexity levels determine a set of weights, which are applied to their corresponding function counts to determine the Unadjusted Function Points (UFP) quantity. This is the function points sizing metric used by COCOMO II. The usual function points procedure, which is not allowed by COCOMO II involves assessing the degree of influence (DI) of fourteen application characteristics on the software project determined according to a rating scale of 0.0 to 0.05 for each characteristic. The fourteen ratings are added together adjustment factor that ranges from 0.65 to 1.35.

Each of these fourteen characteristics, such as distributed functions, performance, and reusability, thus have a maximum of 5% contribution to estimated effort. Having, for example, a 5% limit on the effect of reuse is inconsistent with COCOMO experience; thus COCOMO II uses Unadjusted Function Points for sizing, and applies its reuse factors, cost drivers, and scale factors to this sizing quantity to account for the effects of reuse, distribution, etc. on project effort. The four steps of counting procedure are as follows:

Table 2. FP complexity levels

For Internal Logical Files and External Interface Files			
	Data Elements		
Record Elements	1-19	20-50	51+
1	Low	Low	Avg.
2-5	Low	Avg.	High
6+	Avg.	High	High
For External Output and External inquiry			
	Data Elements		
Record Elements	1-5	6-19	20+
0 or 1	Low	Low	Avg.
2-3	Low	Avg.	High
4+	Avg.	High	High
For External Input			
	Data Elements		
Record Elements	1-4	5-15	16+
1	Low	Low	Avg.
2-3	Low	Avg.	High
3+	Avg.	High	High

Determine function counts by type The unadjusted function counts should be counted by a lead technical person based on information in the software requirements and design documents. The number of each of the five user function types should be counted [Internal Logical File (ILF), External Interface File (EIF), External Input (EI), External Output (EO), and External Inquiry (EQ)].

Determine complexity levels Classify each function count into Low-, Average- and high-complexity levels depending on the number of data element types contained and the number of file types referenced. Use the scheme in Table 2.

Apply complexity weights Weight the number of function types at each complexity level using the scheme in Table 3. (the weights reflect the relative effort

Table 3. UFP complexity weights

Function Type	Complexity-Weight Low	Average	High
Internal Logical Files	7	10	15
External Interface Files	5	7	10
External Inputs	3	4	6
External Outputs	4	5	7
External Inquiries	3	4	6

required to implement the function).

Compute Unadjusted Function Points Add all the weighted functions counts to get one number, the Unadjusted Function Points.

Next, convert the Unadjusted Function Points (UFP) to lines of code. The UFP has to be converted to source lines of code in the implementation language (Ada, C, C++, Pascal, etc.). COCOMO II does this both for both the Early Design and Post-Architecture models by using tables to convert UFP into equivalent SLOC. The current conversion ratios are shown in Table 4 [60].

In addition to the newly built code, code that is taken from another source and used in the product under development also contributes to the product's effective size. Reusable code is composed of code that is reused without modification, and adapted code that is used with modification. New code equivalent size of SLOC can be obtained by adjustment of reused and adapted code. In COCOMO II, Boehm et. al. [21] suggest a reuse model.

$$EquivalentKSLOC = AdaptedKSLOC \times \left(1 - \frac{AT}{100}\right) \times AAM, \tag{2.8}$$

$$AAM = \frac{[AA + AAF(1 + (0.02 \times SU \times UNFM))]}{100}, AAF \leq 50, \tag{2.9}$$

Table 4. Default UFP to SLOC conversion ratios

Language	SLOC/UFP	Language	SLOC/UFP
Access	38	Jovial	107
Ada 83	71	Lisp	64
Ada 95	49	Machine Code	640
APL	32	Pascal	91
Assembly-Basic	320	PERL	27
Basic-ANSI	64	Prolog	64
Basic-Visual	32	Report Generator	80
C	128	2nd Generation Lang.	107
C++	55	Simulation-Default	46
Database-Default	40	3rd Generation Lang.	80
5th Generation Lang.	4	Unix Shell Scripts	107
1st Generation Lang.	320	USR_1	1
Fortran 95	71	USR_4	1
4th Generation Lang.	20	USR_5	1
High Level Lang.	64	Visual Basic 5.0	29
HTML 3.0	15	Visual C++	34
Java	53		

$$AAM = \frac{[AA + AAF(SU \times UNFM))]}{100}, AAF > 50, \tag{2.10}$$

where AA represents assessment and assimilation, AAM adaptation adjustment modifier, AT amount of automatic translated, SU software understanding, $UNFM$ programmer unfamiliarity.

$$AAF = (0.4 \times DM) + 0.3 \times CM) + 0.3 \times IM) \tag{2.11}$$

Adaptation adjustment modifier (AAF) contains the quantities such as percent design modified (DM), percent code modified (CM), and percent of integration required

for adapted software (IM). While reusable code saves much effort, it still requires effort such as understanding the software to be modified, and checking inter module interfaces.

To aggregate the new, adapted and reused code, COCOMO II provide the sizing equation as follows:

$$Size = \left(1 + \frac{REVL}{100}\right) \times (NewKSLOC + EquivalentKSLOC), \tag{2.12}$$

where $REVL$ is percentage of requirements evolution and volatility.

The effort of personnel is the main cost in a software development project. Effort of an organization is the person months required to produce a specific size of a software product. Of all effort estimation models, COCOMO is the most complete and thoroughly documented model. Boehm et al. [21] introduces the COCOMO 2.0 effort estimation model.

$$Effort = A \times Size^{E} \times \prod_{i=1}^{n} \times EM_i, \tag{2.13}$$

where $Size$ is KSLOC, $A = 2.94$, E is an aggregation of scale factors, and EM is effort multipliers. The unadjusted function points (UFP) can be used if converted to $KSLOC$.

The scale factors are: Precedentedness (PREC), Development Flexibility (FLEX), Architecture/Risk Resolution (RESL), Team Cohesion (TEAM), and Process Maturity (PMAT). The exponent E in Equation 2.13 is an aggregation of five scale factors that account for the relative economies or diseconomies of scale encountered for software projects of different sizes. If $E < 1.0$, the project exhibits economies of scale. If the product's size is doubled, the project effort is less than doubled. The project's productivity increases as the product size is increased. Some project economies of scale can be achieved via project-specific tools (e.g., simulations, testbeds), but in

general these are difficult to achieve. For small projects, fixed start-up costs such as tool tailoring and setup of standards and administrative reports are often a source of economies of scale.

If $E = 1.0$, the economies and diseconomies of scale are in balance. This linear model is often used for cost estimation of small projects.

If $E > 1.0$, the project exhibits diseconomies of scale. This is generally because of two main factors: growth of interpersonal communications overhead and growth of large-system integration overhead. Larger projects will have more personnel, and thus more interpersonal communications paths consuming overhead. Integrating a small product, but also the additional overhead effort to design, maintain, integrate, and test its interfaces with the remainder of the product.

There are seventeen EM grouped in four factors: product factors, platform factors, personnel factors, and project factors. They are used to adjust the nominal effort, PM, and to reflect the software product under development. The product factors account for variation in the effort required to develop software caused by characteristics of the product under development. A product that is complex, has high-reliability requirements, or works with a large testing database will require more effort to complete. There are five effort multipliers in the product factors, and complexity has the strongest influence on estimated effort. The effort multipliers are Required Software Reliability (RELY), Database Size (DATA), Product Complexity (CPLX), Developed for Reusability (RUSE), and Documentation Match to Life-Cycle Needs (DOCU).

The platform refers to the target-machine complex of hardware and infrastructure software. There are three effort multipliers in the platform factors: Execution Time Constraints (TIME), Main Storage Constraint (STOR), and Platform Volatility (PVOL).

The personnel factors are for rating the development team's capability and experience - not the individual. These ratings are most likely to change during the course of a project reflecting the gaining of experience or the rotation of people onto and off the project. There are five effort multipliers: Analyst Capability (ACAP), Programmer Capability (PCAP), Personnel Continuity (PCON), Applications Experience (APEX), Platform Experience (PLEX), and Language and Tool Experience (LTEX).

The project factors account for influences for influences on the estimated effort such as use of modern software tools, location of the development team and compression of the project schedule. There are three effort multipliers: Use of Software Tools (TOOL), Multisite Development (SITE), and Required Development Schedule (SCED).

II.6 Commercial Off the Shelf (COTS) Components

COTS components are an increasingly influential factor to the productivity, and widely used in current software projects. It is not likely that a single large software system is being designed without the incorporation of at least one COTS component. COTS components have several characteristics such as: the COTS source code cannot be accessed; the vendor has the control of the COTS software life cycle. Although COTS components reduce the development effort, integrating COTS components involves activities such as assessment, tailoring, and developing glue code. Basili et al. [12] report effort required for COTS based development.

Determining the use of and how many of COTS components are used should affect the productivity. The COTS approach still requires effort such as selecting, installing and configuring to the system but less effort than approach to build system

Table 5. Effort required for COTS based development

Activity	**Average Effort(%)**	**Standard Deviation(%)**
Glue Code	37	± 36
Tailoring	26	± 30
Assessment	24	± 20
Volatility	13	± 11

entirely from scratch. While COTS approach seems to have advantage of reducing development cost, amount of COTS components can affect the economic life of the SP. After including COTS components in a system, they can disappear or evolve in different directions in response to the market demand. As a result, the system depending on the COTS components should be replaced or developed. Abts [5] proposed COTS functional density (CFD) metric to solve the problem of maximizing the amount of functionality in the system provided by COTS components but using as few COTS components as possible.

$$CFD = \left(\frac{CFP}{NCFP + CFP}\right) \times \frac{1}{NCC}, \tag{2.14}$$

where CFP is COTS function points, $NCFP$ is non COTS function points, and NCC is number of COTS components in the system. CFD represents the percentage of overall system functionality delivered per COTS component. The implication is that the larger the CFD, the greater the "efficiency" of a given COTS based system design.

As seen in Table 5, glue code accounts for less than half the total COTS based development effort. However, developing glue code consumes effort three times more than developing same amount of new code. More investment in assessment, therefore, could reduce the total cost for integrating COTS components.

CHAPTER III

BAYESIAN ANALYSIS

III.1 Statistical Analysis

The main purpose of statistical analysis is to derive from observations of a random phenomenon an inference about the probability distribution underlying this phenomenon [26]. That is, it is comprised of two steps: building a probabilistic model based on observed phenomenon; and predicting a future phenomenon of a similar nature with the model. This type of reasoning is called inductive logic or plausible reasoning. Statistical inference obtains conclusions from the data to answer: estimation of a parameter value; testing a hypothesis about the statistical model; and prediction. Figure 9 shows the concept of statistical analysis. However, statistical analysis is not deduction, which means that the conclusions made are subject to error, even when one has accounted for the possibility of error.

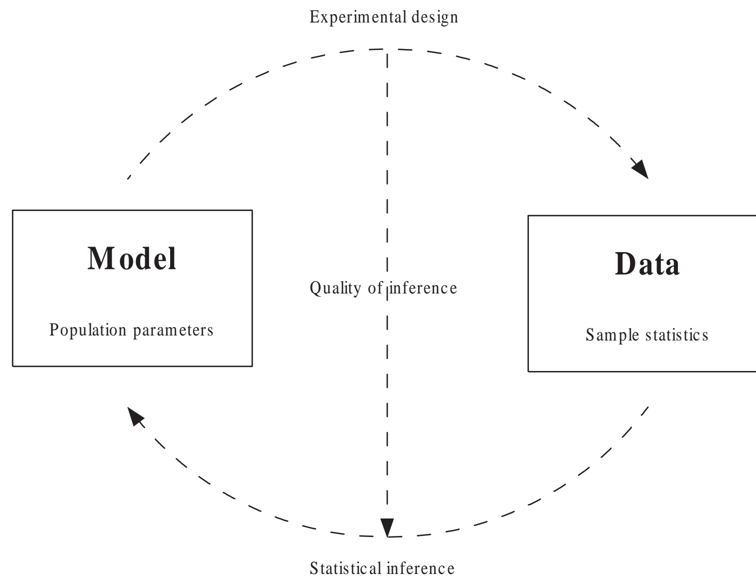


Fig. 9. Statistical analysis

Statistical analysis has two approaches to solve the real world problem. The first approach assumes that statistical analysis must incorporate as much as possible of the real world complexity, and thus aims at estimating the distribution underlying the phenomenon under minimal assumptions. This approach is called nonparametric. Conversely, the parametric approach represents the distribution of the observations through a distribution function $f(x|\theta)$, where only the parameter θ (of finite dimension) is unknown. The second approach is more pragmatic, since it takes into account that a finite number of observations can efficiently estimate only a finite number of parameters. Moreover, a parametric modeling authorizes an evaluation of the inferential tools for finite sample sizes.

Once the statistical model is defined, the main purpose of the statistical analysis is to lead to an inference on the parameter θ. This means that observation x is used to improve the knowledge on the parameter θ, so that one can take a decision related with this parameter, i.e., either estimate a function of θ or a future event whose distribution depends on θ. The inference can deal with some components of θ, precisely "What is the value of θ_1?" or "Is θ_2 larger than θ_3?". More generally, inference covers the random phenomenon directed by θ and thus includes prediction, that is, the evaluation of the distribution of a future observation y depending on θ (and possibly the current observation x), $y \sim g(y|\theta, x)$. Indeed, the ultimate goal of statistical analysis is, in the overwhelming majority of cases, to support to a decision as being optimal (or at least reasonable).

Compared with probability modeling, the purpose of a statistical analysis is fundamentally an inversion purpose, since it aims at retrieving the causes - reduced to the parameters of the probabilistic generating mechanism - from the effects - summarized by the observations. In other words, when observing a random phenomenon directed by a parameter θ, statistical methods allow to deduce from these observa-

tions an inference (that is, a summary, a characterization) about θ, while probabilistic modeling characterizes the behavior of the future observations conditional on θ. A general description of the inversion of probabilities is given by Bayesian theorem [16]: if A and E are events such that $P(E) \neq 0$, $P(A|E)$ and $P(E|A)$ are related by

$$P(A|E) = \frac{P(E|A)P(A)}{P(E|A)P(A) + P(E|A^c)P(A^c)} = \frac{P(E|A)P(A)}{P(E)}. \tag{3.1}$$

The equation expresses the fundamental fact that, for two equiprobable causes, the ratio of their probabilities given a particular effect is the same as the ratio of the probabilities of this effect given the causes. This theorem also is an actualization principle since it describes the updating of the likelihood of A from $P(A)$ to $P(A|E)$ once E has been observed. Thomas Bayes (1764) actually proved a continuous version of this result, namely, that given two random variables x and y, with conditional distribution $p(y|x)$ and marginal distribution $p(x)$, the conditional distribution of x given y is

$$p(x|y) = \frac{p(y|x)p(x)}{\int p(y|x)p(x)dx}. \tag{3.2}$$

While this inversion theorem is quite natural from a probabilistic point of view, Bayes and Laplace went further and considered that the uncertainty on the parameters θ of a model could be modelled through a probability distribution, called prior distribution. The inference is then based on the distribution of θ, $p(\theta|y)$, called posterior distribution and defined by

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{\int p(y|\theta)p(\theta)d\theta}. \tag{3.3}$$

From the equation, we can notice that $p(\theta|y)$ is actually proportional to the distribution of y conditional upon θ, i.e., the likelihood, multiplied by the prior dis-

tribution of θ. The main addition brought by a Bayesian statistical model is thus to consider a probability distribution on the parameters.

III.2 Bayesian Analysis

Bayesian analysis means practical methods for making inferences from data using probability models for quantities to observe and for quantities to know. The essential characteristic of Bayesian methods is the explicit use of probability for quantifying uncertainty in inferences on statistical data analysis. The process of Bayesian data analysis has following steps:

1. Setting up a full probability model
2. Conditioning on observed data
3. Evaluating the fit of the model and the implications of the resulting posterior distribution

In statistical terms, Bayes' Theorem actualizes the information on θ by extracting the information on θ contained in the observation y. Bayesian statistical conclusions about a parameter θ, or unobserved data y, are made in terms of probability statements. These probability statements are conditional on the observed value of y, and are written as $p(\theta|y)$. It is at the fundamental level of conditioning on observed data that Bayesian analysis departs from the approach to statistical analysis, which is based on a retrospective evaluation of the procedure used to estimate θ over the distribution of possible y values conditional on the true unknown value of θ [91].

Bayesian analysis provides a mechanism for updating initial probability statements about parameters with the sample data observed [43].

$$p(\theta|y) \propto p(y|\theta)p(\theta), \tag{3.4}$$

where $p(\theta)$ is the prior distribution, and $p(y|\theta)$ is the sampling distribution. The posterior distribution $p(\theta|y)$ is proportional to the product of prior and sampling distribution. Bayesian inference on a parameter θ is, therefore, conditional on observed sample data y. Typically, Bayesian analysis implies that the inference on θ should rely entirely on the posterior distribution $p(\theta|y)$. Even though θ is not necessarily a random variable, the posterior distribution $p(\theta|y)$ can be used as a regular probability distribution to describe the properties of θ: summarizing indices of the posterior mean; the posterior mode; the posterior median; and the posterior variance [91].

The prior distribution is used to summarize the available information about the parameters (or even lack thereof), as well as the residual uncertainty, thus allowing for incorporation of this imperfect information in the decision process. The prior distribution is the most critical point of Bayesian analysis [16]. The prior distribution is unconditional to the sampling data, while the posterior distribution is conditional to the sampling data and prior information. And this is the key point of Bayesian analysis, since once this prior distribution is known, inference can be led in an almost mechanic way afterwards. However, in practice, it seldom occurs that the available prior information is precise enough to lead to an exact determination of the prior distribution. The following questions may then be asked frequently regarding on the prior distribution.

- Where do the models come from?
- How can we go about constructing appropriate probability specifications?

The difficulty with the choice of the prior distribution comes from that the decision maker, the client or the statisticians do not have the time or resources to hunt for an exact prior and they have to complete the partial information with a subjective input to build a prior distribution. Therefore, it is often to make a partly arbitrary

choice of the prior distribution which leads to drastic change of the subsequent inference. In particular, the systematic use of parametrized distributions such as normal, gamma, beta, etc. and the further reduction to conjugate distributions can be recommended. Some settings nonetheless call for a partly automated determination of the prior distribution when prior information is totally lacking. However, they can trade an improvement in the analytical treatment of the problem for the subjective determination of the prior distribution and therefore ignore part of the prior information. The main points about the prior distribution are as follows:

- Ungrounded prior distribution produce unjustified posterior inference
- There is no such thing as the prior distribution except for very special settings

In Bayesian analysis, the subjectivity issue is always critical, because of the reliance on a prior distribution. However, all statistical methods that use probability are subjective in the sense of relying on mathematical idealizations of the world, and most problems in the science field demand scientific judgement which are subjective in terms of the likelihood [91]. From a philosophical point of view, it is generally agreed that knowledge stems from a confrontation between *a prioris* and experiments. This point of view is found in Poincaré [83]:

> It is often stated that one should experiment without preconcieved ideas. This is simply impossible; not only would it make every experiment sterile, but even if we were ready to do so, we could not make implement this principle. Everyone stands by his own conception of the world, which he cannot get rid of so easily.

