Practices for Fast and Flexible Software Development

by

Pearlin P. Cheung

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 24, 2002

[June 2002]

Author______________________________
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by_______________
Michael A. Cusumano
Thesis Supervisor

Accepted by______________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Practices for Fast and Flexible Software Development
by
Pearlin P. Cheung

Submitted to the
Department of Electrical Engineering and Computer Science

May 24, 2002

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The software industry has expanded worldwide due to the Internet boom. The software industry in each country has its own unique evolution due to cultural and historical differences. As the industry grew, the software community began to explore different methods of developing software. The purpose of this thesis is to determine the practices or processes that contribute to development performance and to give a survey of the development techniques utilized in different countries-mainly India, Japan, and the United States.

Thesis Supervisor: Michael A. Cusumano,
Title: Sloan Management Review Distinguished Professor.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I am thankful for all the support and guidance that I have received during this whole process.

I would like to thank my thesis advisor, Professor Michael Cusumano, for giving me the opportunity to work on such exciting research. Throughout the process, I learned a great deal about software development-the industry, the models, and the practices. Whenever I encountered problems or needed to discuss how the research should proceed, he always made himself available to help me despite his hectic schedule. I would also like to thank him for the generous research asssistantship.

I would also like to thank Professor Alan MacCormack from Harvard Business School and Professor Chris Kemerer from Katz Business School at the University of Pittsburgh. They provided much guidance during the research and writing of this thesis by fielding my numerous questions and cautioning me when I start heading in the wrong direction.

Bill Crandall from Hewlett-Packard has also been a great support with his invaluable advice and enthusiasm. He rallied together the large participation of many in HP and Agilent.

The survey would not have been possible without the help of Jeffrey Schiller and Jagruti Patel from MIT Information Systems. They facilitated and maintained the web survey.

I would also like to thank all the project teams that participated in this research. They have taken much time and effort to complete the extensive survey.

Lastly, I would like to thank my family and my good friend Steven Yang for their continued encouragement during my academic career at MIT.

The software industry has a global reach. According to Standard & Poor's Industry Surveys, in the year 2001, almost $200 billion USD was spent on packaged software worldwide and nearly $400 billion USD on software services [28]. Along with the expansion of the software industry, there has been a shift in the software community as to how software development should proceed. Departing from the waterfall model, incremental development methodologies have emerged.

1 Purpose

With multiple software models and methodologies available, the problem is to determine which software engineering practices allow for fast and flexible software development. The term "fast" means the development team is highly productive, and the term "flexible" means the project team can successfully incorporate late changes to the system without sacrificing quality.

Although many practitioners of various development methodologies believe that their approaches are productive, there has not been extensive research to validate their claims.

The objectives of this thesis are to measure the performance of software development projects and to determine which engineering practices strongly correlate with high performance. Performance can be measured through many factors such as timeliness, quality, customer satisfaction, and productivity.

This thesis will also examine the software development techniques used in the US, Japan, and India. These three countries have different approaches towards software development: incremental development has gained popularity in the United States, Japan takes a manufacturing approach to software development, and much outsourcing work takes place in India.

The thesis is arranged as follows: Section 2 presents background information regarding development models and methodologies followed by a survey of related work in Section 3. The research methodology of this study is presented in Section 4. Hypothesis and results are in Sections 6 and 7. The survey this study used for its data collection instrument is located in Appendix A.

2 Background

The waterfall model has long been the dominant software development process, but it has been slowly losing popularity due to the now fast-paced environment of software development. Responding to the increasing volatility of customer demands, new methodologies and models have emerged. These new methodologies allow for more flexibility in software development than the traditional waterfall method. Some of the adaptive methodologies are also known as lightweight or agile methodologies. Listed below are a few popular models and methodologies, as well as the classic waterfall model.

Waterfall Model

The waterfall model is a sequential development process that was widely used in the 70s and 80s [26]. The process is very structured and provides an orderly way to develop systems. From the beginning, the project team must thoroughly understand the system they are developing because this model does not adjust well to requirement changes. A project team using the waterfall model can backtrack to successive steps but backtracking any further is difficult. Unlike most lightweight methodologies, formal documentation is written for each stage throughout the entire process.

The first step to the waterfall model is to determine the system and software requirements, and to document the software requirements. Next, the project team makes a preliminary design of the system and simulates the development of a smaller version of the final system. After the project team conducts an analysis of the preliminary design, they finalize the program design. The next phases are coding and testing. The project team is divided into smaller teams that are each responsible for developing and testing a module. After the system is successfully integrated, a user manual is written and the system is delivered to the customer.

Spiral Model

The spiral model focuses on reducing project risk by eliminating risks incrementally through iterations [5]. The first step in each iteration is to determine the objectives of the system, multiple ideas for design or implementation, and constraints on the system. Next, the design ideas are evaluated to help identify risks. The next step is to resolve these risks, usually by creating a prototype. At the end of each iteration, the project team reviews the current iteration and plans the next phase or iteration. The remaining risks determine the next steps. If performance or user-interface risks prevail over the other remaining risks, then the project team continues the risk-reducing iterations. If development or control risks prevail, then the project team uses the waterfall or an equivalent model to complete the development process of implementation, integration, and testing.

Microsoft's Synch-and-Stabilize

Microsoft's approach to software development is synch-and-stabilize, which is a loosely organized process that is geared towards developing systems for the mass market [10]. There are three phases to synch-and-stabilize: planning, development, and stabilization.

The planning phase usually lasts more than a year for new products and up to three months for existing products. Product and program managers write a vision statement to help the developers understand the project. The vision statement also includes a list of prioritized features. The program managers and developers then write a specification document that outlines features, the architecture, and component interdependencies. Program managers create prototypes and conduct feasibility studies to explore design alternatives, which help them make better decisions related to specifications. A testing strategy is also formed. Program mangers then schedule milestone deadlines for the development phase, and arrange feature teams. Each feature team consists of one program manager, three to eight developers, and a tester for each developer. When the schedule is completed and the project plan approved, the project team moves onto the development phase.

The development phase consists of three to four milestones, each lasting about two to four months. The most important features are developed in the first milestone while the least critical are developed in the last milestone. Each milestone consists of coding, testing, and stabilizing (debugging and integration) stages. A "build master" conducts a build every day to help coordinate and update the main code. Developers also update their source code files daily in order to synchronize their development with the entire team. When a developer first develops a feature, he implements the feature into his latest copy of the source code. When he is done with implementation, his assigned tester conducts a "quick test" or "smoke test" to make sure that the developer's additions have not affected any other features in the main code. After successful testing, the developer can then check in his code to the main code. There is an internal release at the end of each milestone. After the last milestone release, there is a "visual freeze" where no major changes can be made to the user interface. "Feature complete" occurs when the system is functional but not bug-free. The system is "code complete" when features work according to the specifications.

Developers and testers continue to debug and test the system extensively. Beta releases are conducted to allow end-users to test the system. When no more critical bugs remain and the remaining bugs are scheduled to be fixed in the next version of the system, the "zero bug release" stage has been achieved. The system is then officially released to manufacturers. After the release, the project team writes a postmortem document to summarize what succeeded, what did not succeed, and suggestions for improvement.