Thomas Kuhn also discusses the point of view [69]:

> Some accepted examples of actual scientific practice provide models from which spring particular coherent traditions of scientific research.

In fact, without *a prioris*, that is, without a pre-established structure of the world, observation is meaningless because it does not come as a support of or as a confrontation to a referential model. Therefore, the building of knowledge through experimentation implies the existence of a prior representation system, which is very primitive at the beginning, but gets progressively actualized via these experiments. Bayesian analysis is obviously in accordance with this perspective, since prior distributions are most often based on the results of previous experiments.

The posterior distribution represents the state of knowledge about the truth of the parameter in the light of the data. This distribution operates conditional upon the observations. It thus avoids averaging over the unobserved values of y, which is the essence of the frequentist approach [91]. Indeed, the posterior distribution is the updating of the information available on θ, owing to the information contained in the sampling distribution, while the prior distribution represents the information available *a priori*, that is, before observing y. Furthermore, Bayesian analysis has two advantages: the order in which i.i.d. (identically independently distributed) observations are collected does not matter; updating the prior one observation at a time, or all observations together, does not matter. Therefore, Bayesian analysis can perform sequential analysis. The power of Bayesian analysis lies in the fact that it encapsulates the process of learning, i.e., the prior information is transformed to posterior distribution. And the posterior distribution can be used as a prior distribution to construct a new model when new data are coming.

In summary, Bayesian analysis is the process of fitting a probability model to a set of data and summarizing the result by a probability distribution on the parameters

of the model and on unobserved quantities such as predictions for new observations.

III.3 Model Assumption

The normal distribution is central to statistical inference and modeling. Undoubtedly, the most widely used model for the distribution of a random variable is a normal distribution. The characteristics of the normal distributions are having two parameters, the mean μ as a measure of location, and the variance σ^2 measuring the extent of scatter around that central location.

$$\theta \sim N(\mu, \sigma^2) \tag{3.5}$$

The normal distribution has several advantages such that distributions are very tractable analytically, symmetry bell shape makes it an appealing choice for many population models, and the distribution can be used to approximately a large variety of distributions in large samples under the central limit theorem.

The central limit theorem states that whenever a random sample of size of n is taken from any distribution with mean μ and variance σ^2, the sample mean $\overline{y}$ will have a distribution which is approximately normal with mean μ and variance σ^2/n [26]. The central limit theorem helps further to justify the normal distribution as an approximation for the posterior distribution of many summary statistics, even those deriving from non-normal data, as sample size increases.

III.4 Multi-Parameter Estimation

Virtually every practical problems involve more than one unknown parameter or unobservable quantity. Although a problem can include several parameters of interest, conclusions will often be drawn about one, or only a few, parameters at a time. In this

case, we need to obtain the marginal posterior distribution of the particular parameters of interest [43]. In principle, we first construct the joint posterior distribution of all unknowns, and then we integrate this distribution over the unknowns that are not of immediate interest to obtain the desired marginal distribution. Other than the particular parameters, the others are called nuisance parameters.

Suppose we are interested in two parameters θ_1 and θ_2. The joint posterior distribution will be

$$p(\theta_1, \theta_2|y) \propto p(y|\theta_1, \theta_2)p(\theta_1, \theta_2) \tag{3.6}$$

And we can get marginal distribution of θ_1 by averaging over θ_2.

$$p(\theta_1|y) = \int p(\theta_1, \theta_2|y)d\theta_2 \tag{3.7}$$

This is the final form of the marginal posterior distribution of the parameter of interest given the sampling data.

CHAPTER IV

DEVELOPMENT OF PAMPA II

IV.1 Overview of PAMPA II

In 1997, PAMPA (Project Attributes Monitoring and Prediction Associate) tool was developed to help managers gather project information from any software development environment, save it in an understandable object/attribute/relationship format, view it using an inexpensive workstation, and supply input to expert system building tools used for creating intelligent agents. A version of PAMPA tool is available in the book on software measurement by Simmons et al [99]. PAMPA runs in the Microsoft Windows and Office environment and uses an expert system building tool for creating intelligent agents. PAMPA can gather project information from any software development project that can share directories over a network to a Microsoft Windows client workstation. New features have been added to PAMPA to create the prototype version of PAMPA II. The main new capability is that it operates from Internet browsers and uses a three-tier architecture as shown in Figure 10. PAMPA II is a tool to help a manager view projects by gathering project attributes and presenting project status.

As the complexity of software development environment increases, Computer-Aided Software Engineering (CASE) tools such as MS Project, Rational Rose, RequisitePro, ClearCase, ClearQuest, and Test Studio are used to support developers. PAMPA II has evolved to gather critical project attributes from the CASE tool databases, store them into PAMPA II KB, and provide status of a project via web-based consoles.

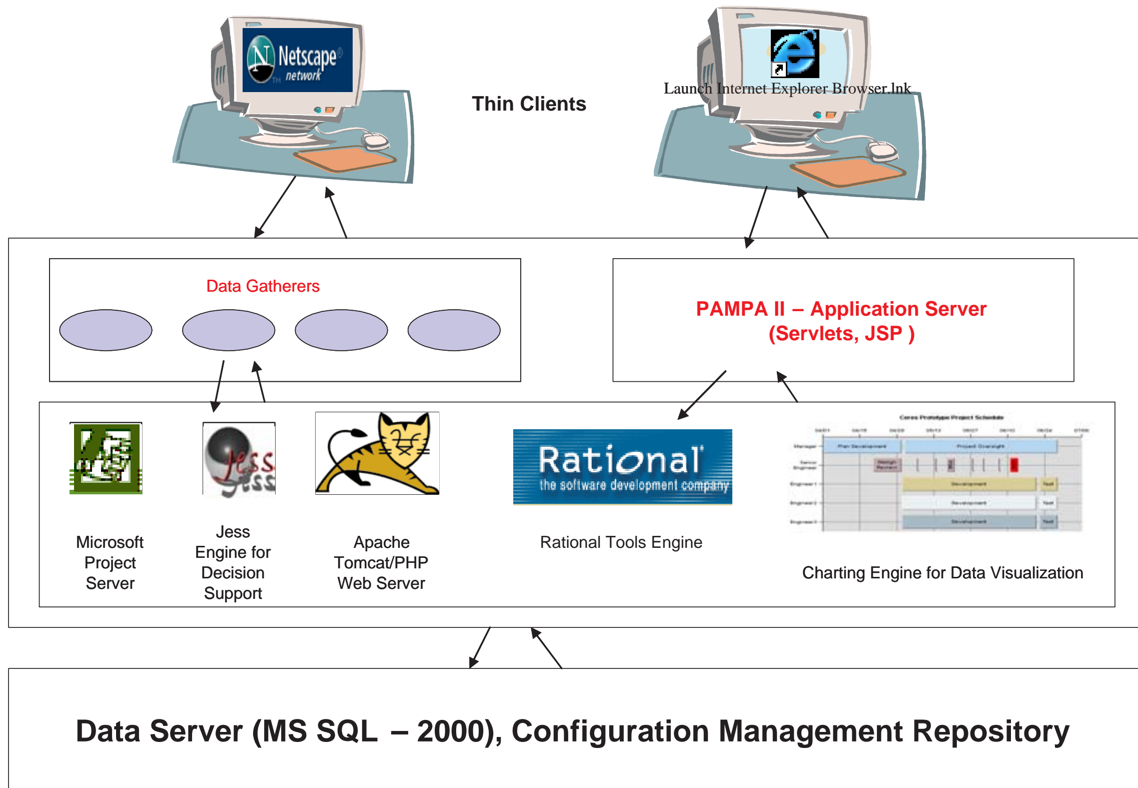


Fig. 10. Overview of PAMPA II system architecture

IV.2 Framework of PAMPA II

PAMPA II expands the PAMPA KB to include a project version, plan, and milestone object classes [100]. The Unified Modelling Language (UML) [22] is used to describe object classes, attributes, and relationships. The objects that comprise a software **Project** are displayed in Figure 11. In this chapter, names of objects such as **Project** are written using a Arial bold font. A KB reflecting all attributes and relationships of a **Project** can be constructed to reflect **Project** status at all stages of development. The **ProjectList** is made up of knowledge bases of all **Project**s being tracked. The fact that there is a single **ProjectList** is shown by the number one (1) next to the

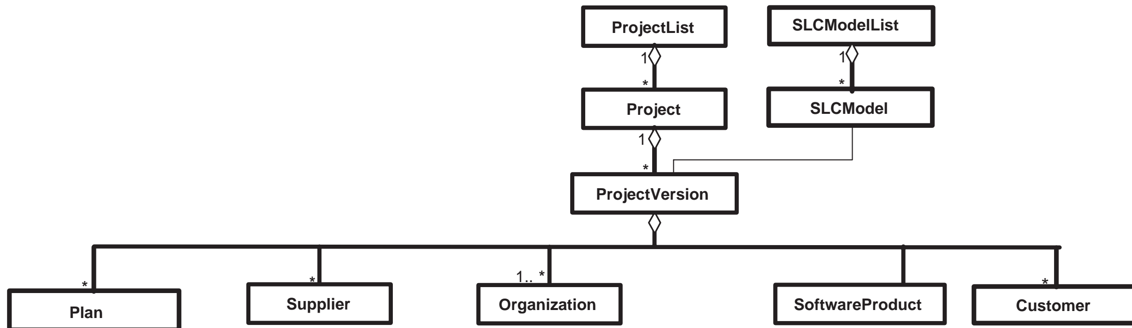


Fig. 11. Project

interconnecting line between **ProjectList** and **Project**. The asterisk adjacent to the line above **Project**s indicates that there are zero or more **Project**s in the list. The objects that make up a **Project** continually change with the passage of time. **ProjectVersions** are archived at selected times during a development. Snapshots of all aspects of a **Project** can be replayed in a manner similar to how airline flight recorders replay flight data to determine what happened during a flight before a plane crashes.

The diamond symbol below **ProjectVersions** in Figure 11 indicates that the objects connected by the bold line are an aggregation that comprises a **ProjectVersion**. Each **ProjectVersion** is composed of zero or more **Plan**s, zero or more **Supplier**s of reusable software, one or more **Organization**s staffed by **Project** personnel, a **SoftwareProduct** that is being created by the **Project** and **Customer**s that will use the **SoftwareProduct**.

A **Plan** is shown in Figure 12. The thin line connecting the bottom side of the **Plan** object to the top side of the **Plan** object indicates that the different **Plan**s are related to each other. A **Plan** contains **Process**es and desired **Activity**(ies).

A **Process** can be related to other **Processes** and is made up of an aggregation

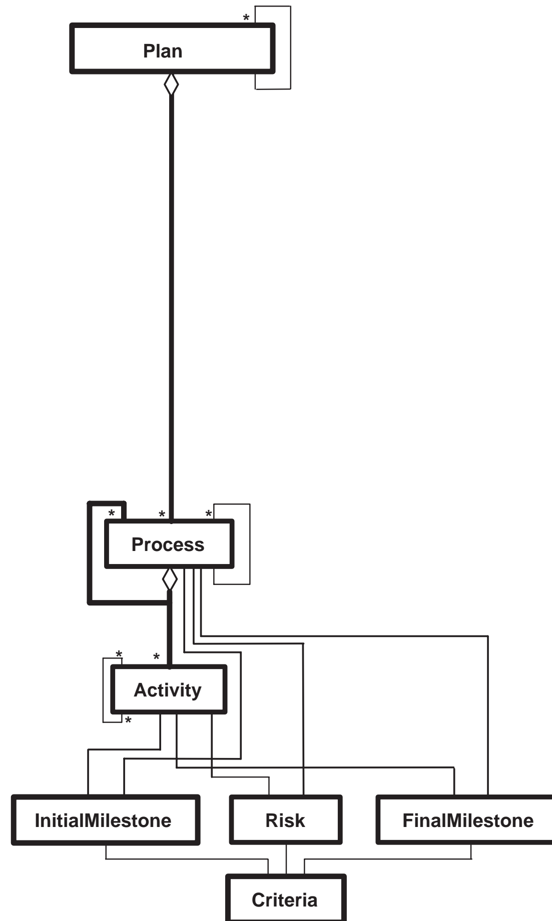


Fig. 12. Plan

of other **Process**es and **Activity**(ies) as shown in Figure 12. A **Process** begins with an **InitialMilestone** and ends with a **FinalMilestone**. As part of planning, **Risk** attributes are assessed for each **Process**. Each **Activity** also has an **InitialMilestone**, a **FinalMilestone**, and **Risk** attributes.

The **Supplier** shown in Figure 13 provides commercial off the shelf (COTS) software or reusable software from software reuse libraries found within **Organization**s. The first is supplied as **COTSRunFile**s and the second is supplied as

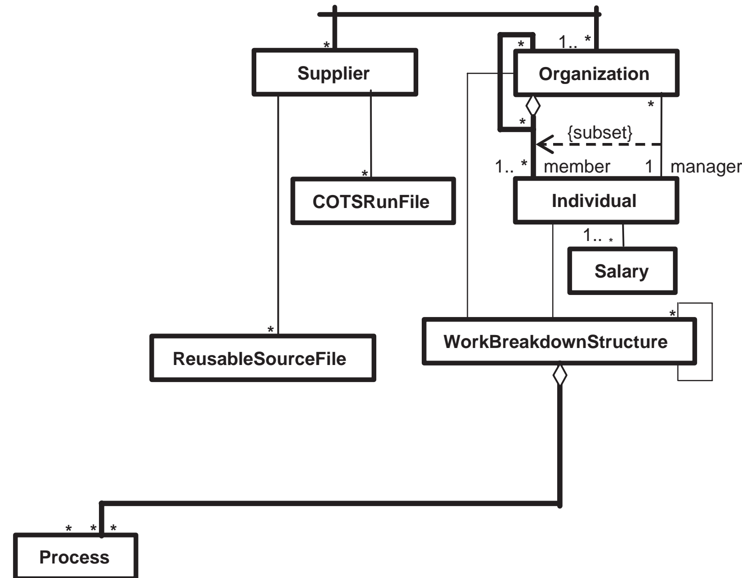


Fig. 13. Work breakdown structure

ReusableSoftwareFiles. Both of these are provide **SoftwareProduct Feature**s. Their relationships to **Feature**s are shown by the named relationship "is related to" in Figure 15.

The **Organization** structure is shown in Figure 13. An **Organization** is composed of other **Organization**s or of **Individual**s. An **Organization** has one or more **Individual**s and has a **WorkBreakdownStructure** assigned to it. At least one of the **Individual**s is the manager of the **Organization**. Each **Individual** has one (1) or more **Salary**(ies) and is assigned a **WorkBreakdownStructure**.

The **WorkBreakdownStructure**s are usually related to other **WorkBreakdownStructure**s and are composed of **Process**es as shown in Figure 13. A **WorkBreakdownStructure** is a hierarchy of elements that decomposes the **Plan** into the discrete work **Process**es or **Activity**(ies). Each **WorkBreakdownStructure** provides a clear task decomposition information for assignment of responsibilities. It

is the baseline for plan scheduling, budgeting, and plan tracking.

A **SoftwareProduct** is created by a **Project**. In Figure 14, we show that **SoftwareProduct**s are composed of the **Feature**s described in the requirements documents, the **Defect**s tracked by a defect tracking system, and the different **Version**s that are built during development.

A **Version** is composed of **Subsystem**s, **VandVTest**s, and **UsabilityTest**s, as described in Figure 14. The **VandVTest**s (verification and validation tests) are often managed using a test management system. **UsabilityTest**s are people intensive and are conducted in usability test cells. The attributes resulting from **UsabilityTest**s are saved as **Usability** attributes.

A **SoftwareProduct Subsystem** is composed of **Artifact**s as shown in Figure 14. **Artifact** stores the artifact information: artifact type, file name, directory, and programming to develop the artifact. An **Artifact** is composed of smaller cognitive **Chunk**s which can be composed of other **Chunk**s. Examples of chunks of code are subroutines, functions, packages, spreadsheets, query commands, and scripts. **Chunk**s are measured in terms of **Volume** and **Structure** attributes. **Volume** attributes are measured in units of source lines of code, function points, logical source statements, object points and unique source lines of code [99].

Named relationships between **Individual**s are shown in Figure 15. **Process** improvement relies upon empowering **Individual**s to help improve a **Process**. Along with empowerment, process improvement requires individuals to be accountable. Empowerment and accountability can be evaluated and tracked by expressing the ownership relationship. **Individual**s are shown to own each **ReusableSourceFile**, **COTSRunFile**, **Subsystem**, **VandVTest**, **UsabilityTest**, **Artifact**, **Feature**, **Defect**, **Version** and **SoftwareProduct**. The **Individual** that owns an object is the person responsible for it. Also, the **Individual** that authors an object is tracked. **Individid-**

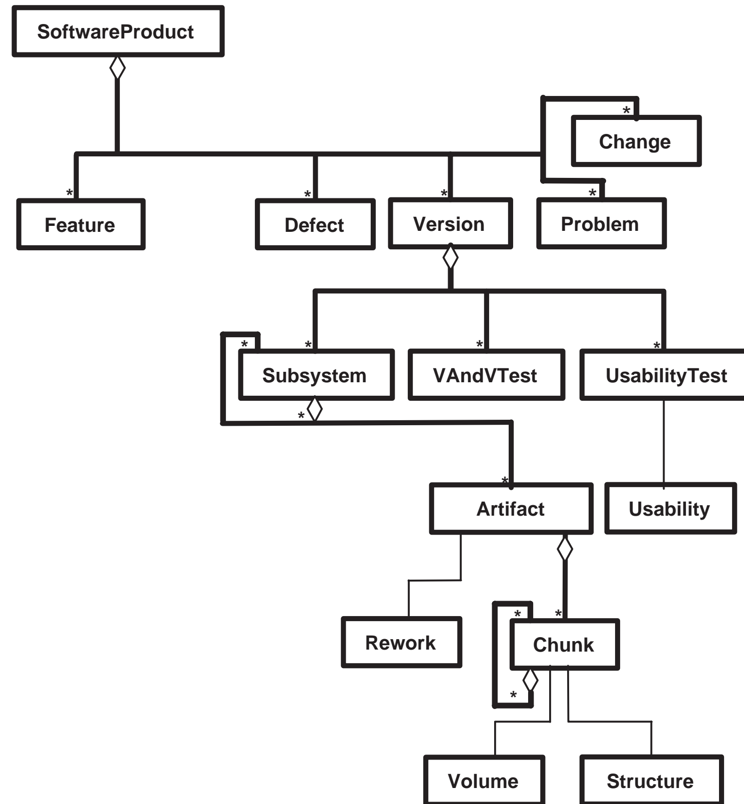


Fig. 14. Software product

uals are shown as authors of **Subsystem**s, **Artifact**s, **VandFTest**s and **UsabilityTest**s. An **Individual** runs **VandVTest**s and **UsabilityTest**s.