Capabilities Maturity Model

In 1984, the US Defense Department formed the Software Engineering Institute (SEI) to help define a discipline for professional software engineering practices and to integrate new technologies and methods into software engineering practice. The result of one SEI project is the Capabilities Maturity Model [15]. Each stage of the Capabilities Maturity Model describes a different discipline or maturity level of software development found in organizations. With this model, an organization can determine on which maturity level of software development they stand and which areas they need to improve in order to proceed from one maturity level to the next. When an organization reaches the last maturity level, it is able to produce a product that meets customer needs within the original cost and schedule estimations. The main objective of using this model is to measure and control software processes to provide a "scientific foundation for continuous improvement" [15]. There are a total of five stages in the model: Initial Process, Repeatable Process, Defined Process, Managed Process, and Optimizing Process.

In the Initial Process, there is little project management, no quality assurance group, and no defined process to handle change. The defining aspect of the Repeatable Process is commitment control, which is achieved through project plans, scheduled reviews of cost and development progress, and orderly procedures for requirement changes. When an organization reaches the Defined Process, it has formed a solid foundation for development progress. At this level, an organization has established a progress group that solely focuses on improving the development process, an architecture that defines the technical and management roles in the development process, and a set of development methods and technologies. The Managed Process focuses on collecting quantitative data by using designated databases to gather and analyze the cost and quality measurements of each product. At the Optimizing Process level, data gathering is automated and the organization's focus is to improve the actual development process using the analyzed data. By the final level, an organization has established the foundations for improvement in both software and productivity.

Cleanroom Process Model

The philosophy behind the Cleanroom process model, or Cleanroom software engineering, is to minimize the number of defects by writing correct code the first time around [27]. The process model incorporates the concepts of incremental development, correctness verification, teamwork, and reliability certification. Software development involves one team that conducts both the developing and certification. For larger projects, an organization can use two separate teams, each restricted to a maximum of eight people. There are five stages defined in the Cleanroom model: specification, increment planning, design and verification, quality certification, and feedback [20].

The process begins with analyzing and verifying customer requirements. In the specification stage, the development team writes the functional specifications, which define the external behavior of the system, and the certification team produces the usage specifications, which define usage scenarios. During the planning stage, both teams create a plan to develop the system in increments based on the defined specifications. Development then cycles through the next three stages for each increment. The development team proceeds through design and correctness verification while the certification team produces test cases to parallel the development progress. The development team progresses to the quality certification stage by sending a partial system to a separate test team for both usage and statistical testing. In the Cleanroom model, the statistical usage testing process is specifically used to determine mean time to failure by testing software the way users intend to use it. During the feedback stage, errors are sent back to the development team for correction. Using the data from statistical testing, managers can determine how well development progress is proceeding. Development continues to cycle through these last three stages until the system is complete.

Extreme Programming

Extreme programming, also known as XP, is a methodology that is mostly a combination and reinforcement of selected software development practices [13]. It is a minimalist methodology that has a small set of rules and assumes that developers need little guidance. XP is designed for small teams of no more than ten developers.

XP stresses design simplicity: developers should design for current requirements, and not for potential future requirements. XP also encourages developers to refactor continuously during the development process. Refactoring is the redesigning of current software to improve its internal structure while maintaining its current external behavior. A unique practice of XP is pair programming where developers work in pairs such that one developer codes or tests while another developer reviews as the first developer works. Using XP, developers integrate frequently, once every few hours, to minimize the number of integration errors early on.

3 Related work

There have been a number of empirical studies conducted on software development in general, but there few focussed on the speed and flexibility of development.

Blackburn, Scudder, Van Wassenhove

This study surveyed software managers from the United States, Japan, and Western Europe [3]. American and Japanese managers completed a total of 49 surveys between the years 1992 and 1993, and 98 surveys were completed by western European participants through field interviews conducted in 1992. The researchers identified 11 factors that could contribute to the overall development time of a software project: use of prototypes, customer specifications, use of CASE (computer-aided software engineering) tools, parallel development, recoding, project team management, testing strategies, reuse of code, module size, communication between team members, and quality of software engineers. The study distinguishes the rate of change in development time from the rate of change in productivity: development time is concerned with the overall time while productivity is concerned with individual contribution such as lines of code (LOC) per man-month. This study uses LOC as a measure of productivity.

The study found that spending more time and effort spent on customer specifications improves both development and productivity speed. The results also indicate that prototyping, better software engineers, and less reworking or recoding contribute to faster development time. Their results show that smaller teams are faster, but there is little evidence outside this study that supports this result. They found that more time and effort spent on testing and integration actually have a negative effect on development speed, and better testing strategies and the newness of the project do not affect development time. Their overall results suggest that doing it right the first time is essential for reducing development time.

Their results support the view that larger teams reduce productivity during most stages of production. Productivity is also negatively correlated with both testing and project times. They could not confirm a definite relation between productivity and project size. They also found that the newness of the project bears no significance to productivity. Productive firms, however, have larger teams devoted to customer specifications. These results suggest that early planning and customer specifications are crucial to overall productivity.

Maxwell, Van Wassenhove, Dutta

Published in 1996, this study consists of 99 software development projects from the European Space Agency database, which began data compilation in 1988 [22]. Only projects developing space, military, and industrial applications were included in this study. The purpose of this study was to determine the metrics to measure productivity and the factors that influence productivity.

Productivity can be measured in different ways. A simple way to measure productivity is dividing source lines of code (SLOC) by man-months of effort. A more complex measure of productivity is process productivity, which was developed by Putnam and Myers. The calculation of process productivity takes developer skill as a function of system size, as well as SLOC and effort into account. Another method is by measuring the number of tokens per man-month. The number of tokens is determined by dividing SLOC by functionality. Productivity can also be measured by function points, which are based on user functionalities such as input and output features. Measuring productivity by feature points is an extension of function point measurement that includes measuring the algorithms in the software.

The study found that productivity increases when the projects had fewer reliability requirements, low storage constraints, fewer execution time constraints, small project teams, shorter durations, and when the projects used tools and modern programming practices. Contrary to previous findings, the results show that productivity increases with increasing system size. For this study, the quality of the applications could not be evaluated, and management factors were not considered.

Cusumano and Kemerer

Published in 1990, the main purpose of this study was to compare the performances of software projects between the United States and Japan to determine if the Japanese software industry is comparable to that of the United States [9]. A total of 40 projects from both countries were analyzed in this study.

The results show that both countries are similar in the following areas: types and sizes of products developed, development tools utilized, and programming languages and hardware platforms used for development. The work experiences of the developers were also comparable.

The study reveals that the Japanese teams concentrated more time on product design while the American teams spent more time on coding. However, both countries exhibited similar levels of productivity and defects.

Upadhyayula

Completed in 2001, this study analyzed 26 software development projects at Hewlett-Packard and Agilent [30]. The purpose of the study was to determine the software engineering practices that promote development flexibility.

The results indicate that projects should obtain and incorporate customer feedback during the early stages of development. In order to incorporate customer feedback more easily, project teams should begin with the design and implementation of high-level architecture instead of with detailed design. The study found that implementing a high percentage of the final product functionality in the first beta is a good way to minimize functionality changes later in the project. The results also show that design reviews improve product quality. If the quality assurance and development teams work closely together from the beginning of the project, the overall testing time is reduced.

4 Research Methodology

Survey Development

A survey was the instrument used to study and measure software engineering practices and how they are related to project performance. The survey in this study was based on a survey developed for a previous study, which Sharma Upadhyayula completed in January 2001 under the guidance of Professor Michael Cusumano.

For his study, Upadhyayula developed a survey and deployed it specifically to software development groups in Hewlett-Packard and Agilent. A team, which included members from both the academic and professional communities, developed this original survey. The team consisted of the following members: Professor Michael Cusumano (MIT Sloan School of Management), Professor Chris Kemerer (Katz Graduate School of Business, University of Pittsburgh), Professor Alan MacCormack (Harvard Business School), Bill Crandall (Hewlett-Packard), Guy Cox (Hewlett-Packard), and Sharma Upadhyayula (MIT).

For this study, Professors Cusumano, Kemerer, and MacCormack, Mr. Crandall and the author revised the original survey from Upadhyayula's study by adding new questions and modifying some of the original questions.

The author also created an addendum survey, which consisted of the new and modified questions of the new survey, for distribution to the past participants of the Hewlett-Packard and Agilent study.

Data Collection

Jeffrey Schiller of MIT Information Systems then set up the newly updated survey online. When participants submitted completed surveys, the survey forms were transmitted using SSL for secure transmission because the survey asked for information that may be confidential to the company. The web-based survey was deployed to select companies through the team's professional contacts. Each contact was encouraged to distribute the survey to the heads of software projects within his company.

Mr. Crandall distributed the addendum survey to the past participants from the Hewlett Packard and Agilent study. The data from the previous study can therefore be updated and utilized for this study.

The author examined the collected data to check for inconsistencies. When clarifications were needed, the author contacted the participant who had submitted the data in question to correct any misunderstandings or errors.

The author used both Microsoft Excel version 9.0 and Data Desk version 6.1 for Microsoft Windows to analyze the data.

Variables

The purpose of this study is to identify practices that affect performance and how development techniques differ across countries. Three types of variables are needed for this study: context, process, and outcome. Context variables describe the circumstances of software projects. Process variables identify the development practices projects employ. Outcome variables measure project performance. The three variables affect each other as shown in Figure 4-1.

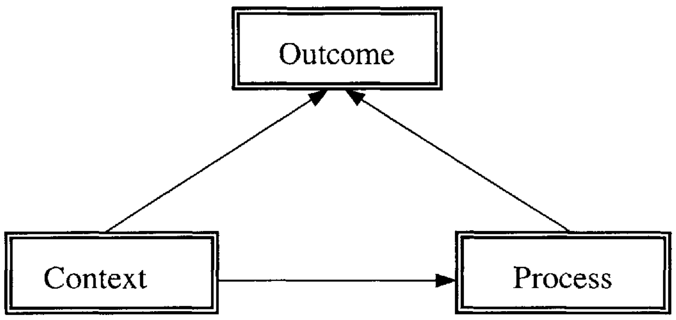

Figure 4-1. Relationships among context, process, and outcome variables. Context affects both the outcome and process variables and process affects the outcome.
Source: Upadhyayula, p. 17.

The survey covers all three variables. The following sections enumerate examples of the data collected for each variable.

Context Variables

- Type of software developed: systems, embedded, applications
- Level of reliability required: high, medium, low
- Target hardware platform: mainframe, workstation, PC, other
- Primary customer: individuals, enterprises, in-house use
- Project size: SLOC
- Project base: country of primary development

Process Variables

- Specification documents: architectural, functional, detailed
- Reviews: design and code
- Build frequency: daily, 2-3/week, weekly, bi-monthly, monthly or less
- Testing techniques: regression or integration, automated
- Sub-cycles: frequency, length of each sub-cycle
- Customer interaction: betas, prototypes

Outcome Variables

Productivity

Before productivity can be measured, the SLOC measurements need to be normalized. This can be done with the SPR (Software Productivity Research, Inc) Programming Languages Table version 7 [17], which lists common programming languages and their corresponding number of average source lines per function point. The author used this table to convert SLOC measurements to their C-equivalents. The author calculated C-equivalents with the following equation, where $\mathtt{SLOC}_L$ is the SLOC measurement in language L, f_L is the conversion factor for language L in terms of source lines per function point, and f_C is the conversion factor for C,

$$\frac{\mathtt{SLOC}_L}{f_L} * f_C$$

Equation 1. Conversion of a SLOC measurement to its C-equivalent.

Productivity is measured in SLOC per person-months. One person-month is the amount of work equivalent to one person working for one month. To calculate productivity, the author used the following equation,

$$\frac{(\text{\%code developed by this team}) * \text{SLOC}_{\text{Ceq}}}{\text{resources}_{\text{Total}} * 12}$$

Equation 2. Productivity in terms of SLOC per person-month.

where SLOC_{Ceq} is the source lines of code measurement converted to its C-equivalent and $\text{resources}_{\text{Total}}$ is the total number of resources in terms of person-years.

Bugginess

Bugginess refers to the technical quality of a product. The author chose to calculate bugginess, in terms of bugs per month per thousands of lines of code, with the following equation,

$$\frac{\text{bugs}}{\text{months-SLOC}_{\text{Ceq}}}$$

Equation 3. Bugginess in terms of bugs per month per thousands of lines of code.

where bugs is the number of bugs reported in the first 12 months after the product launch, months is the number of months of bug data available, and SLOC_{Ceq} is the C-equivalent of lines of code.

Timeliness

Timeliness measures how well a project team follows the original development schedule. The author measured this variable in terms of schedule estimation error, which is calculated by using the following equation:

$$\frac{(\text{actual duration} - \text{estimated duration})}{\text{estimated duration}}$$

Equation 4. Schedule estimation error.

Budget Overrun

This variable measures how much a project team outspends its budget, which is calculated by the following equation:

$$\frac{\texttt{(actual expenditure - budget)}}{\texttt{budget}}$$

Equation 5. Budget error.

Product quality can be measured either objectively or subjectively. An objective measurement is an evaluation of the product's technical quality in terms of errors and failures. A subjective measurement is an assessment of usability quality in terms of customer or user satisfaction.

Variable Choice: Software Size

Software size is usually measured in two different ways. The easier approach is to count the number of source lines of code, or SLOC. This simply entails counting the lines of code that are neither blanks nor comments. The main drawback is that language and implementation style affect this measurement outcome. Some programming languages require more lines of code to implement a function than others. For example, object-oriented languages usually require fewer lines of code than C. In terms of implementation, this method rewards quantity and punishes quality. For example, one developer may write a poorly designed piece of code which consists of more lines of code than necessary, and he will seem productive according to this type of measurement.

A popular method of measuring productivity is by function points. Function points are determined from the perspective of an end-user. Although the outcome of this method is not affected by language or implementation differences, the process of counting function points is a labor-intensive and time-consuming activity. Besides tallying functionalities, it involves calculations that take project complexity into account. Because it is a sensitive process, companies can hire certified experts to count function points.

Despite the disadvantages of measurement by source lines of code, the survey participants are more likely to have the information regarding SLOC available than information about function points. Therefore, the decision was to measure software size by source lines of code for this project.

5 General Project Characteristics

A total of 60 surveys were collected during the period between December 2001 and April 2002. Two surveys were excluded from the sample due to insufficient answers. One other project was omitted because of its small size-a one-person development team. Thus, 57 new projects are included in this study in addition to 30 projects from the 2000 HP-Agilent study. The final sample therefore contains 87 projects.

This section gives a general overview of the projects' characteristics.