IV.3 System Architecture of PAMPA II

PAMPA II uses the three-tier architecture described in Figure 16. The first tier contains thin clients, the second tier contains a middleware server, and the third tier contains a database server. The three-tier architecture enhances separation of business logic from the graphical user interface (GUI) and database, and improves security, performance, and reliability. The first tier communicates with the manager

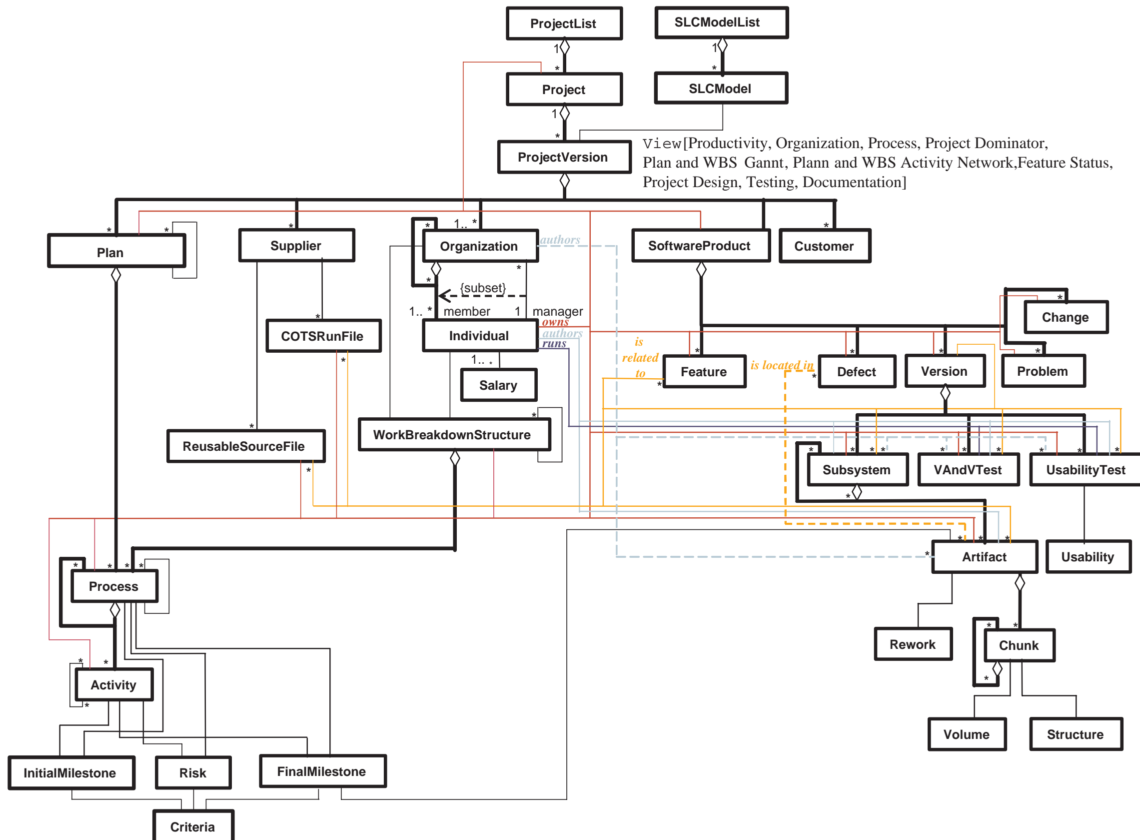


Fig. 15. Knowledge base framework and relationship

and developer workstations. Each workstation represents thin clients that contain only a web browser and a Java virtual machine. Java applets operate in the clients. The middleware server application runs on PHP server. The first-tier client application of PAMPA II is a Java applet, which is served by the second-tier middleware server and downloaded to the first-tier browsers for execution. The advantage of a Java applet over HTML is that it provides a cleaner and friendlier user interface with more powerful functions. In addition, the security restrictions on Java applets make them safe (no viruses, they can not write to a hard disk, etc.) to run. As a Java applet can only make network connection to the middleware server from which the

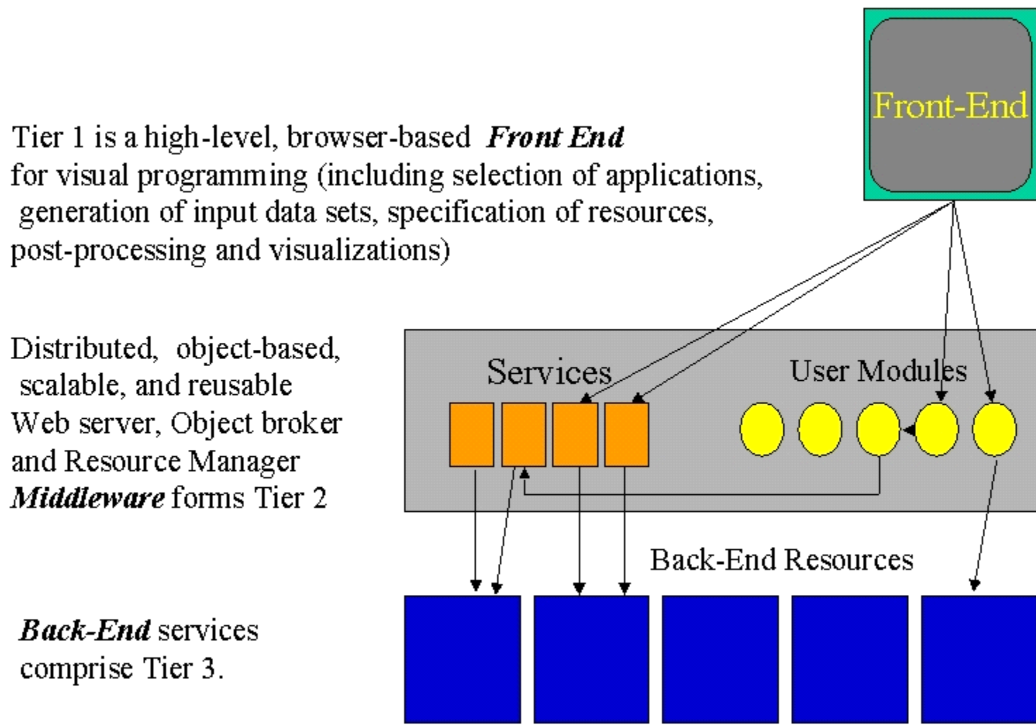


Fig. 16. Three-tier architecture

applet is downloaded, it can not communicate directly with the database.

The second-tier houses the middleware server. PAMPA II system and Java Expert System Shell (JESS) operate on the Middleware server. The second-tier communicates with the third-tier database servers via Java Database Connectivity (JDBC). Database files are stored in relational database management systems (RDBMS). A plan, organization, work breakdown structure, software product, and project knowledge base all reside on third-tier database servers. Currently, we are using a centralized relational MS SQL 2000 server to store our databases, but any RDBMS could be used.

Recently, more and more businesses have turned to three-tier architecture instead

Table 6. Subsystems of PAMPA II

Subsystems	**Primary features**
Application/Data server	Project project attributes and draw charts Store project attributes
Plan gatherer	Transfer project plan from MS project
Software metric parser	Calculate volume from source files
Data gatherer	Rational tools

of two-tier architecture because three-tier architecture offers clearer logic, better security and reliability. In a three-tier system, the application logic (the core) is properly separated from the user interface on the client side and the persistence domains on the server side. This separation makes code more portable.

IV.4 Subsystems of PAMPA II

Features have been developed to help managers of software project from those manned by a small team at one location to those with many teams dispersed all over the world. PAMPA II includes the following subsystems specified in Table 6. A prototype version of PAMPA II has been developed to run on the Internet. The following subsystems are also included in the prototype version. Figure 17 shows the outline of the system.

PAMPA II was created based on Internet technology. The Apache web server was used for HTTP service. We used the Tomcat and PHP engines to support script language capability to process data. JpGraph engine was used to draw charts with the project attributes. The engine is based on PHP technology. Jess is a rule engine and scripting environment written entirely in Sun's Java language, which provides a technology to create an intelligent agent for decision support. Database engine is

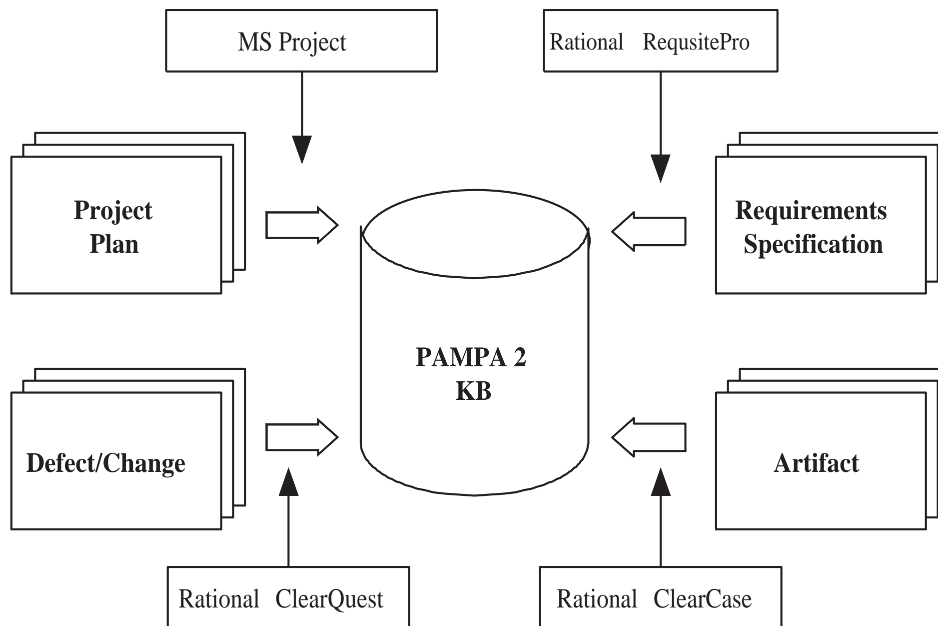


Fig. 17. Outline of the system

the MS SQL 2000 server, which is a typical RDBMS and supports Structured Query Language (SQL).

Plan gatherer transfers plan, process, activity and resource attributes from MS Project and stores them into the PAMPA II KB. MS Project is a software package tool to make a project plan. A project plan consists of activities, work breakdown structure (WBS), resource information such as skill, salary, and experience.

The software metric parser takes the source files as inputs from the data transformation module and parse them into tokens. By analyzing tokens, the software metric parser can come up with results of software metrics. The metrics of source files are indications of how a software project is going. Then, the parser can generate metrics stored in a database and presented with a GUI that allows the manager to gather and parse a particular version of a project. After gathering files from a particular version of a working directory, the GUI invokes its parser to start generating metrics.

When the parsing is complete, the manager is prompted and can, in turn, store the data into the PAMPA II KB. The software metrics are used to identify the current progress of the on-going project.

The Software metric parser provides some of the popular metrics from the literature, such as following:

- Bytes
- Source Line of Code (SLOC)
- Unique SLOC
- Chunks
- Volume
- Unique Reference LOC
- Source Statements (SS),

The source files can be written in Java, C++, C, or PHP programming language. Today, multiple programming languages are used in developing an SP. And it is needed to compare cost between artifacts created with different programming languages. Jones provided a conversion ratio chart of more than 20 programming languages [60]. The chart shows ratios to convert volume of one programming language to that of another. For example, if we use C++ as a main programming language, the volume of other languages can be converted to the equivalent volume of C++. Therefore, it is possible to equate product volume between programming languages.

Data gatherer collects feature attributes from RequisitePro, change/defect attributes from ClearQuest, and volume attributes from ClearCase, respectively, and

stores them into the KB. RequisitePro is a requirements management system. Rational RequisitePro is used to help software developers manage requirements to create software products. Requirement is a "condition or capability that must be met or possessed by a system or system component to satisfy a contract, specification, standard, or other formally imposed documentation" [38]. The requirements are the most important one in the project planning stage, because requirements are things to which the system being built must conform, and conformance to some set of requirements defines the success or failure of projects. However, requirements management has some problems such that requirements are frequently changed, requirements are related to one another and to other deliverables of the process in a variety of ways, and requirements must be managed by cross-functional groups of people like customers, analyst, and developers/testers. Rational RequisitePro is a tool to enable managers to organize, prioritize, trace relationships, and easily track changes to project requirements. The tool supports the database connection. Data gatherer gathers requirement attributes from the RequisitePro storage, and stores them into the PAMPA II knowledge base.

ClearQuest is a change and defect management system. ClearQuest works with ClearCase to track change/defect in an evolving project. ClearQuest is used to help software developers track defect/change in developing software products. ClearQuest is a customizable defect/change tracking system which supports developer, tester, project leader, and administrator. The tool can help manage every type of change activity associated with software development, including enhancement requests, defect reports, and documentation modifications. Data gatherer gathers defect/change attributes from the ClearQuest storage, and stores them into the PAMPA II knowledge base.

ClearCase is a Configuration Management System (CMS) to help software de-

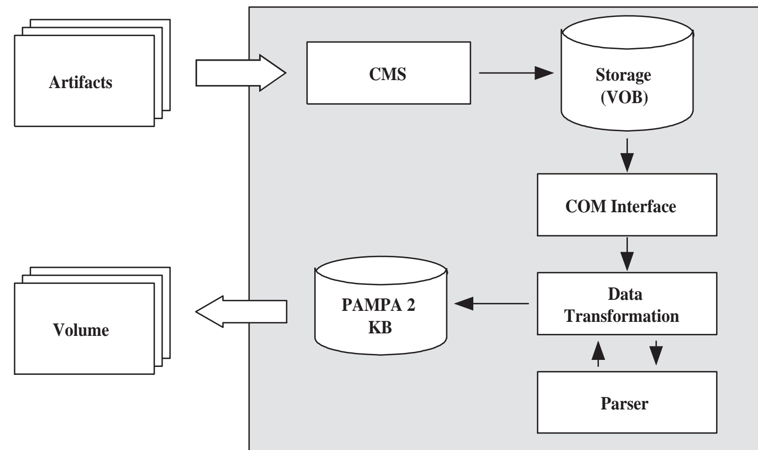


Fig. 18. Data transformation module

velopers track files and directories used to create an SP. Configuration Management System (CMS) is used to help software developers track files and directories used to create a software product. During the development stage, developers create artifacts according to requirements. Rational ClearCase is used for on-line storage of project artifacts and version control management.

We created a **Data Transformation** module which accesses to source files in ClearCase storage through the COM interface (ClearCase Automation Library). The **Data Transformation** module collects source files and passes them to the software metric parser to calculate volume. The volume attribute returns to the **Data Transformation** module which stores it into the PAMPA II KB. The detailed diagram is shown in Figure 18.

ClearCase supports parallel software development and software reuse across geographically distributed project teams. Developers at different locations can use the same VOB. Each site has its own copy, or replica, of that VOB. The set of replicas for a particular VOB is called a VOB family. At any time, a site can propagate changes

to other sites, using either an automatic or manual synchronization process.

CHAPTER V

PRODUCTIVITY PREDICTION MODEL

V.1 Model Building Based on Bayesian Analysis

Productivity is a major attribute for project management in estimating budget and time. Without an accurate objective productivity estimate, a software development project could result in budget overrun and project failure. And predicting development effort based on productivity estimate is central to project management. Project management consists of planning process such as scheduling activities and defining work breakdown structures, and controlling process such as measuring progress and reallocating resources.

In the planning process, we estimate initial productivity of a project using one of the effort estimation models, and predict development effort to make a plan. As the project evolves, we measure productivity from volume and effort of finished artifacts to determine the progress of the project. If the project's progress continues to match the plan, the project is in good shape. If there are some mismatches between the progress and plan, then corrective action must be taken. To make the project management harder, the initial productivity estimate given by any effort estimation model has high error rate so the plan based on it tends to be fraught with inaccuracies. Furthermore, the control based on the poor productivity estimate would lead the project to nowhere but failure. Therefore, it is necessary to transmit feedback information for reevaluation of the initial productivity estimate.

Bayesian analysis comes in handy to reevaluate the initial productivity estimate, because we have prior information about productivity (initial productivity), and sample data (measured productivity). Thus we can use Bayesian analysis to integrate

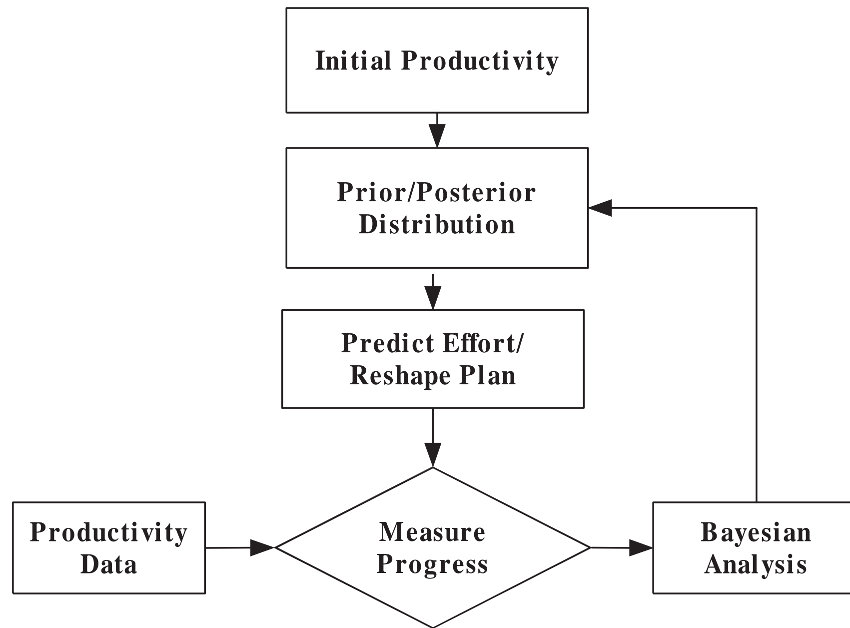


Fig. 19. Productivity prediction model

the information about productivity. And the updated productivity can be used to predict development effort for the rest of the project and to reshape the plan.

Figure 19 shows the process of productivity prediction model. With the initial productivity estimate, we can create a prior distribution of productivity. And we can predict development effort to make a plan. During the development process, we can gather productivity data which are used to measure the progress of the project. Also, the productivity data can be combined with the prior distribution via Bayesain analysis to get the posterior distribution. After getting the updated productivity, then we can predict development effort of the remaining activities of the project, and reshape the plan according to the new predicted effort. And the posterior distribution can act as the prior distribution for the next observed productivity data. Therefore, this procedure is continuously performed till the project ends.

The prior distribution is the most critical point of Bayesian analysis. The prior

distribution is unconditional to the sampling data, while the posterior distribution is conditional to the sampling data and prior information. And this is the key point of Bayesian analysis, since, once this prior distribution is known, inference can be led in an almost mechanic way afterwards. We used the following steps to build a prior/posterior distribution of productivity.

Step 1: In this research, we use the popular effort estimation model, COCOMO II, to determine the prior distribution. While not accurate enough to give the true estimate, the effort estimation model can provide a substantial estimate of productivity of a project. COCOMO II is an algorithmic model to estimate effort [21]. It requires volume of artifacts and cost drivers to estimate effort in PM. A cost driver is a model factor that affects the effort to complete a project. There are 17 cost drivers in COCOMO II. Effort multiplier (EM) is a value of rating level of a cost driver. And effort is estimated with the model:

$$PM = 2.94 \times Volume^{E} \times \prod_{i=1}^{n} EM_i, \tag{5.1}$$

where $Volume$ is the volume, EM is the effort multiplier, and E is the scale factor. We manipulated the equation (5.1) to calculate initial productivity directly estimate. The following model provides productivity estimate, Pr for a project:

$$Pr = \frac{Volume^{E}}{PM} = \frac{1}{2.94 \times \prod_{i=1}^{n} EM_i}. \tag{5.2}$$

There are two ways to determine the prior distribution with COCOMO II. First, the estimation of the COCOMO II depends on human judgment on the cost drivers. Therefore, the judgment can be different between humans. Multiple managers who have experience in using the effort estimation model can participate. Each of them

can provide different productivity estimate of the same project. And with these estimates, we can create the prior distribution of the project.

Second, we can use the team level productivity estimation. A project can consist of multiple teams. The 18 cost drivers are divided into four factors: product, platform, team and project. For the same project, the product, platform and project factors have same values while the team factor varies between teams. For example, if there are 9 teams in a project, a manager can obtain 9 productivity estimates. The prior distribution can be created with the estimates. The distribution of productivity is well known of its positive skewness [21][62]. To approximate the normal distribution, natural log transformation should be applied to productivity.

Step 2: In this research, we use a two-parameter univariate normal sampling model to make inferences about mean and variance of productivity. And we assume that the mean and variance are interdependent. After observing n sample data, the marginal posterior distribution of μ is:

$$p(\mu|y) \sim N(k, \sigma^2/(m_0+n)), \tag{5.3}$$

where $k = (m_0\mu_0 + n\overline{y})/(m_0+n)$ is the precision weighted average of the prior and sample data mean, σ^2 is the variance, μ_0 is the prior mean, $\overline{y}$ is the sample mean, m_0 is the prior sample size, and n is the sample size [30].

The variance, σ^2 is the inverse of the precision. The precision is an important parameter, because the higher the precision, the more highly concentrated are observations expected to be around the mean. The precision has a Gamma distribution. The marginal posterior distribution of τ is:

$$\tau = \sigma^{-2} \sim G(v/2, v\sigma_n^2/2), \tag{5.4}$$

where v is the posterior degrees of freedom, and σ_n^2 is the sample variance [30]. The

expected value of the precision τ is then $1/\sigma^2$

V.2 Productivity Console

The productivity console is developed to help managers keep track of project status in the life cycle of a software development project as well as to incorporate the productivity update model. The system is working with PAMPA II KB. The system is based on the architecture as shown in Figure 20.

Productivity measure collects facts from PAMPA II KB

Expert system fires rules and facts to track progress

Productivity update update productivity to reevaluate standards

Visual interface visualize progress status

PAMPA II gathers project attributes such as effort, salary, and volume from CASE tools and stores them into the KB. The productivity console uses the project attributes to visualize project status. The system is built on the Internet technology to help managers of software projects from those manned by a small team at one location to those with many teams dispersed all over the world.

The productivity console helps managers keep track of current project status based on expert system approach. In various AI approaches of interest, expert system is a very successful approximate solution to the classic AI problem of programming intelligence. An expert system makes extensive use of specialized knowledge to solve problems at the level of a human expert. An expert is a person who has expertise in a certain area. That is, the expert has knowledge or special skills that are not known or available to most people.

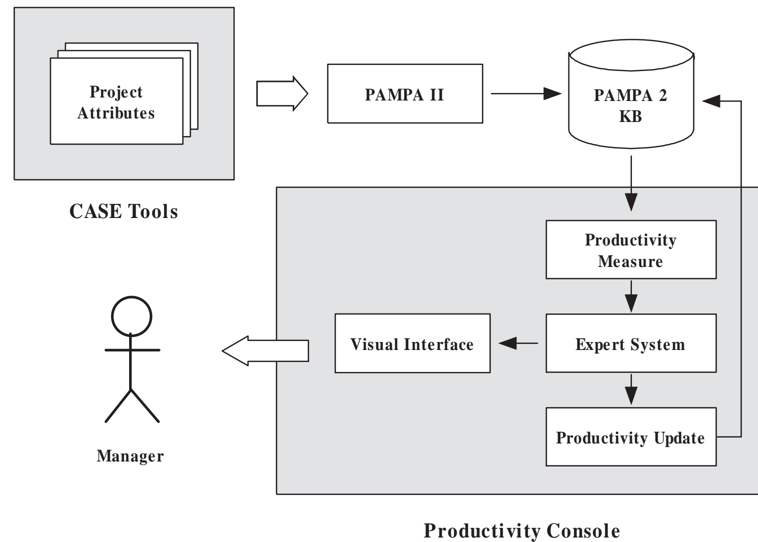


Fig. 20. Productivity console and PAMPA II

An expert system crystallizes and codifies the knowledge and skills of experts into a tool that can be used by non-specialists [61]. An expert system consists of a knowledge base and an inference engine. The knowledge base contains the domain-specific knowledge of a problem. The inference engine consists of procedures for processing the encoded knowledge of the KB together with any further specific information at hand.

Many Expert System Building tools used for academic research and industry have been evaluated from the literature [39][47][50][54][101]. In this research, JESS (Java Expert System Shell) is used to create an expert system. JESS is a Java version of the C Language Integrated Production System (CLIPS), an Expert System Building Tool. JESS allows users to build Java applets and applications that have the capacity to reason using knowledge supplied in the form of declarative rules and facts. Figure 21 shows the basic architecture used in the research.

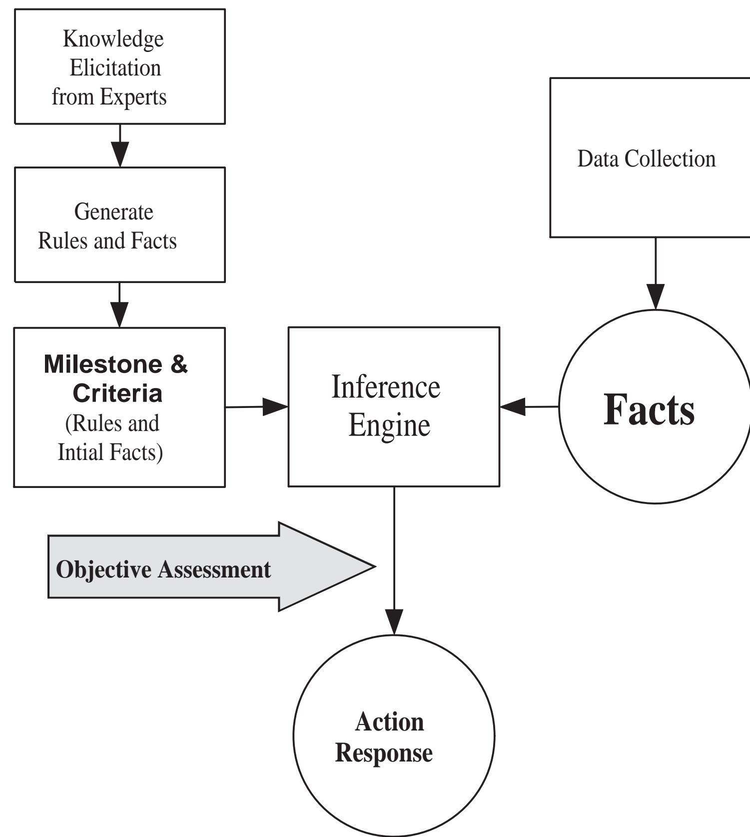


Fig. 21. Expert system diagram

V.3 Rules and Facts

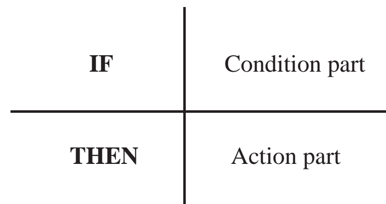
Many factors affect a project. Factors exist as rules and facts which can be acquired from experts' knowledge. Knowledge in the form of rules and facts is acquired from experts. PAMPA II stores the knowledge into the KB. PAMPA II gathers facts (project attributes) from an ongoing project. Inference engine analyzes project attributes (**Facts**) with the knowledge (**Rules and Initial Facts**), and reports objective assessment as shown in Figure 21. Therefore, when a software project encounters problems such as progress delay or resource deficit, expert system assists a manager in making appropriate decisions.

Table 7. Initial facts

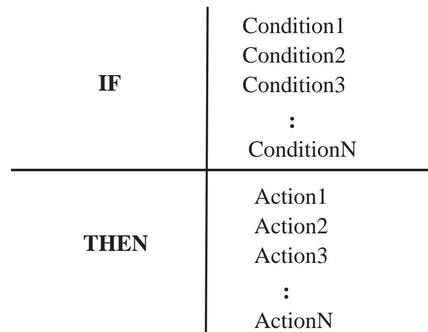
Fact	**Unit**	**Values**
Planned Effort	PM	LowerLimit, Expected, UpperLimit
Planned Productivity	Volume/Dollar	LowerLimit, Expected, UpperLimit
Planned Cost	Dollar	LowerLimit, Expected, UpperLimit
Planned Artifact Volume	Volume	Expected

Main facts primarily related to productivity are shown in Table 7. A software development expert defines the facts, and sets the values of the facts as initial facts for each activity. **Planned Effort** is the amount of effort allocated to an activity in plan, which has **LowerLimit**, **Expected** and **UpperLimit** value. The **Expected** value means that an activity would be best to finish in the time. And the **LowerLimit** and **UpperLimit** values give an interval within which the completion of an activity is expected to fall with a marginal effect on plan. If an activity takes longer than **UpperLimit** or finishes earlier than **LowerLimit**, then the expert system will report the problem to managers. The values of **Planned Productivity**, Planned Cost, and **Planned Artifact Volume** can be set as well. The project attributes (**Facts**) gathered from an ongoing project are:

- Measured effort to create artifacts
- Measured productivity



Rule 1: A simple form of the IF-THEN rule



Rule 2: An enhanced form of IF-THEN rule

Fig. 22. Forms of the IF-THEN rule

- Measured cost to create artifacts
- Measured volume of artifacts

The expert system takes the knowledge (**Rules and Initial Facts**) stored in the KB and tests them against the project attributes (**Facts**). By firing the rules and facts from the KB, the expert system can dynamically advise the manager of the project status.

Generally, a rule consists of a condition-part and an action-part as shown in rule 1 of Figure 22. And some of rules contain more than one condition in the condition-part or more than one action in the action-part as shown in rule 2 of Figure 22. Following are some examples of the rules:

Rule example 1: This example shows that the expert system tests if measured productivity of an activity is lower than the **LowerLimit** value of the **Planned**

Productivity of the activity.

```
(defrule isLowerThanLowerLimitOfPlannedPr
?activity<-(activity (id ?x)
           (name ?a) (date ?xx))
 =>
 (new PrintMan "The activity is"
   (create
    ?a
  )
)
)


(deffunction isLower (?a ?b)
   (if (> ?a ?b) then
    (return TRUE)
   else
    (return FALSE)
  )
 )
```

Rule example 2: This example shows that the expert system tests if measured cost of an activity is higher than the **UpperLimit** value of the **Planned Cost** of the activity.

```
(defrule isHigherThanUpperLimitPlannedCost
?activity<-(activity (id ?x)
           (name ?a) (date ?xx))
```

```
 =>
 (new PrintMan "The activity is"
   (create
    ?a
  )
)
)

(deffunction isHigher (?a ?b)
   (if (> ?a ?b) then
    (return TRUE)
   else
    (return FALSE)
  )
 )
```

Once defined and set, **Rules and Initial Facts** are stored in the **Criteria** of the KB. The expert system monitors new **Facts** and tests the **Rules and Initial Facts** on them without intervention of a manager.

V.4 Project Attributes Gathering from CASE Tools

Many software development projects do not gather metrics because of the expenses involved with the metric gathering process. PAMPA II reduces the cost to a minimum. PAMPA II can automatically on a periodic or continuous basis gather project attributes in an Internet environment. Once project attributes have been gathered and stored into the KB, measurement and assessment of activities can be easily per-

formed.

A software project has a plan. The software project plan tells the manager the desired software project status. Accurately measured status can be compared with planned status. Inaccurate status information could lead to faulty decision making and cause project delays. On the other hand, accurate status comes from measuring software project attributes. The measured project attributes can then be compared with attributes of planned work activities.

The basic planning information is a work activity, which contains the following attribute:

- Id number/Owner
- Owner
- Name/Activity type
- Description/Feature/Requirement
- Predecessor and successor work activities
- Artifact volume to be produced
- Type of language
- Initial milestone for beginning an activity
- Final milestone for terminating an activity.

During the execution of the work activity, resources used, actual time, work products produced, and so forth are measured and compared to the work activities defined in the plan. The planned effort, predecessor, and successors of each work activity can be used to prepare an activity network analysis for the project and

identify the critical path(s), which determines the overall schedule. PAMPA II gathers the planning information from MS Project, and stores it into the KB. The information will be used as the knowledge (**Initial Facts**) for the expert system.

With a solid detailed project plan and accurate status, the manager can take corrective action when problems or risks occur. Software project control is concerned with initiating corrective actions, tracking them to closure, and analyzing corrective action trends. Corrective action strategies include describing the requirements or changing the design, and/or tailoring the project plan to extend the schedule; adding, modifying, or replacing resources; extending the work hours (overtime); and/or cutting corners on planned work activities such as reviews, testing, source codes, documentation, and artifacts. Status indicators include (at least) the quantity and quality of work products developed or modified, schedule milestones achieved, resources expended, risk indicators, and rework to correct defects. Project control is concerned with comparing the current status of the software project to planned status and applying corrective action if current status, as measured by the status indicators, does not conform to plans.

During the development stage, developers create artifacts based on the requirements. ClearCase provides on-line storage of artifacts and version control management. We used **Data Transformation** module to access to the storage via the COM interface (ClearCase Automation Library) as shown in Figure 18. PAMPA II has an interface to relate the information to a specific activity in the plan.

An activity is the smallest work package that has milestones and assigned individuals such as manager, designer, developer, and tester. As shown in Figure 23, PAMPA II stores activity information (id number, activity name, activity type, etc) in **Activity**, and milestones in **InitialMilestone** and **FinalMilestone**. **Individual** is many-to-many mapping to **Activity**. Each **Individual** has salary rate in

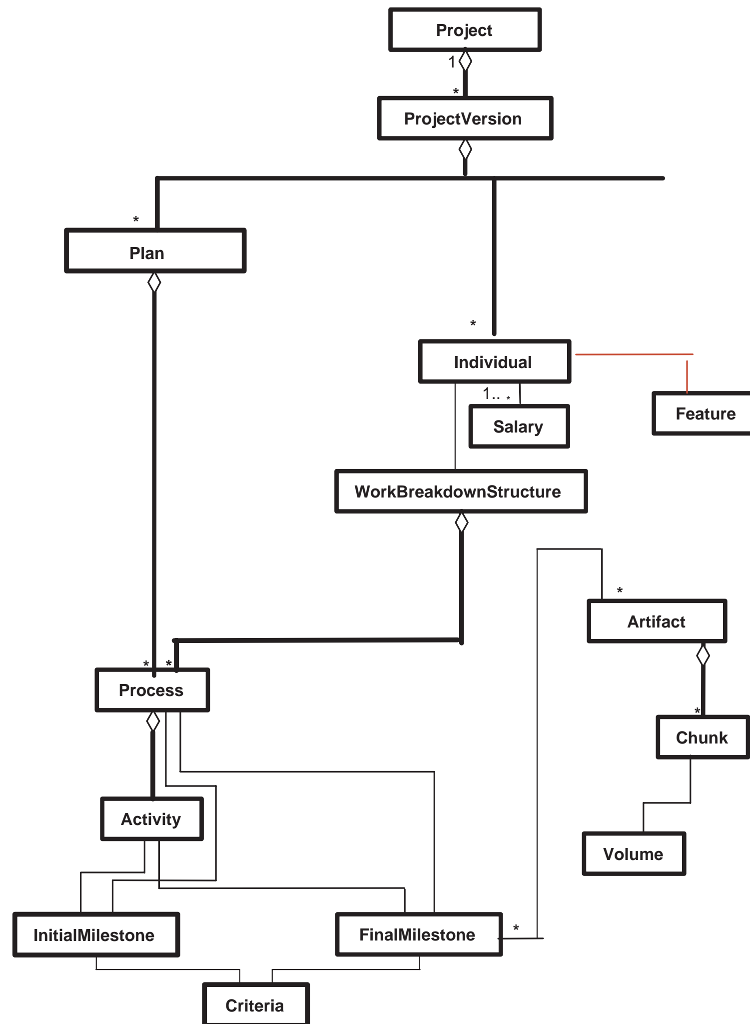


Fig. 23. Detailed KB schema on plan

Salary. As a software project evolves, programmers create artifacts according to the requirements. **Artifact** stores the artifact information: artifact type, file name, directory, and programming language to develop the artifact. **Artifact** is one-to-one mapping to **Activity**. An artifact can have several chunks. Volume metric is stored in **Volume**.

The volume and cost gathered in the KB are used to measure productivity. For example, if an activity is completed, we can get effort from the difference of

Table 8. Earned value

Value Name	Definition
BCWS	Budgeted cost of work scheduled
BCWP	Budgeted cost of work performed
ACWP	Actual cost of work performed
SV	Schedule Variance: BCWP - BCWS
CV	Cost Variance: BCWP - ACWP
BV	Budget Variance: BCWS - ACWP
CPI	Cost performance index: BCWP / ACWP
SPI	Schedule performance index: BCWP / BCWS
CR	Critical ratio: CPI * SPI

InitialMilestone and **FinalMilestone**, salary rate from **Salary**, and volume from **Volume**.

In addition to productivity, we gathers earned value to follow the progress of project. The earned value compares work completed to work planned for completion and cost of work completed to cost of work planned for completion in each reporting interval [42]. If cumulative work completed is less than planned work, the project is behind schedule; if cumulative cost of work completed is greater than planned (budgeted) cost the project is over budget. All combinations are possible: ahead of schedule, under budget, behind schedule, over budget, and so forth.