Region

The scope of the projects is international, covering mainly Asia and Northern America. Table 5-1 below shows the regions where the projects were based. Excluding Japan, the region of Asia includes China, Hong Kong, India, and Taiwan. The two projects from Europe are based in France and Switzerland. There is one project based in Bahrain.

Region	Count
USA and Canada	40
Japan	27
Asia-excluding Japan	17
Europe	2
Other	1
Total	87

Table 5-1. Locations of project bases.

Type of Software

More than half of the sample developed software applications, which include both include custom and general-purpose applications. In the sample, there were more custom applications than general-purpose. The other types of projects include embedded software, systems software, and other. The "other" category includes projects that developed any combination of the previously mentioned three types. Table 5-2 below summarizes the types of software developed by the reported projects.

Type	Count
Applications	51
Embedded	11
Systems	25
Total	87

Table 5-2. Summary of software type.

Customers

Table 5-3 below shows the rundown of the projects' targeted customers. There are roughly three times as many projects developed for external customers or external use than projects developed for internal purposes.

Customer	Count
External	64
Internal	23
Total	87

Table 5-3. Customers or use.

Programming Languages

Table 5-4 below shows a percentage breakdown of the programming languages used in the projects. Note that most projects used more than one programming language, the percentages do not sum to 100%.

Programming Language	Percentage
Assembly	3.4
C	24.1
C++	35.6
COBOL	4.6
Java	25.3
Visual Basic	10.3
Web-content languages	13.8
Other	26.4

Table 5-4. Programming Languages. Web-content languages include ASP, HTML, Java Script, JSP, PHP, and VB Script. A total of 85 surveys were used in this summary.

From the projects that have been collected, object-oriented languages, i.e. C++ and Java, dominate as the primary development language of choice with C++ leading in popularity. C remains a popular development language.

Reviews

Although high percentages of the projects conducted design reviews and code reviews, more projects had the practice of design reviews than code reviews, as shown in Tables 5-5 and 5-6.

Design Reviews		Code Reviews	
Yes	77	Yes	62
No	10	No	25
Total	87	Total	87

Table 5-5. Summary of design and code reviews.

		Design Reviews	
		Yes	No
Code Reviews	Yes	58	4
	No	19	6

Table 5-6. Cross summary of design and code reviews. A total of 87 surveys were used in this summary.

Approximately 95% of the projects performed one or both types of reviews, with nearly 70% of the projects surveyed conducting both design and code reviews.

Sub-cycles

A sub-cycle usually consists of all the phases in the waterfall method: design, implementation, build, test, integration, and release. A project that has sub-cycles means that its development occurred through iterations whereas a project with no sub-cycles followed the more traditional waterfall approach. As shown in Table 5-7 below, the majority of the projects used sub-cycles during their development phase. The number of sub-cycles ranged from 2 to 36. The average number of sub-cycles per project is 3.8 with a 4.9 standard deviation and a median of 3.

Sub-cycles Used	Count
Yes	55
No	32
Total	87

Table 5-7. Summary of sub-cycles usage.

Project Size

The range of the project sizes in the sample reaches the millions regardless of normalization. Table 5-8 below summarizes characteristics of the project sizes including the statistics for both the overall size and the new code the current project team developed. Project size is measured in terms of source lines of code, and the sizes were normalized to their C-equivalents as described in section 4. Some projects did not provide enough information to normalize their SLOC measurements.

SLOC	#Projects	Min	Max	Average	Std Dev	Median
overall (normalized)	77	2376	97M	2.71M	11.5M	362264
overall	86	500	45M	1.40M	5.54M	160000
new (normalized)	77	2257	4.9M	545709	1.05M	134095
new	86	475	4.7M	247360	593162	59244

Table 5-8. Summary of project sizes. All measurements are in terms of source lines of code, except "#Projects".

Staff Size and Resources

Along with a large range in project sizes, there is a large range of staff size in this sample. Table 5-9 summarizes average staff size and resources of the development team and the whole project team. Staff averages are measured in numbers of people, and resources are measured in person-years.

	#Projects	Min	Max	Average	Std Dev	Median
average development	87	0.4	500	22	73	6
average total	87	1	790	40	109	13
resources development	87	0.1	940	34	126	6
resources total	87	0.2	1260	58	181	12

Table 5-9. Summary of average staff size and resources for development and entire project team.

Version

Most of the projects in the sample developed new products as opposed to developing the next version of an existing product. A project that includes more than 50% of the existing code from a previous version of the product is considered a product extension. Table 5-10 summarizes the sample below.

Version	Count
New product	54
Product extension	33
Total	87

Table 5-10. Summary of product version.

6 Summary of Hypotheses

Staff Experience

Staff experience is measured by the following percentages: staff with 0-5 years of development experience and staff with 6+ years of development experience. The higher the percentage of staff with 6+ years of experience, the more experienced the project team. The higher the percentage of staff with 0-5 years of experience, the less experienced the project team. The author hypothesizes that more experienced project teams will have the following properties:

- higher levels of productivity. In Boehm's COCOMO II model [7], the equation to estimate the number of person-months required to develop a project indicates that more experienced developers will need less time than less experienced developers to finish a project. This indicates that experienced developers are more productive overall.
- lower levels of bugginess. More experienced developers generally write more robust code which reduces the number of potential bugs in the code.
- lower levels of overspending budget. Since more experienced developers should be more productive, they are less likely to go over schedule, which decreases the chances of going over budget.
- lower schedule estimation error. More experienced teams should be able to better manage their schedule and work more productivity which should help the project finish more timely.

Specification documents

Formal documentation of specifications usually indicates thoroughly planned designs. A project that lacks a specification document has 0% completion of that document. Higher levels of completion of specifications before coding begins mean more time was put into design. There are three different specification documents: architectural, functional or requirements, and detailed design. The author hypothesizes that higher levels of specification completion before coding contribute the following:

- higher levels of productivity. Since "design serves as the foundation [for project activities]...effective design is essential to achieving maximum development speed" [23, p. 63]. Therefore a more well thought out design will increase productivity.
- lower levels of bugginess. A poor design that cannot be easily extended or modified may need to be redesigned later in the development process. Redesigning, which can affect many code modules, is prone to bugs. Thus, poor designs ultimately contribute to more bugs.
- lower schedule estimation error. If good design increase productivity, then it should in turn increase the likelihood that the project will finish more timely.
- lower levels of overspending budget. If a good design decreases the amount of time a project to goes over deadline, then the project will be less likely to overspend their budget. The longer a project lasts, the more money is spent on the project.
- higher customer satisfaction perception rating. With a good design and fewer bugs, customer satisfaction should be high.

Feedback

There are two types of feedback: market and technical. Prototypes and betas provide market feedback and integrations offer technical feedback. Feedback timing is measured by the percentage of final functionality present in the first releases or the first integration. Projects that lack any one of these project events have 0% of the final functionality in the missing event. Lower percentages mean the project teams received earlier feedback. The author hypothesizes that feedback will affect the following outcome variables:

- productivity. Early market feedback will increase levels of productivity because changes to the requirements or functionalities are easier to incorporate earlier in the project than later. Early technical feedback will increase levels of overall productivity because fixing defects early requires less time than fixing them later.
- bugginess. Early market feedback minimizes major rework because any large functionality changes that would want to make will more likely occur early in the process rather than later. Avoiding major rework later in the project, in turn,

decreases the number of errors. Early technical feedback detects errors early in the project so that the errors can be corrected early. It is easier to find and fix bugs earlier than later in the projects so that early correction reduces the overall level of bugginess.

- timeliness. Early market feedback minimizes rework late in the project, which helps a project finish more timely. Early technical feedback decreases schedule estimation error because detecting and fixing defects early reduces the amount of time spent on the project [23].
- level of overspending budget. Since early market feedback minimizes rework, a time-consuming process, late in the project the time spent on a project reduces which also reduces levels of expenditure. As a result, the level of overspending decreases. Early technical feedback, which exposes errors early, reduces the level of overspending budget because fixing defects later in the project is more expensive [23].
- customer satisfaction perception rating. Early market feedback demonstrates a project's progress and allows the customer to participate in the project. These two consequences increase customer satisfaction with the project. Early technical feedback does not have an effect on customer satisfaction.

Design and code reviews

Design reviews allow project teams to evaluate and improve upon their original designs. Code reviews can help developers correct potential mistakes in the code or improve the readability of their code. Design review is measured by whether the project team conducted design reviews and the number of design reviews done. Similarly, code review is measured by whether the project team conducted code reviews and percentage of final code that was reviewed. The author hypothesizes that the practice of these two reviews results in the following:

- higher levels of productivity. A design error left undetected until testing can take up to 10 times as long to fix as it would if it were detected at design time [11]. Conducting design reviews can uncover design errors before coding begins, which increases overall productivity because rework is minimized. Although code reviews take away from coding, readable and relatively error-free code speeds up development. This contributes to an increase in overall productivity.

- lower levels of bugginess. Technical reviews can find a large percentage of bugs. For example, informal reviews can detect 30%-70% of the errors in a system [24, 4, 32].
- lower schedule estimation error. Technical reviews uncover errors earlier in the project which helps a project finish more timely because errors detected later in the project take longer to fix than errors detected earlier [23].

Sub-cycles

Projects can develop incrementally by using sub-cycles. This study will analyze the following aspects regarding sub-cycles: the practice of sub-cycles, the number of sub-cycles and the length of sub-cycles. The hypotheses are that sub-cycles will contribute to the following outcome variables:

- lower levels of productivity. Development in sub-cycles decreases productivity because constant incorporation of customer feedback which involves removing and adding functionalities. The more requirement specifications change, the lower the level of productivity [31].
- lower levels of bugginess. With each sub-cycle, new errors are found and fixed which reduces the number of bugs of the system.
- lower schedule estimation error. Developing in sub-cycles allows a project team to constantly re-evaluate and adjust their schedule, which lowers the percentage of schedule estimation error. As a project progresses, estimations become more accurate [6].
- higher customer satisfaction perception rating. Progress is visible with incremental development, which increases customer satisfaction.

Regression or Integration Tests

A regression or integration test is used to detect if any new bugs were introduced after previously tested code was modified. This test is more than just a simple compile and link test. Running regression tests after each time the code base is modified is hypothesized to affect the following results:

- higher levels of productivity. By detecting and fixing bugs early with this practice, the overall productivity of the project increases because uncovering and correcting bugs early in the project consumes less time than if the bugs were found later in the project.
- lower levels of bugginess. Running a regression test after each code change helps pinpoint errors more easily. Therefore, the uncovered errors are easier to fix which reduces the overall level of bugginess.
- lower schedule estimation error. According to Capers Jones [16], one main contributor to schedule overrun is poor quality. Since this practice should reduce the bugginess level of a project, the schedule estimation error should decrease.

Testing Techniques

Developer participation in writing test cases and automated testing are two testing techniques hypothesized to have the following effects:

- higher levels of productivity. Developer participation in testing and automated tests contribute to testing efficiency, which increases overall productivity.
- lower levels of bugginess. Developers know what to test for because they know the code so their participation in testing helps uncover more errors, which lowers the levels of bugginess. Automated tests also help detect errors, which reduces the number of bugs in a system.

7 Data Analysis

7.1 Preliminary Data Analysis: Productivity

In this study, productivity is calculated in terms of SLOC per person-month using the equation in Section 4. A summary of productivity is provided in the Table 7-1 below.

Productivity	#Projects	Min	Max	Average	Std Dev	Median
With outliers	77	16.7715	101230	3949.32	13292.7	895.807
Without outliers	74	16.7715	8369.39	1605.69	2011.45	828.973

Table 7-1. Summary of productivity with and without outliers.

Three outliers were removed from the productivity analysis. Since there is a large range of productivity levels, an ANOVA analysis was run to see if the type of software (applications, embedded, and systems) developed affects productivity level.

```
Analysis of Variance For              Productivity
No Selector
87 total cases of which 13 are missing

Source    df    Sums of Squares       Mean Square    F-ratio    Prob
Const      1    190.789e6             190.789e6      49.368     ≤ 0.0001
aST        2     20.9628e6             10.4814e6      2.7121      0.0733
Error     71    274.391e6               3.86466e6
Total     73    295.353e6
```

Table 7-2. ANOVA of productivity and all three software types.

With a F-ratio of 2.7121 and a probability greater than 0.05, productivity levels differ depending on the software type. A boxplot reveals the difference, as shown below in Figure 7-1.

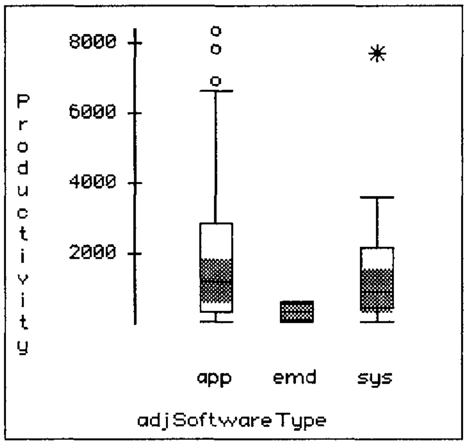

Figure 7-1. Boxplot of productivity and software types. Extreme outliers are denoted by "*" and a "o" indicates an outlier.

Applications and systems have similar mean values for productivity so an ANOVA was run on applications and systems, as shown below in Table 7-3.

```
Analysis of Variance For                         Productivity
No Selector
87 total cases of which 13 are missing

Source    df    Sums of Squares     Mean Square     F-ratio     Prob
Const      1    190.789e6           190.789e6       49.71       ≤ 0.0001
twT        1     19.0136e6           19.0136e6       4.954        0.0292
Error     72    276.34e6              3.83805e6
Total     73    295.353e6
```

Table 7-3. ANOVA of productivity and the software types of applications and systems.

Since there is little difference between the productivity levels of applications and systems, the two software types are combined into one category. The resulting boxplot is shown below in Figure 7-2.

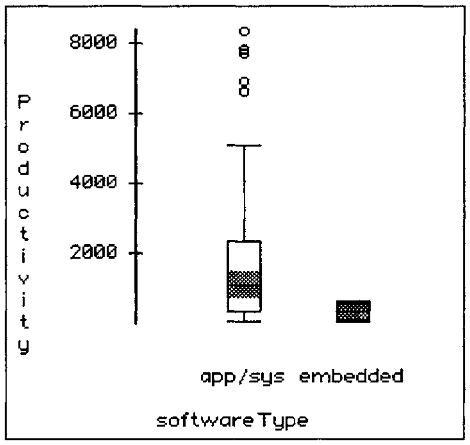

Figure 7-2. Boxplot of productivity and the software types of application/systems and embedded.