Fundamental values gathered directly from the KB are BCWS, BCWP, and ACWP as shown in Table 8. The other six variables are calculated from the three fundamental values. With evaluation of the earned values, managers easily detect current resource or schedule problem. For instance, a cost performance index (CPI) of 0.8 means the project spent 25% more resource than scheduled. The earned values are the barometer to the resource and schedule status.

The KB schema showing the relationship between project attributes and CASE tools is in Tables 9 and 10 at the end of this chapter.

V.5 Visual Interface

Productivity console provides four dial charts for managers to quickly discern the true status of a project. Productivity console gives a manager a quick view of project status, which will be used to monitor project management. It consists of **Schedule**, **Progress**, **Productivity**, and **Resource meter**. **Schedule** meter shows time spent in a schedule, **Progress** meter progress of a project, **Productivity** meter productivity, and **Resource** meter resource balance. In short, the productivity console helps managers do the following tasks:

- Oversee person-month spent
- Check the status of activities (finished, and goal)
- View the current and estimated productivity
- Check the resource balance

Figure 24 shows the project level status view on the review date, 2-27-2004. **Schedule** meter has an arrow which points to the expended time in plan. In the figure, we can tell the project spent 1590 out of 6890 PM. **Progress** has an arrow and a line. The arrow points to the number of finished activities, while the line points to the number of activities currently working on the review date. Currently, 37 activities were completed out of 55 current activities. The meter shows the total number of activities, 159. **Productivity** meter has an arrow and a line as well. The arrow indicates the measured productivity while the line shows the updated productivity.

Project's current productivity of the project is 0.26 whereas the updated productivity is 0.301 SLOC/dollar. **Resource** meter has two areas. The left area shows that resource is deficient, and the right one shows otherwise. The arrow points to the center line when resource is consumed as planned. We tell the project is suffering from budget deficits of $7,328. The project view provides information about team such as team name, description, and each team's current productivity.

The productivity console can provide the status view of team level as well. Figure 25 shows the detailed view of a team progress: 180 PM were spent out of 780 PM; 5 activities were finished out of 18; current productivity is 0.25; resource balance is $360 below the plan. The team's updated productivity is 0.289. And the view shows each member's name, title and current productivity.

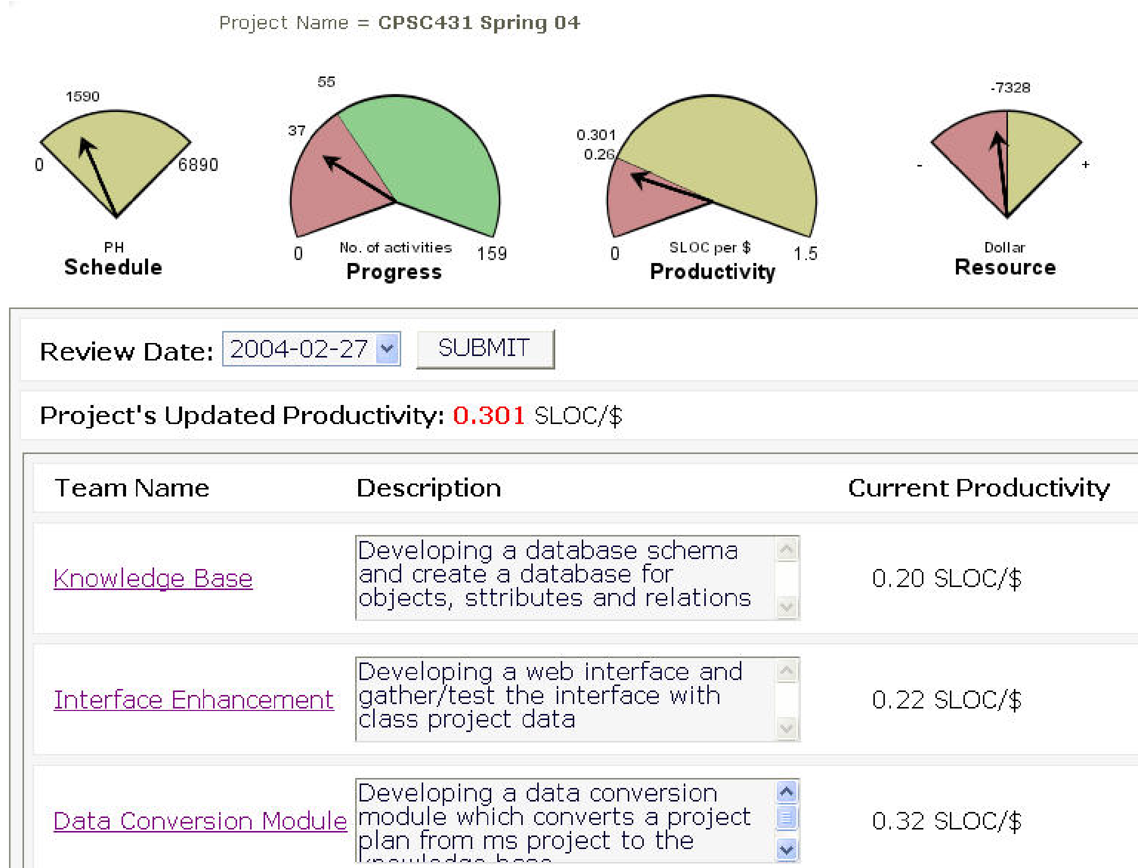


Fig. 24. Productivity console shows a project level view

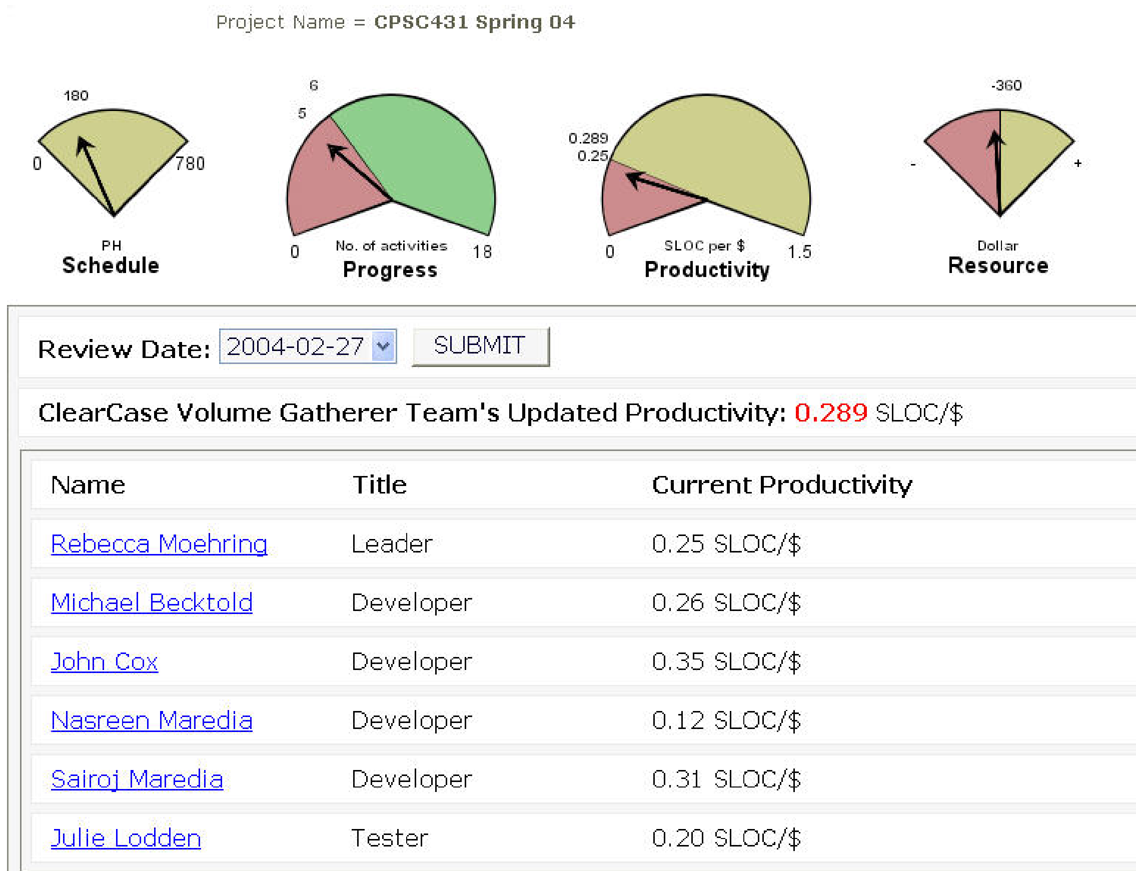


Fig. 25. Productivity console shows a team level view

Table 9. PAMPA II schema 1

PAMPA II			Source
Objects	**Attributes**	**Relationships**	
ProjectList	Name, Description	contains Projects	MS Project
Project	Name, Description, Cost, EffortToDate	contained in a ProjectList	MS Project
ProjectVersion	Name, Description, Cost, Time	contained in a Project	MS Project
Plan	Name, Description	part of a ProjectVersion	MS Project
Process	Name, Description, InitialMilestone, FinalMilestone	contained in a Plan and WorkBreakdownStructure	MS Project
Activity	Name, Description, InitialMilestone, FinalMilestone	contained in a Process and related to Activit(y)ies	MS Project
InitialMilestone	PlannedStartDate, AcualStartDate	an attribute of Process, Activity	MS Project
FinalMilestone	PlannedEndDate, AcualEndDate	an attribute of Process, and Activity	MS Project
Criteria	Rule	an attribute of InitialMilestone, FinalMilestone	Project Object
Supplier	Name, Description	are contained in a ProjectVersion	Project Object
ReusableSourceFile	Name, Description	provided by Suppliers	ClearCase
COTSRunFile	Name, Description	provided by Suppliers	ClearCase
Organization	Title, Description	contain Individual, perform WorkBreakdownStructure	MS Project
Individual	Title, Productivity	authors Artifacts, perform WorkBreakdownStructure	MS Project

Table 10. PAMPA II schema 2

PAMPA II			Source
Objects	**Attributes**	**Relationships**	
Salary	Amount, EffectiveDate	are related to an Individual	MS Project
WorkBreakdown Structure	Name, Description	associated with an Organization, Individual	MS Project
SoftwareProduct	Name, Description, Size	contained in a ProjectVersion	Project Object
Feature	Name, Description	contained in a SoftwareProduct	Project Object
Version	PreviousVerId, SourceDir, Id, DateCreated	contained in a SP owned by an Individual related to Features	ClearCase
Subsystem	Name, Type	contained in a Version	ClearCase
Artifacts	Name, Language	authored by an Individual, Organization	ClearCase
Chunk	Name, Size	contained in Artifact	ClearCase
Volume	ObjectPoint, FunctionPoint, SLOC	attribute of a Chunk	ClearCase

CHAPTER VI

EXPERIMENTAL RESULTS

VI.1 Project Description

In order to illustrate the research result of the productivity prediction model, software project data have been collected from nine undergraduate class projects of the Computer Science Software Engineering course at Texas A&M University. The nine software projects of the Software Engineering course focus on developing Internet application software packages. Those course projects developed Web Based Software Metrics Collection/Visualiztion systems and used the Extreme Programming development process for their projects. Extreme Programming or XP is a development process that can be used by small to medium sized teams to develop high quality software within a predictable schedule and budget and with a minimum of overhead [13]. XP is currently one of the most widely used agile processes in the industry.

The course projects, used for the demonstration of the research result, lasted from January 20, 2004 through May 14, 2004. All projects used PHP, Java, Visual Basic and Visual C++ as the programming language to implement the three-tier architecture running on the Internet. In this research, we chose the Java as a base langauge to calculate volume. We developed an equation to convert the volume of a language to the equivalent SLOC based on the SLOC conversion ratios table 4.

$$EquiSLOC = 53 \times (\frac{y_1}{34} + \frac{y_2}{29} + \frac{y_3}{15}). \tag{6.1}$$

where $EquiSLOC$ is equivalent SLOC, y_1 the volume of Visual C++, y_2 the volume of Visual Basic, and y_3 the volume of PHP.

Each software package involved the development of database, middleware, core

modules for parsing source codes, and a Graphical User Interface to represent the metrics information. ClearCase from the IBM Company was used as the CMS running on the Microsoft Windows Server environment.

Project teams and descriptions are as follows:

Project Team 1 (Knowledge Base) It had one team leader and four team members and developed a database schema and prepare database schema for all objects, attributes and relations.

Project Team 2 (Interface Enhancement) It had one team leader and five team members and developed the web interface and gather/test the interface with class project data.

Project Team 3 (Data Conversion Module) It had one team leader and six team members and developed the data transformation module which converts a project plan from MS Project to the KB. The module includes database connection, and transforms plain data format to database format.

Project Team 4 (ClearCase Volume Gatherer) It had one team leader and five team members and developed programs to gather volume from ClearCase. The program includes a parser to convert programming language and gather SLOC volume of chunks from ClearCase.

Project Team 5 (Visual Studio Volume Gatherer) It had one team leader and four team members and developed programs to gather volume from Visual Studio. The program includes a parser to convert programming language and gather SLOC volume of chunks from Visual Studio.

Project Team 6 (Source Safe Volume Gatherer) It had one team leader and six team members and developed programs to gather volume from Source Safe.

The program includes a parser to convert programming language and gather SLOC volume of chunks from Source Safe.

Project Team 7 (Gantt Chart) It had one team leader and five team members and developed Gantt chart program.

Project Team 8 (Activity Network Chart) it had one team leader and five team members and developed Activity Network chart program.

Project Team 9 (Control Center) It had one team leader and five team members and developed control center.

The above projects went through the following software development phases:

1. Requirement analysis
2. System design
3. Unit design
4. Coding
5. Unit testing
6. Integration testing
7. Product release/demo and final report.

For better control and tracking the progress of the project, each student had to turn in a Weekly Status Report at the end of each week. The format of the Weekly Status Report is shown in Figure 26. The report is for the manager's reference to the current status of each person's activity on the project. In this case, by comparing the progress information reported from each developer with the productivity console, the

Weekly Individual Status Report Template

Date:

Name:

TEAM STORIES AND TASKS:

INDIVIDUAL STORIES AND TASKS:

ACCOMPLISHMENTS:

CURRENT TASKS:

PROBLEMS:

SUGGESTED PROBLEMS SOLUTIONS:

TASKS TO ACCOMPLISH NEXT WEEK:

Fig. 26. Format of the weekly status report

manager can see if any incorrect report of the project information from developers may exist.

VI.2 Preliminary Productivity Estimation

The project initiated on 1-20-04. The project teams started analysis of the project requirements, and designed the system to develop till 2-6-04. After finishing the analysis and design, they started the actual programming work on 2-9-04. Deliverables

in the analysis and design period were documents as follows:

- Functional requirement document
- Non functional requirement document
- User interface story
- Metric document
- Knowledge Base design document
- Module design document
- Test plan

Based on the analysis and design, we estimated productivity of the project to determine the prior distribution of productivity with the COCOMO II effort estimation model. While there are 4 factors in the COCOMO II model, the product, platform, and project factors are same for one project. For example, the target products were the same Internet applications, and they used the same development platform and worked at the same environment. Therefore, we fixed the three factors at same values for all project teams and judged the values of the personnel factors of each team. The personnel factors are for rating the development team's capability and experience. The detailed descriptions are as follows:

Analyst Capability (ACAP) Analysts are personnel who work on requirements, high-level design, and detailed design. The major attributes are analysis, and design ability, efficiency and thoroughness, and the ability to communicate and cooperate. See Table 11.

Table 11. ACAP cost driver

Descriptors	15th percentile	35th	55th	75th	90th	
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	1.42	1.19	1.00	0.85	0.71	n/a

Table 12. PCAP cost driver

Descriptors	15th percentile	35th	55th	75th	90th	
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	1.34	1.15	1.00	0.88	0.76	n/a

Table 13. PCON cost driver

Descriptors	48%/year	24%	12%	6%	3%	
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	1.34	1.15	1.00	0.88	0.76	n/a

Programmer Capability (PCAP) Major attributes are ability, efficiency and thoroughness, and the ability to communicate and cooperate. See Table 12.

Personnel Continuity (PCON) Major attributes are annual personnel turnover. See Table 13.

Application Experience (APEX) Major attributes are level of applications experience of the project team developing the software system or subsystem. See Table 14.

Platform Experience (PLEX) Major attributes are level of experience of the use of more powerful platform, including more graphic user interface, database,

Table 14. APEX cost driver

Descriptors	$\leq$ 2 months	6	1 year	3 years	6 years	
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	1.22	1.10	1.00	0.88	0.81	n/a

Table 15. PLEX cost driver

Descriptors	$\leq$ 2 months	6	1 year	3 years	6 years	
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	1.22	1.10	1.00	0.88	0.81	n/a

Table 16. LTEX cost driver

Descriptors	$\leq$ 2 months	6	1 year	3 years	6 years	
Rating Levels	Very Low	Low	Nominal	High	Very High	Extra High
Effort Multipliers	1.20	1.09	1.00	0.91	0.84	n/a

networking, and distributed middleware capabilities. See Table 15.

Language Experience (LTEX) Major attributes are level of programming language and software tool experience of the project team developing the software system or subsystem. See Table 16.

Figure 27 shows productivity estimates and values of personnel factors of 9 project teams. The average productivity is 914 SLOC per PM. COCOMO II treats the number of person-hours per person-month as an adjustable factor with a nominal value of 152 hours per PM. And we used an estimated equivalent cost of $15.00 per hour. After applying the adjustment, the average productivity was then 0.4 SLOC per dollar. We used this productivity estimate as an initial productivity for the project.

	Prior Distribution	Cost Drivers					
		Personnel Factors					
Team	Pr. In SLOC	ACAP	PCAP	PCON	APEX	PLEX	LTEX
1	525	1.19	1.19	1	1	1.19	0.91
2	769	1	1.19	1	0.88	1.19	0.84
3	1005	1.19	0.88	1	1	0.91	0.84
4	1076	0.85	0.88	1	1	1.19	0.84
5	993	1.19	1	1	0.88	0.85	0.91
6	873	1.19	0.88	1	0.88	1.19	0.84
7	1028	0.85	0.85	1	1	1.19	0.91
8	1054	1.19	0.88	1	0.88	0.91	0.91
9	904	1.19	1.19	1	0.88	0.85	0.84

Fig. 27. Productivity estimates for the project

The distribution of productivity is well known of its positive skewness [21][62]. To approximate the normal distribution, natural log transformation was applied to the productivity. We created a prior distribution with mean μ, -0.914, and variance σ^2, 0.052 in log form.

Figure 28 shows a sample of the planned activities and effort/cost of a team. This project continued for about three months beginning from 1-20-04 to 5-14-04 and the effort of two work hours on the project per day for each student. The actual development started on 2-9-04 and finished on 4-23-04. There were 9 teams in the project, which had 159 activities with planned 6890 person-hour and $103,350 for the budget. The development process consisted of unit build and weekly build. Each developer built his/her programming modules in his/her own work place in the CMS till Thursday and submitted the modules to a shared work space to integrate with other team member's modules on Friday. We collected project attributes on every Friday.

ID	Phase/Activity Name	Owner	Intial Milestone	Final Milestone	Estimated Volume	Estimated Effort (PH)	Estimated Cost ($)
1	**Project Initiation**		2004-01-20	2004-01-23			
1.1	Project Goals and Objectives	AH	2004-01-20	2004-01-23	Document	8	136
1.2	Requirement documents	BR	2004-01-20	2004-01-23	Document	8	136
1.3	Non functional requirements	SH	2004-01-20	2004-01-23	Document	8	136
2	**System Design**		2004-01-26	2004-02-06			
2.1	User Interface story	AH	2004-01-26	2004-01-30	Document	10	170
2.2	Metric/KB design	PL	2004-01-26	2004-01-30	Document	10	170
2.3	Module design	BC	2004-02-02	2004-02-06	Document	10	170
2.4	Test plan	JK	2004-02-02	2004-02-06	Document	10	170
3	**System Development**		2004-02-09	2004-04-23			
3.1	User interface prototype	AO	2004-02-09	2004-02-27	180	30	510
3.2	User interface module	AO	2004-03-01	2004-04-02	300	50	850
3.3	User interface projection	AO	2004-04-05	2004-04-23	168	28	476
4.1	KB interaction module	PL	2004-02-09	2004-03-05	240	40	680
4.2	Attribute collection module	PL	2004-03-08	2004-03-26	180	30	510
4.3	Attribute projection module	PL	2004-03-29	2004-04-23	228	38	646
5.1	API module for VS	BR	2004-02-09	2004-03-19	360	60	1020
5.2	Input module for VS	BR	2004-03-22	2004-04-09	180	30	510
5.3	Output module for VS	BR	2004-04-12	2004-04-23	120	20	340
6.1	Parser input for C	SH	2004-02-09	2004-03-19	360	60	1020
6.2	Parser output for C	SH	2004-03-22	2004-04-09	180	30	510
6.3	Parser generating for C	SH	2004-04-12	2004-04-23	120	20	340
7.1	Parser input for C++	BC	2004-02-09	2004-03-19	360	60	1020
7.2	Parser output for C++	BC	2004-03-22	2004-04-09	180	30	510
7.3	Parser generating for C++	BC	2004-04-12	2004-04-23	120	20	340
8.1	Parser input for Java	AH	2004-02-09	2004-03-19	360	60	1020
8.2	Parser output for Java	AH	2004-03-22	2004-04-09	180	30	510
8.3	Parser generating for Java	AH	2004-04-12	2004-04-23	120	20	340
9.1	Parser input for PHP	JK	2004-02-09	2004-03-19	360	60	1020
9.2	Parser output for PHP	JK	2004-03-22	2004-04-09	180	30	510
9.3	Parser generating for PHP	JK	2004-04-12	2004-04-23	120	20	340
10	**System Test**		2004-02-09	2004-05-07			
10.1	Unit test	JK	2004-02-09	2004-04-23	Document	110	1870
10.2	Integration test	JK	2004-04-26	2004-05-07	Document	30	510
11	**Project Finalization**		2004-05-03	2004-05-14			
11.1	Project report	AH	2004-05-03	2004-05-14	Document	20	340
	Total				4596	**990**	**16830**

Fig. 28. A sample of planned activities and effort/cost

VI.3 Data Collection and Experimental Results

To evaluate the research, the project attributes were collected from the software project environment. The project attributes information from project teams will be used in different cases to effectively validate the productivity prediction model. The comparison for both Bayesian analysis and conventional statistical analysis will be explained in detail.

In the software project environment, each developer had a working directory

that stored all source files. The working directory had a link to the directory under the control of the CMS. In order to test the accuracy of the productivity prediction model, some of the source files were collected and then analyzed by hands. This type of project attributes collection has been assured to the same project attributes collected from PAMPA II. The project attributes used by the research are:

- Activities
- Planned artifact volume
- Planned Productivity
- Measured Productivity
- Planned effort
- Measured effort
- Planned cost
- Measured cost

We measured productivity on weekly basis. The actual development took 13 weeks starting from 2-9-04. The development activities finished on 4-23-04 when we obtained the true average productivity of each team. Every Friday during the development, we checked out the volume change of artifacts and effort to build them, and stored the measured productivity into the KB. After collecting productivity data, we then applied Bayesian analysis to get the posterior distribution of productivity. With the posterior distribution, we obtained the productivity mean, standard deviation, and 95% confidence intervals of each team. Confidence refers to the probability that the ultimate conclusion will be a correct statement.

Table 17. Posterior distribution on 2-27-04

Team No.	**Posterior Mean**	**Posterior SD**	**2.5% Percentile**	**97.5% Percentile**	**True Productivity**
1	0.250	0.017	0.217	0.283	0.217
2	0.253	0.013	0.228	0.278	0.239
3	0.337	0.032	0.274	0.400	0.317
4	0.289	0.021	0.248	0.330	0.265
5	0.305	0.020	0.266	0.344	0.266
6	0.221	0.012	0.197	0.245	0.243
7	0.332	0.032	0.269	0.395	0.316
8	0.299	0.015	0.270	0.328	0.317
9	0.308	0.021	0.267	0.349	0.305

We analyzed the true average productivity with the confidence intervals which we obtained each Friday. And we found that all teams' true average productivity fell within their confidence intervals on 2-27-04 when the developers spent three weeks which were less than 25% of the total development period. The results of the evaluation are shown in Table 17. The table shows the experimental results of 9 teams which participated in the experiment: posterior mean; posterior standard deviation; 95% confidence interval; and true average productivity. As shown in the table, for example, team 1's confidence interval covers its true average productivity. In Bayesian terms, we can claim that team 1's true average productivity is inside the confidence interval with 95% probability. We can claim the other teams' true average productivity as well.

Considering the time to obtain the confidence intervals, we can claim that the

reevaluation of the initial productivity with productivity data shows promise in reducing the uncertainty caused by an effort estimation model in the early life cycle of a software development project.

CHAPTER VII

CONCLUSIONS AND FUTURE EXTENSION

VII.1 Conclusions

In this research we explored the possibility to build a productivity prediction model based on Bayesian analysis. But the model is not developed as a replacement of any effort estimation model. There is a big difference in them: an effort estimation model provides productivity estimate before a project starts; the productivity prediction model provides productivity estimate during an ongoing project. The productivity prediction model plays an important role to reevaluate the initial productivity estimate and provide better guidelines to control over the development process.

Bayesian analysis shows its strength through the research experiment. The choice of a prior distribution stirs many debates among statisticians because it is actually rare to have a completely specified prior distribution. However, the prior distribution can be considered either a tool that provides a single inferential procedure or a way that summarizes the available prior information and the uncertainty surrounding this information [91]. As shown in the research, the carefully chosen prior distribution leads to considerably good inferences about the parameter of interest.

From a practical point of view, the development of a prior distribution relies on the ability of individuals to represent their knowledge (or even the limitations of this knowledge thereof) in terms of probabilities. The effort estimation model used in this research suffers from a large margin of error, however, both researchers and industry practitioners have devoted considerable effort to improve the accuracy of the model. Therefore, a manager can construct a good prior distribution about productivity when s/he trains to get better knowledge and experience on the model.

The main hypothesis of this research proved to be true according to the research results - productivity prediction based on Bayesian analysis reduces the uncertainty by providing a better productivity estimate in a software development project. And we strongly believe that the productivity prediction model clearly proves to be a good tool to predict productivity of developers in an ongoing software development project since it updates the inaccurate productivity estimate with real data automatically. Therefore, managers can command better control over the development process with the aid of the model.

This research describes also a productivity console that is created to assess project attributes and to provide graphical charts to visualize the status of a project. Actual status represented on the console can help project managers continually monitor projects and control developers and resources. Tools based on this technology can help managers make timely assessment of project status and will allow plan modification early in a project life cycle before major problems develop. More over, the console works on the Internet, which allows managers to monitor projects taken place remotely.

Several other exploratory studies of this research have been conducted in the following areas: the use of the dynamic collection of project data as facts for updating an initial standard during the software development process; knowledge elicitation from the manager to define rules; the use of project attributes for objective measurement/assessment instead of subjective observation from the developers and the managers.

In this research, several objectives have been reached. First, a productivity prediction model has been created, which can be used in a software development project. Second, we have gathered real data from a course project to validate the productivity prediction model. Third, a productivity console has been created to

help managers monitor and control a software development process. Finally, The primary benefits of this research are:

- Productivity prediction: Bayesian approach provides a convenient way of updating productivity. And it gives a powerful inference in terms of probability. It help managers better control software development processes.
- Attributes gathering: The automatic attribute gathering feature of PAMPA II increases effectiveness of project attributes measurement and assessment.
- Adopting labor cost: It gives a manager a view of controlling resource expense in the international development environment.
- Project attributes assessment: Objective assessment of project attributes helps a manager easily find out problems and take corrective actions.
- Web-based console: It helps a manager monitor project status via Internet.

In conclusion, we believe that the productivity prediction model can provide a unique opportunity for software development project managers to control resources in a software development project, and that the productivity console can give a better view of monitoring a software development process on Internet environment.

VII.2 Future Extension

While doing the experiment, we encountered some problems. Those problems are missing values, personnel turnover, quality problem of software products, and inaccurate productivity estimation by COCOMO II. The first two problems inhabit in academic projects. Some of the difficulties found in the project are: students' negligence of observance of the development process, no previous experience of using CASE

tools for the development, missing class attendance, and dropping the course during the semester. We found the missing values cause the reliability and credibility of the model. Therefore, the generic applicability of the research should be tested further using more real world project data from a variety of corporations or the military.

The third problem suggests the problem of a quantity based approach. Success of a software development project depends not only on the expense of time and budget, but also on the quality of products. Thus, quantity based approach alone doesn't guarantee the success of a software development project. Therefore, we suggest the use of standard functional and usability test at each evaluation.

The forth problem confirms again the inaccuracy of the current effort estimation models. Especially, the academic environment in which we used COCOMO II to estimate productivity is different from that in which COCOMO II has been built. And the suggested calibration method needs a lot of historical data and expertise which academic institutes usually don't have. Therefore, it is recommended for a manager to use productivity estimate by any effort estimation model with a grain of salt.

The productivity console we have created provides basic features to monitor the software development process. To enhance the console to become a versatile software project management tool, it is recommended to add more features as follows:

- Using more cost estimation tools for more accurate cost comparisons
- Risk warning and recommendations for avoiding project failure
- Suggestions which help the manager balance cost, quality, and time in making decisions about the project progress in compliance to the planed activities
- Measurement which enables the manager to visualize how well the software

project is reaching greater project goals and re-plan the way to reach these goals if necessary

Overall, the future extension of this research is encouraged to improve the integrated software project management, control and tracking system in the following aspects:

1. PAMPA II provides many useful software metrics. However, for the feasibility of the demonstration purpose, only some of the important project attributes have been used in this research. Future experiments may include all the metrics from PAMPA II to strengthen the capability of the productivity console.

2. The productivity console should have ability to keep track of the effects caused by the requirement changes, since requirement changes bring about reevaluation of the project plan in cost, effort and resource to reflect the change.

3. The productivity console should have a more advanced expert system to suggest an alternative control for the software development process. Therefore, the expert system may assist the manager in making the appropriate decisions when the software project encounters problems or risks.

REFERENCES

[1] *Function Point Counting Practices: Manual Release 4.0.* Blendonview Office Park, OH: International Function Point Users' Group, 1994.

[2] T. Abdel-Hamid and S. Madnick, "Lessons Learned from Modeling the Dynamics of Software Development," *Communications of the ACM*, vol. 32, no. 12, pp. 1426–1438, 1989.

[3] T. Abdel-Hamid, "Adapting, Correcting, and Perfecting Software Estimates: A Maintenance Metaphor," *Computer*, vol. 26, no. 3, pp. 20–29, 1993.

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

[5] C. Abts, "COTS-based Systems (CBS) Functional Density—a Heuristic for Better CBS Design," *International Conference on COTS-Based Software Systems*, pp. 1–9, 2002.

[6] A. Albrecht, "Measuring Application Development Productivity," *Proceedings in Joint SHARE/GUIDE/IBM Application Development Symposium*, pp. 83–92, 1979.

[7] J. Aron, *Estimating Resources for Large Programming Systems.* Litton Education Publishing, 1976.

[8] K. Atkison and M. Sheppered, "The Use of Function Points to Find Cost Analogies," *Proceedings in European Software Cost Modelling Conference*, 1994.

[9] A. Awani, *Data Processing Project Management.* Princeton, NJ: Petrocelli, 1986.

[10] L. Badley and M. Lehman, *The Characteristics of Large Systems.* Cambridge: MIT Press, 1979.

[11] J. Bailey and V. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*, pp. 107–116, 1981.

[12] V. Basili and B. Boehm, "COTS-Based Systems Top 10 List," *Software Management*, vol. 34, no. 5, pp. 91–93, 2001.

[13] K. Beck, *Extreme Programming Explained: Embrace Change.* Boston, MA: Addison-Wesley, 1999.

[14] C. Behrens, "Measuring the Productivity of Computer Systems Development Activities with Function Points," *IEEE Transactions on Software Engineering*, vol. 9, no. 6, pp. 648–652, 1983.

[15] E. Bennatan, *On Time Within Budget: Software Project Management Practices and Techniques.* New York, NY: John Wiley & Sons, 2000.

[16] J. Bernardo and A. Smith, *Bayesian Theory.* New York, NY: John Wiley & Sons, 2001.

[17] L. Bernstein, "Software in the Large," *AT&T Technical Journal*, vol. 1, pp. 5–14, 1996.

[18] B. Boehm, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[19] B. Boehm, "Theory-W Software Project Management: Principles and Examples," *IEEE Transactions on Software Engineering*, vol. 15, no. 7, pp. 902–925, 1989.

[20] B. Boehm, "Anchoring the Software Process," *IEEE Software*, vol. 13, no. 4, pp. 73–82, 1996.

[21] B. Boehm, C. Abts, A. Brown, S. Chulani, B. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece, *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice-Hall, 2000.

[22] G. Booch, J. Rumbaugh, and I. Jacobson, *The Universal Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

[23] L. Briand, V. Basili, and W. Thomas, "A Pattern Recognition Approach for Software Engineering Data Analysis," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 931–942, 1992.

[24] F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1995.

[25] Carnegie Mellon Software Engineering Institute, *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, MA: Addison-Wesley, 1995.

[26] G. Casella and R. Berger, *Statistical Inference*. Pacific Grove, CA: Duxbury, 2001.

[27] C. Chang, C. Chao, and T. Nguyen, "Software Project Management Net: A New Methodology on Software Management," *Proceedings of International Computer Software and Applications Conference*, pp. 534–539, 1998.

[28] S. Chulani, B. Boehm, and B. Steece, "Bayesian Analysis of Empirical Software Engineering Cost Models," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 573–583, 1999.

[29] L. Chung and K. Chan, "Integrating Project Planning and Process Modeling for Software Development," *IEEE Symposium on Application-Specific Systems and Software Engineering and Technology*, pp. 276–279, 1999.

[30] P. Congdon, *Bayesian Statistical Modeling.* New York, NY: John Wiley & Sons, 2001.

[31] S. Conger, *The New Software Engineering.* Boston, MA: International Thomson Publishing, 1994.

[32] R. Connor and J. Jenkins, "Using Agents for Distributed Software Project Management," *IEEE Eighth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp. 54–60, 1999.

[33] S. Conte, H. Dunsmore, and V. Shen, *Software Engineering Metrics and Models.* Menlo Park, CA: Benjamin Cummings, 1986.

[34] R. Courtney and D. Gustafson, "Shotgun Correlations in Software Measures," *Software Engineering Journal*, vol. 8, no. 1, pp. 5–13, 1993.

[35] A. Cuelenaere, M. van Genuchten, and F. Heemstra, "Calibrating a Software Cost Estimation Model: Why and How," *Information and Software Technology*, vol. 29, pp. 558–567, 1994.

[36] M. Cusumano and C. Kemerer, "A Quantitative Analysis of U.S. and Japanese Practice and Performance in Software Development," *Management Science*, vol. 36, no. 11, pp. 1384–1406, 1990.

[37] T. DeMarco and T. Lister, *Peopleware.* New York: Dorset House, 1987.

[38] M. Dorfman and R. Thayer, *Software Engineering.* Los Alamitos, CA: IEEE Computer Science Press, 1997.

[39] T. Escamilla, D. Simmons, and N. Ellis, "The Management of Uncertainty in Commercial Expert System Building Tools," *Proceedings of the Second Intelligence, IEEE Computer Society*, pp. 471–477, 1990.

[40] R. E. Fairley and R. H. Thayer, *Work Breakdown Structures.* Los Alamitos, CA: IEEE Computer Society Press, 1997.

[41] G. Finnie and G. Wittig, "A Comparison of Software Effort Estimation Techniques: Using Function Points with Neural Networks, Case-Based Reasoning and Regression Models," *Journal of Systems Software*, vol. 39, pp. 281–289, 1997.

[42] Q. Fleming and J. Koppelman, *Earned Value Project Management.* Newtown Square, Pennsylvania: Project Management Institute, 2001.

[43] A. Gelman, J. Carlin, H. Stern, and D. Rubin, *Bayesian Data Analysis.* Boca Raton, FL: Chapman & Hall, 1995.

[44] W. Goethert, E. Bailey, and M. Busby, *Software Effort and Schedule Measurement: A Framework for Counting Staff Hours and Reporting Schedule Information.* Pittsburgh, PA: Software Engineering Institute, 1992.

[45] N. Goth, "Bottlenecked in Bangalore," *Red Herring*, vol. 2, pp. 17–23, 1997.

[46] J. Grundy and J. Hosking, "Serendipity: Integrated Environment Support for Process Modeling, Enactment and Work Coordination," *Automated Software Engineering: Special Issue on Process Technology*, vol. 5, no. 1, pp. 27–60, 1998.

[47] M. Harandi, "Building a Knowledge-Based Software Development Environment," *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 5, pp.

862–868, 1988.

[48] F. J. Heemstra, "Software Cost Estimation," *Information on Software Technology*, vol. 34, no. 10, pp. 627–639, 1992.

[49] K. Huang and D. Simmons, "An Object Knowledge Canonical Form for Knowledge Reuse," *Expert Systems with Applications: An International Journal*, vol. 10, pp. 135–146, 1996.

[50] K. Huarng and D. Simmons, "Integration of Uncertainty Management Systems with Object-Oriented Expert System Building Tools," *Proceedings of the Eighteenth Annual International Computer Software and Applications Conference*, pp. 51–56, 1994.

[51] W. S. Humphrey, *Managing the Software Process.* Reading, MA: Addison-Wesley, 1989.