As a result of the above analysis, the sample for productivity analysis was partitioned into two samples: application/systems and embedded. Only the productivity analysis for application/systems was performed due to the small number of embedded projects in the overall sample. The five outliers from the applications/systems sample were removed.

7.2 Data Analysis: Productivity of Application/Systems

This section contains the productivity analyses for the combined samples of application and systems software. A summary of the productivity characteristics for the applications and systems sample is shown below in Table 7-4.

Productivity	#Projects	Min	Max	Average	Std Dev	Median
Applications/Systems	59	16.7715	5093.83	1321.81	1292.54	900.364

Table 7-4. Summary of productivity for applications and systems.

Hypothesis 1: More experienced project teams will be more productive than less experienced teams.

	Productivity	*Statistical Significance*
% of Developers with 0-5 years experience	-0.025	0.8666
% of Developers with 6+ years experience	0.025	0.8666

Table 7-5. Pearson Product-Moment Correlation of productivity and percentages of developers with different levels of experience.

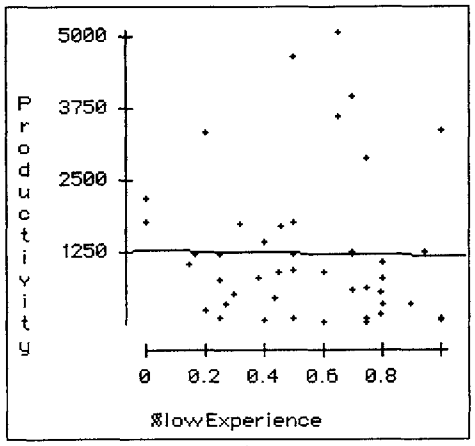

Figure 7-3a.

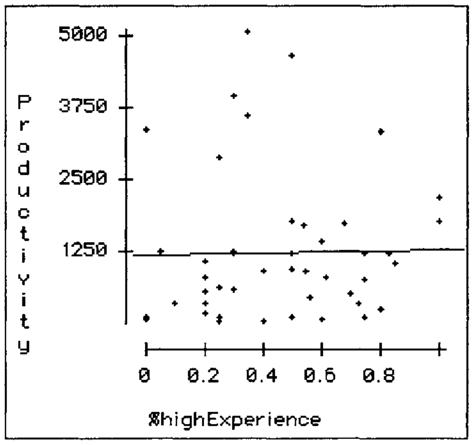

Figure 7-3b.

Figures 7-3: Scatterplots of productivity vs. percentage of developers with different levels of experience. A total of 56 projects were in this sample.

a. Percentage of developers with 0-5 years of experience
b. Percentage of developers with 6+ years of experience

There is no statistically significant correlation between developers' experience and productivity as shown in Table 7-5 and Figures 7-3a and 7-3b. This outcome suggests two possibilities: 1) a developer with many years of experience does not necessarily outperform, in terms of productivity, a developer with little experience and 2) a more experienced developer writes more concise code than a less experienced developer; therefore he seems to be less productive. A function point analysis would be helpful in understanding this relationship.

Hypothesis 2: Project teams that complete more of their specification documents before coding are more productive.

	Productivity	*Statistical Significance*
% of Architectural specifications finished	0.219	0.0983
% of Functional specifications finished	0.100	0.4565
% of Detailed specifications finished	0.123	0.3582

Table 7-6. Pearson Product-Moment Correlation of productivity and percentage of specifications completed before coding.

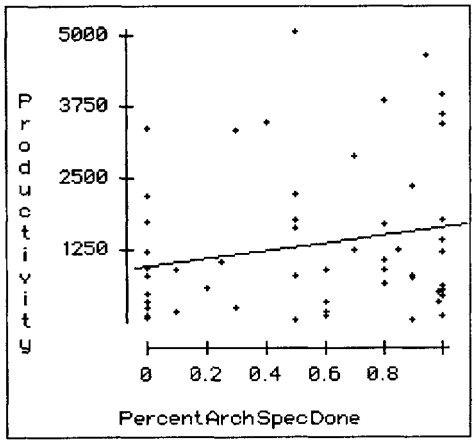

Figure 7-4a.

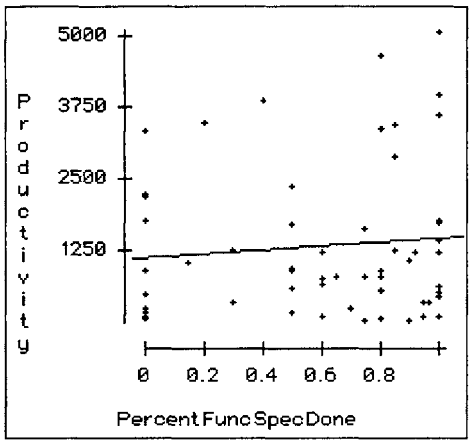

Figure 7-4b.

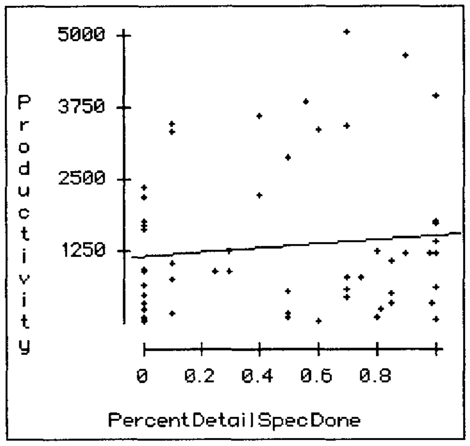

Figure 7-4c.

Figures 7-4. Scatterplots of productivity vs. percentage of specification documents completion before coding. There were 58 projects in the sample.

a. Architectural specification
b. Functional (or requirements) specification
c. Detailed specification

The correlations between productivity and the percentages of specification documentation completed before coding are positive but none are statistically significant. The results indicate that the more complete the architectural specification document is before coding begins, the higher the productivity levels. With most of the design laid out, developers should be able to focus more on coding and to code more efficiently. The degrees of completeness of functional and detailed specifications have little impact on productivity levels.

Hypothesis 3: Projects that receive early technical and market feedback have higher levels of productivity.

	Productivity	*Statistical Significance*
% of Final functionality in first prototype	-0.140	0.3022
% of Final functionality in first integration	-0.051	0.7076
% of Final functionality in first beta	-0.083	0.5442

Table 7-7. Pearson Product-Moment Correlation of productivity and percentage of the final functionality present in early project events.

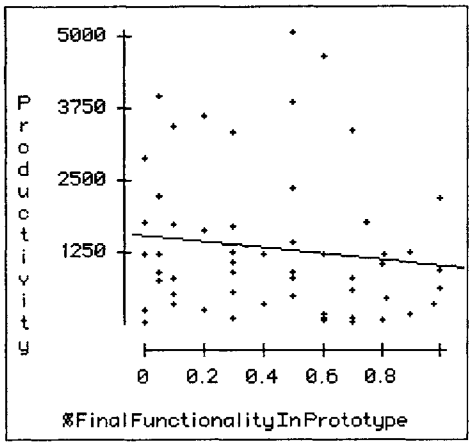

Figure 7-5a.

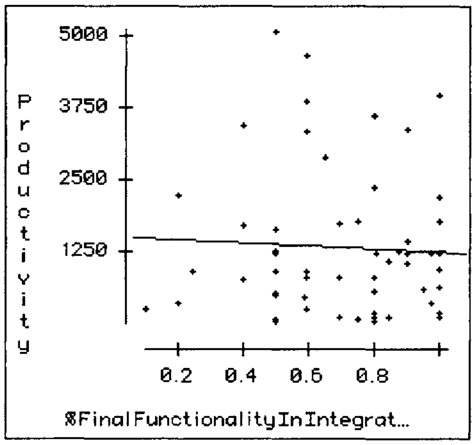

Figure 7-5b.

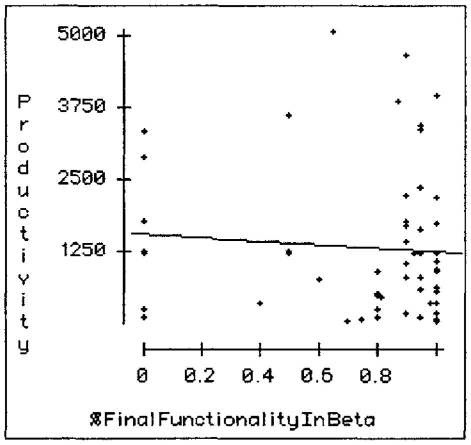

Figure 7-5c.

Figures 7-5. Scatterplots of productivity vs. percentage of the final functionality present in early project events. There were 56 projects in the sample.

a. First prototype
b. First integration
c. First beta

There are no statistically significant correlations between productivity and feedback. However, there is a higher negative correlation between productivity and market feedback (% final functionality in first beta and % final functionality in first prototype) than the correlation between productivity and technical feedback. Project teams that receive early market feedback are more likely to incorporate the feedback, which involves modifying, adding, and deleting functionality. This slows down overall productivity because the team needs to rework their original implementation.

Hypothesis 4: The practice of design reviews makes a project team more productive.

	Productivity	*Statistical Significance*
Conducted design reviews (1=Yes, 0=No)	0.117	0.3856
# Design reviews	-0.146	0.2793

Table 7-8. Pearson Product-Moment Correlation of productivity and design reviews.

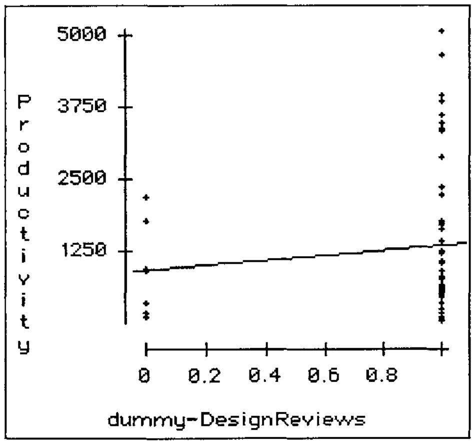

Figure 7-6a.

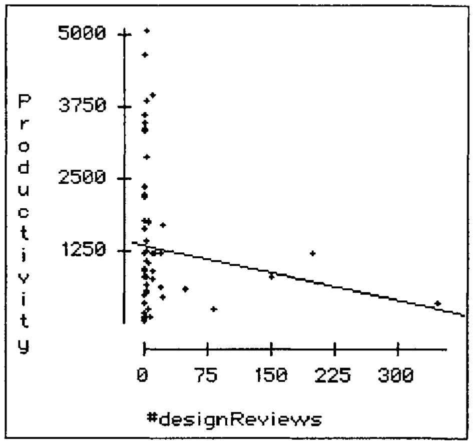

Figure 7-6b.

Figures 7-6. Scatterplots of productivity vs. design reviews. There were 68 projects in the sample.

a. Dummy variable of design reviews

b. Number of design reviews conducted

A dummy variable was used to denote the practice of design reviews such that the variable was set to 1 if the team conducted design reviews and 0 if the team did not. The correlation between productivity and the practice of design reviews is positive but not statistically significant. This suggests that design reviews contribute to overall project efficiency. Design reviews help to uncover design errors and to improve the overall project design. So by doing design reviews, the overall design of the project should not require any major modifications later in the development process.

Although there is a negative correlation between productivity and the number of design reviews conducted, the correlation is not statistically significant. This suggests that conducting more design reviews slows down developers' productivity. However, it must be noted that larger projects are more likely to conduct more design reviews than smaller projects, and larger projects are traditionally less productive than smaller projects. Therefore, the size of the project may be affecting the relationship between the number of design reviews and productivity.

<u>Hypothesis 5</u>: The practice of code reviews is associated with higher levels of productivity.

	Productivity	*Statistical Significance*
Conducted code reviews (1=Yes, 0=No)	-0.005	0.9699
% Code reviewed	0.004	0.9750

Table 7-9. Pearson Product-Moment Correlation of productivity and code reviews.

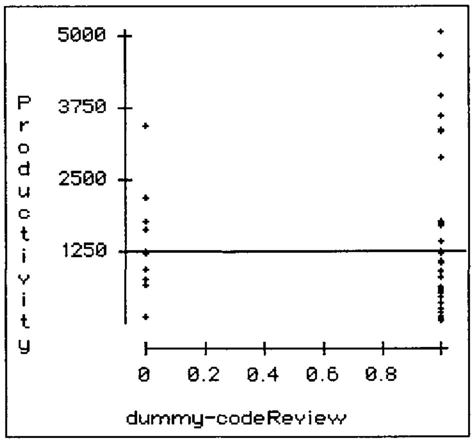

Figure 7-7a.

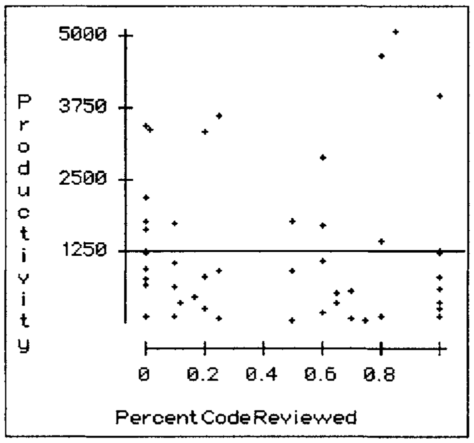

Figure 7-7b.

Figures 7-7. Scatter plots of productivity vs. code reviews. There were 51 projects in the sample.
a. Dummy variable of code reviews
b. Percentage of code reviewed

Like design reviews, a dummy variable was used to denote whether a project team conducted code reviews such that the variable was set to 1 if the project team did code reviews and 0 otherwise. There is essentially no correlation between productivity and the practice of code reviews. A possible explanation is the fact that code reviews help streamline code such that the original code is rewritten to be more concise. As a result, the productivity levels seem lower. Further investigation of the number of product functionality would help in understanding this relationship.

Hypothesis 6: The use of sub-cycles will result in lower levels of productivity.

	Productivity	*Statistical Significance*
Used sub-cycles (1=Yes, 0=No)	0.065	0.6241
# Sub-cycles	0.183	0.1664
Sub-cycle length (% of project duration)	0.019	0.8862

Table 7-10. Pearson Product-Moment Correlation of productivity and sub-cycles.

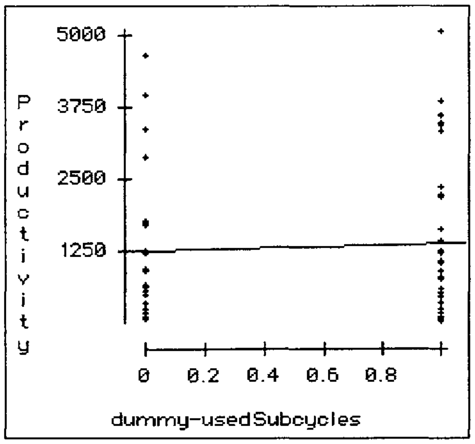

Figure 7-8a.