[52] W. S. Humphrey, *A Discipline for Software Engineering.* Reading, MA: Addison-Wesley, 1995.

[53] K. Hutchens, M. Oudshoorn, and K. Maciunas, "Web-based Software Engineering Process Management," *IEEE Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, pp. 676–685, 1997.

[54] L. Jain, *Knowledge-Based Intelligent Techniques in Industry.* Boca Raton, FL: CRC Press, 1999.

[55] D. Jeffrey, "Time-Sensitive Cost Models in the Commercial MIS Environment," *IEEE Transactions on Software Engineering*, vol. 13, no. 7, pp. 852–859, 1987.

[56] A. Jenkins, J. Naumann, and J. Wetherbe, "Empirical Investigation of Systems Developed Practices and Results," *Information Management*, vol. 7, pp. 73–82, 1984.

[57] C. Jones, "Management Tools and Software Failures and Success," *CrossTalk*, vol. 7, no. 10.

[58] C. Jones, "Determining Software Schedules," *IEEE Computer Magazine*, vol. 28, no. 2, pp. 73–75, 1995.

[59] C. Jones, "Patterns of Large Software Systems: Failure and Success," *IEEE Computer*, vol. 28, no. 3, pp. 86–87, 1995.

[60] C. Jones, *Applied Software Measurement: Assuring Productivity and Quality.* New York, NY: McGraw-Hill, 1996.

[61] G. Joseph and R. Gary, *Expert Systems: Principles and Programming.* Boston: PWS Publishing Company, 1998.

[62] M. Katrina, *Applied Statistics for Software Managers.* Upper Saddle River, NJ: Prentice-Hall, 2002.

[63] C. Kemerer, "An Empirical Validation of Software Cost Estimation Models," *Communications of the ACM*, vol. 30, no. 5, pp. 416–429, 1987.

[64] C. Kemerer and M. Patrick, *Staffing Factor in Software Cost Estimation Models.* New York, NY: Windcrest/McGraw-Hill, 1993.

[65] W. Keuffel, "People Based Processes: A RADical Concept," *Software Development*, vol. 4, pp. 27–30, 1995.

[66] B. Kitchenham, "Empirical Studies of Assumptions That Underlie Software Cost-Estimation Models," *Information and Software Technology*, vol. 34, no. 4, pp. 211–218, 1992.

[67] B. Kitchenham and K. Kansala, "Inter-Item Correlation Among Function Points," *Proceedings of the 15th International Conference on Software Engineering*, pp. 477–480, 1993.

[68] B. Kitchenham and N. Taylor, "Software Cost Models," *ICL Technology Journal*, vol. 4, no. 3, pp. 73–102, 1984.

[69] T. Kuhn, *The Structure of Scientific Revolutions*. Chicago: The University of Chicago Press, 1996.

[70] M. J. Lanigan, "Project Control with Delta Analysis," *Engineering Management Journal*, vol. 4, no. 1, pp. 36–42, 1994.

[71] M. Lawrence, "Programming Methodology, Organizational Environment, and Programming Productivity," *Journal of Systems and Software*, vol. 2, no. 3, pp. 257–269, 1981.

[72] A. L. Lederer and J. Prasad, "Information System Cost Estimating: A Current Assessment," *Journal of Information*, vol. 8, no. 1, pp. 22–33, 1993.

[73] J. Lehman, "How Software Projects Are Really Managed," *Datamation*, vol. 3, pp. 119–129, 1979.

[74] G. Low and R. Jeffery, "Function Points in the Estimation and Evaluation of the Software Process," *IEEE Transactions on Software Engineering*, vol. 16, no. 1, pp. 64–71, 1990.

[75] F. McGrath, *16 Critical Software Practices for Performance-Based Management.* Norfolk, VA: Software Program Management Network, 1999.

[76] Y. Miyazaki and K. Mori, "COCOMO Evaluation and Tailoring," *Proceedings of the Eighth International Conference on Software Engineering*, pp. 292–299, 1985.

[77] T. Mukhopadhyav, S. Vicinanza, and M. Prietula, "Estimating the Feasibility of a Case-Based Reasoning Model for Software Effort Estimation," *MIS Quarterly*, vol. 16, pp. 155–171, 1992.

[78] R. Park, *Software Size Measurement: A Framework for Counting Source Statements.* Pittsburgh, PA: Software Engineering Institute, 1992.

[79] M. C. Paulk, *Key Practices of the Capability Maturity Model.* Reading, MA: Addison-Wesley, 1993.

[80] D. Phillips, "Project Management: Filling in the Gaps," *IEEE Software*, vol. 13, no. 4, pp. 17–18, 1996.

[81] D. Phillips, *The Software Project Management Handbook, Principles That Work at Work.* Los Alamitos, CA: IEEE Computer Society, 2000.

[82] R. B. Pittman, "Product & Project Planning: Key to Getting It Right the First Time," *IEEE WESCON/96*, pp. 91–95, 1996.

[83] H. Poincaré, *La Science and l'Hypothèse.* Paris: Flammarion [Reprinted in Champs 1989], 1902.

[84] U. Pooch and P. Gehring, *Advances in Computer Programming Management.* Philadelphia, PA: Heyden, 1980.

[85] R. Pressman, *Software Engineering: A Practitioner's Approach.* New York, NY: McGraw-Hill, 1997.

[86] L. Putnam, "Trends in Measurement, Estimation, and Control," *IEEE Software*, vol. 8, no. 2, pp. 105–107, 1991.

[87] L. Putnam and D. Paulish, *Software Metrics: A Practitioner's Guide to Improved Product Development.* London: Chapman & Hall, 1993.

[88] L. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," *IEEE Transactions on Software Engineering*, vol. 4, no. 4, pp. 345–361, 1978.

[89] J. Reel, "Critical Success Factors in Software Projects," *IEEE Software*, vol. 16, no. 3, pp. 18–23, 1999.

[90] H. Rehesaar and E. Beames, "Project Plans and Times Budgets in Information System Projects," *International Conference on Software Engineering: Education & Practice*, pp. 120–124, 1998.

[91] C. Robert, *The Bayesian Choice.* New York, NY: Springer, 2001.

[92] M. D. Rosenau and M. D. Lewin, *Software Project Management, Step by Step.* Belmont, CA: Lifetime Learning Publications, 1984.

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

[94] B. Samson, D. Ellison, and P. Dugard, "Software Cost Estimation Using an Albus Perception (CMAC)," *Journal of Systems Software*, vol. 12, pp. 209–218, 1997.

[95] K. Sengupta and T. Abdel-Hamid, "Impact of Schedule Estimation on Software Project Behavior," *IEEE Software*, vol. 3, no. 4, pp. 70–75, 1986.

[96] A. Shenhar, "Strategic Project Management: The New Framework," *Portland IEEE International Conference on Management of Engineering and Technology*, pp. 382–386, 1999.

[97] D. Simmons, "A Win-Win Metric Based Software Management Approach," *IEEE Transactions on Engineering Management*, vol. 39, no. 1, pp. 32–41, 1992.

[98] D. Simmons, N. Ellis, and T. Escamilla, "Manager Associate," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 3, pp. 426–438, 1993.

[99] D. Simmons, N. Ellis, H. Fujihara, and W. Kuo, *Software Measurement—A Visualization Tool Kit for Process Control and Process Improvement*. Upper Saddle River, NJ: Prentice-Hall, 1998.

[100] D. Simmons and C. Wu, "Plan Tracking Knowledge Base," *Proceedings of the Twenty-Fourth Annual International Computer Software and Applications Conference*, pp. 299–304, 2000.

[101] D. B. Simmons, "Communication: A Software Group Productivity Dominator," *Software Engineering Journal*, vol. 6, no. 6, pp. 454–462, 1991.

[102] C. R. Snyder, "The Software Development Plan: A Key to Achieve SEI Capability Maturity Model Compliance," *ACM*, vol. 3, no. 7, pp. 106–112, 1992.

[103] I. Sommerville, *Software Engineering*. Harlow, England: Addison-Wesley, 1996.

[104] K. Srinivasan and D. Fisher, "Machine Learning Approach to Estimating Development Effort," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 126–137, 1995.

[105] K. Standish, "Chaos," *Open Computing*, vol. 7, pp. 4–7, 1995.

[106] A. Tatnall and P. Shackleton, "IT Project Management: Developing On-going Skills in the Management of Software Development Projects," *Proceedings of Software Engineering: Education and Practice*, pp. 400–405, 1996.

[107] R. C. Tausworthe, "The Work Breakdown Structure in Software Project Management," *Systems and Software*, vol. 81, pp. 181–186, 1980.

[108] J. Vosburgh, B. Curtis, R. Wolverton, B. Albert, H. M. S. Hoben, and Y. Liu, "Productivity Factors and Programming Environments," *Proceedings of Seventh International Conference on Software Engineering*, pp. 143–152, 1984.

[109] F. Walkerden and R. Jeffery, "Software Cost Estimation: A Review of Models, Process, and Practice," *Advances in Computers*, vol. 44, pp. 59–125, 1997.

[110] C. Walston and C. Felix, "A Method of Programming Measurement and Estimation," *IBM System Journal*, vol. 16, no. 1, pp. 54–73, 1977.

[111] R. Whiting, "News Front: Development in Disarray," *Software Magazine*, vol. 10, p. 20, 1998.

[112] G. Wittig and G. Finnie, "Using Artificial Neural Networks and Function Points to Estimate 4GL Software Development Effort," *Australian Journal of Information Systems*, vol. 1, no. 2, pp. 87–94, 1994.

[113] C. Wu and D. Simmons, "Software Project Planning Associate (SPAA): A Knowledge Based Approach for Dynamic Software Project Planning and Tracking," *Proceedings of the Twenty-Fourth Annual International Computer Software and Applications Conference*, pp. 305–310, 2000.

[114] S. Yun and D. B. Simmons, "Continuous Productivity Assessment and Effort Prediction Based on Bayesian Analysis," *Proceedings of the Twenty-Eighth Annual International Computer Software and Applications Conference*, pp. 44–49, 2004.

[115] S. Yun and D. B. Simmons, "Continuous Productivity Assessment and Prediction Tool," *Proceedings of International Conference on Software Engineering of Research and Practice*, pp. 366–371, 2004.

APPENDIX A

16 CRITICAL SOFTWARE PRACTICES FOR PERFORMANCE-BASED MANAGEMENT

PROJECT INTEGRITY

1. Adopt Continuous Program Risk Management

Practice Essentials

1. Risk management is a continuous process beginning with the definition of the concept and ending with system retirement.

2. Risk management is a program responsibility impacting on and supported by all organizational elements.

3. All programs need to assign a risk officer as a focal point for risk management and maintain a reserve to enable and fund risk mitigation.

4. Risk need to be identified and managed across the life of the program.

5. All risks identified should be analyzed, prioritized-by impact and likelihood of occurrence-and tracked through an automated risk management tool.

6. High-priority risks need to be reported to management on a frequent and regular basis.

Implementation Guidelines

1. Risk management should commence prior to contract award and shall be a factor in the award process.

2. The DEVELOPER needs to establish and implement a project Risk Management Plan that, at a minimum, defines how points 3 through 8 will be implemented. The plan and infrastructure (tools, organizational assignments, and management procedures) will be agreed to by the ACQUIRER and the DEVELOPER and need to be placed under configuration management (CM).

3. DEVELOPER and ACQUIRER senior management should establish reporting mechanisms and employee incentives in which all members of the project staff are encouraged to identify risks and potential problems and are rewarded when risks and potential problems are identified early. The ACQUIRER needs to address risk management explicitly in its contract award fee plan, and the DEVELOPER needs to provide for the direct distribution to all employees in furtherance of establishing and maintaining a risk culture.

4. Risk identification should be accomplished in facilitated meetings attended by project personnel most familiar with the area for which risks are being identified. A person familiar with problems from similar projects in this area in the past should participate in these meetings when possible. Risk identification should include risks throughout the life cycle in at least the areas of cost, schedule, technical, staffing, external dependencies, supportability, and maintainability and should include organizational and programmatic political risks. Risk identification need to be updated at least monthly. Identified risks should be characterized in terms of their likelihood of occurrence and the impact of their occurrence. Risk mitigation activities need to be included in the project's task activity network.

5. Both the DEVELOPER and the ACQUIRER should designate and assign senior members of the technical staff as risk officers to report directly to their respective

program managers and should charter this role with independent identification and management of risks across the program and grant the authority needed to carry out this responsibility.

6. Each medium-impact and high-impact risk should be described by a complete Risk Control Profile.

7. Periodically updated estimates of the cost and schedule at completion should include probable costs and schedule impact due to risk items that have not yet been resolved.

8. The DEVELOPER and ACQUIRER risk officers need to update the risk data and database on the schedule defined in the Risk Management Plan. All risks intended for mitigation and any others that are on the critical path and their status against the mitigation strategy should be summarized. Newly identified risks should go through the same processes as the originally identified risks.

2. Estimate Cost and Schedule Empirically

Practice Essentials

1. Initial software estimates and schedules should be looked on as high risk due to the lack of definitive information available at the time they are defined.

2. The estimates and schedules should be refined as more information becomes available.

3. At every major program review, costs-to-complete and rescheduling should be presented to identify deviations from the original cost and schedule baselines and to anticipate the likelihood of cost and schedule risks occurring.

4. All estimates should be validated using a cost model, a sanity check should be conducted comparing projected resource requirements, and schedule commitments should be made.

5. Every task within a work breakdown structure (WBS) level need to have an associated cost estimate and schedule. These tasks should be tracked using earned value.

6. All costs estimates and schedules need to be approved prior to the start of any work.

Implementation Guidelines

1. Estimate the cost, effort, and schedule for a project for planning purposes and as a yardstick for measuring performance (tracking). Software size and cost need to be estimated prior to beginning work on any incremental release.

2. Software cost estimation should be a reconciliation between a top-down estimate (based on an empirical model; e.g., parametric, cost) and a bottom-up engineering estimate.

3. Software cost estimation should also be subjected to a "sanity check" by comparing it with industry norms and specifically with the DEVELOPER's past performance in areas such as productivity and percentage of total cost in various functions and project phases.

4. All of the software costs need to be associated with the appropriate lower-level software tasks in the project activity network. Allocate the estimated total project labor effort among all the tasks in the activity network.

3. Use Metrics to Manage

Practice Essentials

1. All programs should have in place a metrics program to monitor issues and determine the likelihood of risks occurring.

2. Metrics should be defined as part of definition of process, identification of risks or issues, or determination of project success factors.

3. All metrics definition need to include description, quantitative bounds, and expected areas of application.

4. All programs need to assign an organizational responsibility for identification, collection, analysis, and reporting of metrics throughout the program's life.

5. Metrics information should be used as one of the primary inputs for program decisions.

6. The metrics program needs to be continuous.

Implementation Guidelines

1. Every project should have a project plan with a detail activity network that defines the process the team will follow, organizes and coordinates the work, and estimates and allocates cost and schedule among tasks. The plan should be broad enough to include each sub-process/phase. The project plan needs to include adequate measurement in each of these five categories. early indications of problems, the quality of the products, the effectiveness of the processes, the conformance to the process, and the provision of a basis for future estimation of cost, quality, and schedule.

2. Metrics should be sufficiently broad based. Data should be collected for each process/phase to provide insight into the above 5 categories.

3. To use these metrics effectively, thresholds need to be established for these metrics. These thresholds should be estimated initially using suggested industry norms for various project classes. Local thresholds will evolve over time, based upon experience (see 1.e above). Violation of a threshold value should trigger further analysis and decision making.

4. Examples of data, initial thresholds, and analysis of size, defect, schedule, and effort metrics can be found at http://www.qsm.com.

5. Continuous data on schedule, risks, libraries, effort expenditures, and other measures of progress should be available to all project personnel along with the latest revision of project plans.

4. Track Earned Value

Practice Essentials

1. Earned value project management requires a work breakdown structure, work packages, activity networks at every WBS level, accurate estimates, and implementation of a consistent and planned process.

2. Earned value requires each task to have both entry and exit criteria and a step to validate that these criteria have been met prior to the award of the credit.

3. Earned value credit is binary with zero percent being given before task completion and 100% when completion is validated.

4. Earned value metrics need to be collected on a frequent and regular basis consistent with the reporting cycle required with the WBS level. (At the lowest level

of the work package, the earned value reporting should never be less frequent than 2 weeks).

5. Earned value, and the associated budgets schedules, and WBS elements need to be replanned whenever material changes to the program structure are required (e.g., requirements, growth, budget changes, schedule issues, organizational change).

6. Earned value is an essential indicator and should be used as an essential metric by the risk management process.

Implementation Guidelines

1. Progress towards producing the products should be measured within the designated cost and schedule allocations.

2. THE DEVELOPER should develop and maintain a hierarchical task activity network based on allocated requirements that includes the tasks for all effort that will be charged to the program. All level of effort (LOE) tasks need to have measurable milestones. All tasks that are not LOE should explicitly identify the products produced by the task and have explicit and measurable exit criteria based on these products.

3. No task should have a budget or planned calendar time duration that is greater than the cost and schedule uncertainty that is acceptable for the program. The goal for task duration is no longer than two calendar weeks of effort.

4. Each task that consumes resources needs to have a cost budget allocated to it and the corresponding staff and other resources that will consume this budget. Staff resources should be defined by person hours or days for each labor category working on the task.

5. For each identified significant risk item, a specific risk mitigation/resolution task should be defined and inserted into the activity network.

6. The cost reporting system for the total project needs to segregate the software effort into software tasks so that the software effort can be tracked separately from the non-software tasks.

7. Milestones for all external dependencies should be included in the activity network.

8. Earned value metrics need to be collected for each schedule level and be made available to all members of the DEVELOPER and government project teams monthly. These metrics are: a comparison of Budgeted Cost of Work Scheduled (BCWS), Budgeted Cost of Work Performed (BCWP), and Actual Cost of Work Performed (ACWP). A comparison of BCWP and ACWP, a Cost Performance Index, a Schedule Performance Index, and a To-Complete Cost Performance Index.

9. The lowest-level schedules should be statused weekly.

10. The high-level schedules should be statused at least monthly.

11. Earned value reports should be based on data that is no more than two weeks old.

5. Track Defects against Quality Targets

Practice Essentials

1. All programs need to have pre-negotiated quality targets, which is an absolute requirement to be met prior to acceptance by the customer.

2. Programs should implement practices to find defects early in the process and as close in time to creation of the defect as possible and should manage this defect rate against the quality target.

3. Metrics need to be collected as a result of the practices used to monitor defects, which will indicate the number of defects, defect leakage, and defect removal efficiency.

4. Quality targets need to be redefined and renegotiated as essential program conditions change or customer requirements are modified.

5. Compliance with quality targets should be reported to customers on a frequent and regular basis, along with an identification of the risk associated with meeting these targets at delivery.

6. Meeting quality targets should be a subject at every major program review.

Implementation Guidelines

1. The ACQUIRER and the DEVELOPER need to establish quality targets for subsystem software depending on its requirements for high integrity. A mission-critical/safety-critical system may have different quality targets for each subsystem component. System Quality Assurance needs to monitor quality targets and report defects as per the Quality Plan.

2. Quality targets can be under change control and established at the design, coding, integration, test, and operational levels.

3. Quality targets should address the number of defects by priority and by their fix rate.

4. Actual quality or defects detected and removed should be tracked against the quality targets.

5. Periodic estimates of the cost and schedule at completion should be based on the actual versus targeted quality.

6. Treat People-as the Most Important Resource

Practice Essentials

1. A primary program focus should be staffing positions with qualified personnel and retaining this staff through the life of the project.

2. The program should not implement practices (e.g., excessive unpaid overtime) that will force voluntary staff turnover.

3. The staff should be rewarded for performance against expectations and program requirements.

4. Professional growth opportunities such as training should be made available to the staff.

5. All staff members need to be provided facilities, tools, and work areas adequate to allow efficient and productive performance of their responsibilities.

6. The effectiveness and morale of the staff should be a factor in rewarding management.

Implementation Guidelines

1. DEVELOPER senior management needs to work to ensure that all projects maintain a high degree of personnel satisfaction and team cohesion and should identify and implement practices designed to achieve high levels of staff retention

as measured by industry standards. The DEVELOPER should employ focus groups and surveys to assess employee perceptions and suggestions for change.

2. DEVELOPER senior management should provide the project with adequate staff, supported by facilities and tools to develop the software system efficiently. Employee focus groups and surveys should be used to assess this adequacy.

3. The training of DEVELOPER and ACQUIRER personnel should include training according to a project training plan in all the processes, development and management tools, and methods specified in the software development plan.

4. The DEVELOPER and the ACQUIRER should determine the existing skills of all systems, software, and management personnel and provide training, according to the needs of each role, in the processes, development and management tools, and methods specified in the Software Development Plan (SDP)

CONSTRUCTION INTEGRITY

7. Adopt Life Cycle Configuration Management

Practice Essentials

1. All programs, irrespective of size, need to manage information through a preplanned configuration management (CM) process.

2. CM has two aspects: formal CM, which manages customer-approved baseline information, and development CM, which manages shared information not yet approved by the customer.

3. Both formal and development CM should uniquely identify managed information, control changes to this information through a structure of boards, provide status of all information either under control or released from CM, and conduct

ongoing reviews and audits to ensure that the information under control is the same as that submitted.

4. The approval for a change to controlled information must be made by the highest-level organization which last approved the information prior to placing it under CM.

5. CM should be implemented in a centralized library supported by an automated tool.

6. CM needs to be a continuous process implemented at the beginning of a program and continuing until product retirement.

Implementation Guidelines

1. CM plans need to be developed by the ACQUIRER and the DEVELOPER to facilitate management control of information they own. The CM procedures of the ACQUIRER serve as the requirements for the CM plan that describes and documents how the DEVELOPER will implement a single CM process. This plan should control formal baselines and will include engineering information, reports, analysis information, test information, user information, and any other information approved for use or shared within the program. The CM process should include DEVELOPER-controlled and -developed baselines as well as ACQUIRER-controlled baselines. It should also include release procedures for all classes of products under control, means for identification, change control procedures, status of products, and reviews and audits of information under CM control. The CM plan needs to be consistent with other plans and procedures used by the project.

2. The two types of baselines managed by CM are developmental and formal. Developmental baselines include all software, artifacts, documentation, tools, and other products not yet approved for delivery to the ACQUIRER but essential for successful production. Formal baselines are information/products (software, artifacts, or documentation) delivered and accepted by the ACQUIRER. Developmental baselines are owned by the DEVELOPER while formal baselines are owned by the ACQUIRER.

3. All information placed under CM as a result of meeting task exit criteria need to be uniquely identified by CM and placed under CM control. This includes software, artifacts, documents, commercial off-the-shelf (COTS), government off-the-shelf (GOTS), operating systems, middleware, database management systems, database information, and any other information necessary to build, release, verify, and/or validate the product.

4. The CM process should be organizationally centered in a project library. This library will be the repository (current and historical) of all controlled products. The ACQUIRER and the DEVELOPER will implement an organizationally specific library. The library(s) will be partitioned according to the level of control of the information.

5. All information managed by CM is subject to change control. Change control consists of: Identification Reporting Analysis Implementation

6. The change control process needs to be implemented through an appropriate change mechanism tied to who owns the information: Change control boards, which manage formal baseline products. Interface boards, which manage jointly owned information Engineering review boards, which manage DEVELOPER-

controlled information.

7. Any information released from the CM library should be described by a Version Description Document (Software Version Description under 498). The version description should consist of any inventory of all components by version identifier, an identification of open problems, closed problems, differences between versions, notes and assumptions, and build instructions. Additionally, each library partition should be described by a current version description that contains the same information.

8. Manage and Trace Requirements

Practice Essentials

1. Before any design is initiated, requirements for that segment of the software need to be agreed to.

2. Requirements tracing should be a continuous process providing the means to trace from the user requirement to the lowest level software component.

3. Tracing shall exist not only to user requirements but also between products and the test cases used to verify their successful implementation.

4. All products that are used as part of the trace need to be under configuration control.

5. Requirements tracing should use a tool and be kept current as products are approved and placed under CM.

6. Requirements tracing should address system, hardware, and software and the process should be defined in the system engineering management plan and the software development plan.

Implementation Guidelines

1. The program needs to define and implement a requirements management plan that addresses system, hardware, and software requirements. This plan should be linked to the SDP.

2. All requirements need to be documented, reviewed, and entered into a requirements management tool and put under CM. This requirements information should be kept current.

3. The CM plan should describe the process for keeping requirements data internally consistent and consistent with other project data.

4. Requirements traceability needs to be maintained through specification, design, code, and testing.

5. Requirements should be visible to all project participants.

9. Use System-Based Software Design

Practice Essentials

1. All methods used to define system architecture and software design should be documented in the system engineering management plan and software development plan and be frequently and regularly evaluated through audits conducted by an independent program organization.

2. Software engineering needs to participate in the definition of system architectures and should provide an acceptance gate before software requirements are defined.

3. The allocation of system architecture to hardware, software, or operational procedures needs to be the result of a predefined engineering process and be tracked through traceability and frequent quality evaluations.

4. All agreed to system architectures, software requirements, and software design decisions should be placed under CM control when they are approved for program implementation.

5. All architecture and design components need to be approved through an inspection prior to release to CM. This inspection should evaluate the process used to develop the product, the form and structure of the product, the technical integrity, and the adequacy to support future applications of the product to program needs.

6. All system architecture decisions should be based on a predefined engineering process and trade studies conducted to evaluate alternatives.

Implementation Guidelines

1. The DEVELOPER should ensure that the system and software architectures are developed and maintained consistent with standards, methodologies, and external interfaces specified in the system and software development plans.

2. Software engineers need to be an integral part of the team performing systems engineering tasks that influence software.

3. Systems engineering requirements trade studies should include efforts to mitigate software risks.

4. System architecture specifications need to be maintained under CM.

5. The system and software architecture and architecture methods need to be consistent with each other.

6. System requirements, including derived requirements, need to be documented and allocated to hardware components and software components.

7. The requirements for each software component in the system architecture and derived requirements need to be allocated among all components and interfaces of the software component in the system architecture.

10. Ensure Data and Database Interoperability

Practice Essentials

1. All data and database implementation decisions should consider interoperability issues and, as interoperability factors change, these decisions should be revisited.

2. Program standards should exist for database implementation and for the data elements that are included. These standards should include process standards for defining the database and entering information into it and product standards that define the structure, elements, and other essential database factors.

3. All data and databases should be structured in accordance with program requirements, such as the DII COE, to provide interoperability with other systems.

4. All databases shared with the program need to be under CM control and managed through the program change process.

5. Databases and data should be integrated across the program with data redundancy kept to a minimum.

6. When using multiple COTS packages, compatibility of the data/referential integrity mechanisms need to be considered to ensure consistency between databases.

Implementation Guidelines

1. The DEVELOPER needs to ensure that data files and databases are developed with standards and methodologies.

2. The DEVELOPER needs to ensure that data entities and data elements are consistent with the DoD data model.

3. All data and databases should be structured in compliance with DII COE to provide interoperability with other systems.

4. Data integrity and referential integrity should be maintained automatically by COTS DBMSs or other COTS software packages. The DEVELOPER should avoid developing its package, if at all possible. Before selecting multiple COTS software packages, the DEVELOPER should study the compatibility of the data/referential integrity mechanisms of these COTS packages and obtain assurance from the COTS vendors first.

5. Unnecessary data redundancy should be reduced to minimum.

6. Data and databases should be integrated as much as possible. Except data for temporary use or for analysis/report purposes, each data item should be updated only once, and the changes should propagate automatically everywhere.

11. Define and Control Interfaces

Practice Essentials

1. Before completion of system-level requirements, a complete inventory of all external interfaces needs to be completed.

2. All external interfaces need to be described as to source, format, structure, content, and method of support and this definition, or interface profile, needs to be placed under CM control.

3. Any changes to this interface profile should require concurrence by the interface owners prior to being made.

4. Internal software interfaces should be defined as part of the design process and managed through CM.

5. Interfaces should be inspected as part of the software inspection process.

6. Each software or system interface needs to be tested individually and a test of interface support should be conducted in a stressed and anomalous test environment.

Implementation Guidelines

1. All internal and external interfaces need to be documented and maintained under CM control.

2. Changes to interfaces require concurrence by the interface owners prior to being made.

3. Milestones related to external interfaces should be tracked in the project activity network. [Keep these milestones off your critical path.]

4. Subsystem interfaces should be controlled at the program level.

12. Design Twice, Code Once

Practice Essentials

1. All design processes should follow methods documented in the software development plan.

2. All designs need to be subject to verification of characteristics, which are included as part of the design standards for the product produced.

3. All designs should be evaluated through a structured inspection prior to release to CM. This inspection should consider reuse, performance, interoperability, security, safety, reliability, and limitations.

4. Traceability needs to be maintained through the design and verified as part of the inspection process.

5. Critical components should be evaluated through a specific white-box test level step.

6. Design can be incrementally specified when an incremental release or evolution life cycle model is used provided the CM process is adequate to support control of incremental designs and the inspection process is adapted to this requirement.

Implementation Guidelines

1. When reuse of existing software is planned, the system and software architectures should be designed to facilitate this reuse.

2. When an incremental release life cycle model is planned, the system and software architectures need to be completed in the first release or, at most,extended in releases after the first without changes to the architecture of previous releases.

3. The system and software architectures will be verified using methods specified in the SDP. This verification will be conducted during a structured inspection of the software architecture and will include corroboration that the architecture will support all reuse, performance, interoperability, security, safety, and reliability requirements. The architecture will be under CM.

13. Assess Reuse Risks and Costs

Practice Essentials

1. The use of reuse components, COTS, GOTS, or any other non-developmental items (NDI) should be treated as a risk and managed through risk management.

2. Application of reuse components, COTS, GOTS, or any other NDI will be made only after successful completion of a NDI acceptance inspection. This inspection needs to consider the process used to develop it, how it was document, number of users, user experience, and compliance with essential program considerations such as safety or security.

3. Before a decision is made to reuse a product or to acquire COTS, GOTS, or NDI, a complete cost trade-off should be made considering the full life cycle costs, update requirements, maintenance costs, warranty and licensing costs, and any other considerations which impact use of the product throughout its life cycle.

4. All reuse products, COTS, GOTS, or NDI decisions should be based on architectural and design definitions and be traceable back to an approved user requirement.

5. All reuse components, COTS, and COTS need to be tested individually first

against program requirements and in an integrated software and system configuration prior to release for testing according to the program test plan.

6. Reuse, COTS, GOTS, and NDI decisions will be continuously revisited as program conditions change.

Implementation Guidelines

1. The DEVELOPER will establish a reuse plan for the integration of COTS, GOTS, and in-house software. This plan needs to include discussion and allocation of whom and by what process reused software code is tested, verified, modified, and maintained.

2. The reuse plan should be in the SDP and document an approach for evaluating and enforcing reused functionality against system requirements.

3. The reuse plan should suggest a system engineering process that identifies software requirements by taking existing, reusable software components into account.

4. The test plan should identify the testing of the integrated reused code.

5. When integrating COTS, GOTS, and in-house software, ensure accurate cost estimation of integrating the reused code into the system. The cost of integrating unmodified reused code is approximately one-third the cost of developing code without reuse.

6. The DEVELOPER and the ACQUIRER need to be able to plan for the estimated costs of obtaining the necessary development and run-time licenses over the system's life cycle and the maintenance/support critical to the product, including source code availability.

PRODUCT STABILITY AND INTEGRITY

14. Inspect Requirements and Design

Practice Essentials

1. All products that are placed under CM and are used as a basis for subsequent development need to be subjected to successful completion of a formal inspection prior to its release to CM.

2. The inspection needs to follow a rigorous process defined in the software development plan and should be based on agreed-to entry and exit criteria for that specific product.

3. At the inspection, specific metrics should be collected and tracked which will describe defects, defect removal efficiency, and efficiency of the inspection process.

4. All products to be placed under CM should be inspected as close to their production as feasible.

5. Inspections should be conducted beginning with concept definition and ending with completion of the engineering process.

6. The program needs to fund inspections and track rework savings.

Implementation Guidelines

1. The DEVELOPER will implement a formal, structured inspection/peer review process that begins with the first system requirements products and continue through architecture, design, code, integration, testing, and documentation products and plans. The plan needs to be documented and controlled as per the SDP.

2. The project should set a goal of finding at least 80% of the defects in every product undergoing a structured peer review or other formal inspection.

3. Products should not be accepted into a CM baseline until they have satisfactorily completed a structured peer review.

4. The DEVELOPER needs to collect and report metrics concerning the number of defects found in each structured peer review, the time between creating and finding each defect, where and when the defect was identified, and the efficiency of defect removal.

5. Successful completion of inspections should act as the task exit criteria for non-Level-of-Effort earned value metrics (and other metrics used to capture effectiveness of the formal inspection process) and as gates to place items under increasing levels of CM control.

6. The DEVELOPER should use a structured architecture inspection technique to verify correctness and related system performance characteristics.

15. Manage Testing as a Continuous Process

Practice Essentials

1. All testing should follow a preplanned process, which is agreed to and funded.

2. Every product that is placed under CM should be tested by a corresponding testing activity.

3. All tests should consider not only a nominal system condition but also address anomalous and recovery aspects of the system.

4. Prior to delivery, the system needs to be tested in a stressed environment, nominally in excess of 150% of its rated capacities.

5. All test products (test cases, data, tools, configuration, and criteria) should be released through CM and be documented in a software version description document.

6. Every test should be described in traceable procedures and have pass-fail criteria included.

Implementation Guidelines

1. The testing process must be consistent with the RFP and the contract. The award fee should incentivize implementation of the testing practices described below.

2. The ACQUIRER and DEVELOPER need to plan their portion of the test process and document this plan with test cases and detailed test descriptions. These test cases should use cases based on projected operational mission scenarios.

3. The testing process should also include stress/load testing for stability purpose (i.e., at 95% CPU use, system stability is still guaranteed?)

4. The test plan should include a "justifiable testing stoppage criteria." This gives testers a goal. If your testing satisfies these criteria, then the product is ready for release.

5. The test process should thoroughly test the interfaces between any in-house and COTS functionality. These tests should include timing between COTS functionality and the bespoken functionality. The test plans need to pay serious attention to how to demonstrate that, if the COTS software fails, how to test that the rest of the software can recover adequately. This involves some very serious stress testing using fault injection testing.

6. Software testing should include a traceable white-box and other test process verifying implemented software against CM-controlled design documentation and the requirements traceability matrix.

7. A level of the white-box test coverage should be specified that is appropriate for the software being tested.

8. The white-box and other testing should use automated tools to instrument the software to measure test coverage.

9. All builds for white-box testing need to be done with source code obtained from the CM library.

10. Frequent builds require test automation, since more frequent compiles will force quick turnaround on all tests, especially during regression testing. However, this requires a high degree of test automation.

11. A black-box test of integration builds needs to include functional, interface, error recovery, stress, and out-of-bounds input testing.

12. Reused components and objects require high-level testing consistent with the operational/target environment.

13. Software testing includes a separate black-box test level to validate implemented software. All black-box software tests should trace to controlled requirements and be executed using software built from controlled CM libraries.

14. In addition to static requirements, a black-box test of the fully integrated system will be against scenarios-sequences of events designed to model field operation.

15. Performance testing for systems (e.g., performing 10,000 tests/second still yields response times under 2 seconds) should be tested as an integral part of the black-box test process.

16. An independent QA team should periodically audit selected test cases, test traceability, test execution, and test reports providing the results of this audit to the ACQUIRER. (The results of this or similar audits may be used as a factor in the calculation of Award Fee.)

17. Each test developed needs to include pass/fail criteria.

16. Compile and Smoke Test Frequently

Practice Essentials

1. All tests should use systems that are built on a frequent and regular basis (nominally no less than twice a week).

2. All new releases should be regression tested by CM prior to release to the test organization.

3. Smoke testing should qualify new capability or components only after successful regression test completion.

4. All smoke tests should be based on a pre-approved and traceable procedure and run by an independent organization (not the engineers who produced it).

5. All defects identified should be documented and be subject to the program change control process.

6. Smoke test results should be visible and provided to all project personnel.

Implementation Guidelines

1. From the earliest opportunity to assess the progress of developed code, the DEVELOPER needs to use a process of frequent (one- to two-week intervals) software compile-builds as a means for finding software integration problems early.

2. It is required that a regression facility that incorporates a full functional test suite be applied with the build strategy.

3. Results of testing of each software build should be made available to all project personnel.

VITA

Seok Jun Yun was born in Seoul, Korea. He received a Bachelor of Science in environmental science at the Korean Military Academy in 1988. After graduation, he was commissioned as a Second Lieutenant in the R.O.K. Army and worked as a Field Artillery Officer at the 5th Division. He graduated from the Naval Postgraduate School at Monterey, CA, in 1995 and obtained a master's degree in computer science. Upon graduation, he got a position as a lecturer at the R.O.K. Military Academy. In February of 1997, he was selected by the R.O.K. Army HQ as a manager of the project to install the Video-On-Demand system at the Military Academy, which supports military cadets' language education. In November of 1997, he was assigned as a leader of the project to install the network system at the Military Academy to support the Intranet/Internet environment to faculty and cadets. After this project, he was assigned as a computer system engineer at the Computing Center at the Military Academy. In August of 2000, he received a government scholarship to study for his Ph.D. in computer science in the United States. He received his Ph.D. in computer science in May 2005. He can be reached at Canaan Animal Hospital 461-61 Su-Yu 1 Dong Kang-Buk Gu Seoul Korea 142-875.