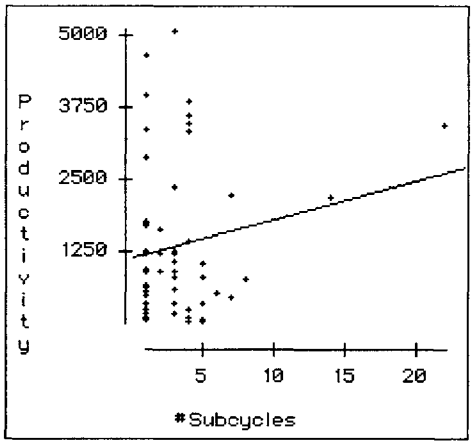

Figure 7-8b.

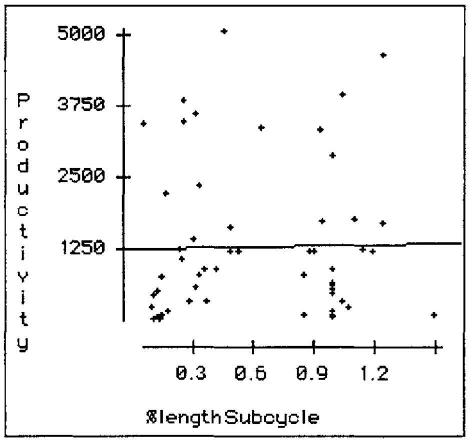

Figure 7-8c.

Figures 7-8. Scatter plots of productivity vs. sub-cycles. There were 55 projects in the sample.

a. Dummy variable of sub-cycle use
b. Number of sub-cycles
c. Sub-cycle length as a percentage of the project duration

Although there are no statistically significant correlations between productivity and sub-cycles, all three correlations are positive. There is a stronger relation between productivity and the number of sub-cycles than the relations between productivity and the other two factors regarding sub-cycles. This suggests that using more sub-cycles increases productivity level. Iterative development allows the designs and requirements to be reassessed and improved if necessary, which means that the likelihood of unnecessary implementation is reduced. This increases overall productivity.

Hypothesis 7: A project team that runs regression tests after each modification to the code base will have higher levels of productivity.

	Productivity	*Statistical Significance*
Ran regression test (1=Yes, 0=No)	0.251	0.0325

Table 7-11. Pearson Product-Moment Correlation of productivity and the use of regression test after code base modifications.

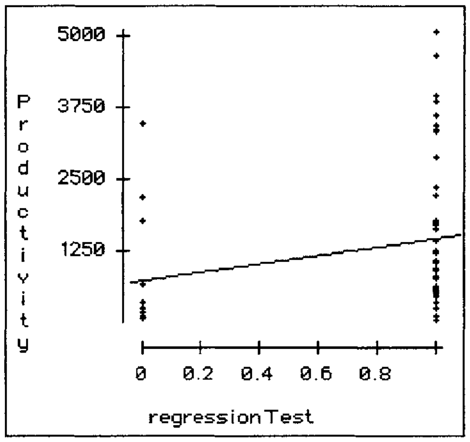

Figure 7-9. Scatterplot of productivity and the practice of running regression tests after code base modifications. There were 54 projects in this sample.

The correlation between productivity and the dummy variable for running regression tests is statistically significant at the 0.05 level. Testing for bugs after each code base modification prevents bugs from accumulating as development progresses and ensures that developers do not introduce new bugs to the existing code. It also makes it easier to pinpoint the source of the error. As a result, developers can spend more time on coding and less time on finding and fixing bugs.

Hypothesis 8: More developer participation in testing contributes to higher levels of productivity.

	Productivity	*Statistical Significance*
Developers write test cases (1=Yes, 0=No)	0.047	0.7533
% of Code tested by developers	0.161	0.2737

Table 7-12. Pearson Product-Moment Correlation of productivity and the developer participation in testing.

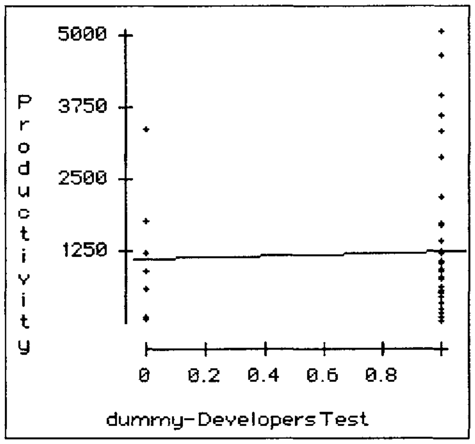

Figure 7-10a.

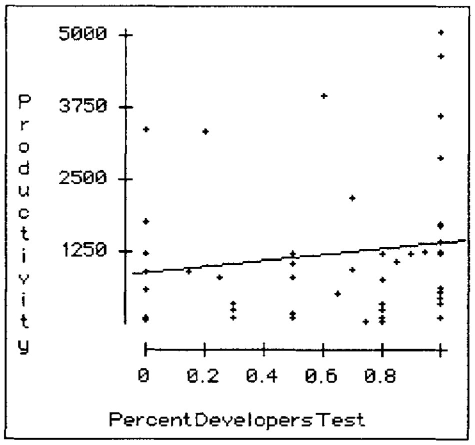

Figure 7-10b.

Figures 7-10. Scatterplots of productivity and developer participation in testing. There were 48 projects in this sample.

a. Dummy variable for developer participation in testing
b. Percentage of code tested by developers

The correlation between productivity and developer participation in testing is not statistically significant but there is a slight positive relation. The reason could be that developers know the code better so that testing is overall more efficient, and therefore overall productivity levels increase.

Hypothesis 9: Levels of productivity increase as higher percentages of tests are automated.

	Productivity	*Statistical Significance*
% of Automated tests	-0.228	0.0858

Table 7-13. Pearson Product-Moment Correlation of productivity and the percentage of automated tests.

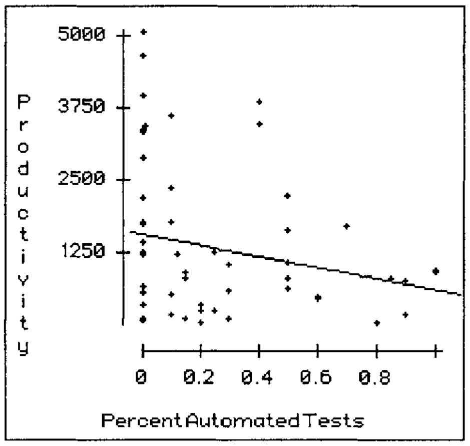

Figure 7-11. Scatterplot of productivity and the percentage of automated tests. There were 58 projects in this sample.

There is a negative correlation between productivity and the percentage of automated tests that is not statistically significant at the 0.05 level. A possible reason is that the effort put into writing the automated tests takes away from testing which could reduce overall productivity levels.

Sensitivity Analysis

A sensitivity analysis was performed to determine whether a particular company (HP-Agilent) had a significant impact on the overall productivity analysis. The results are shown below in Table 7-14.

	Original	HP-Agilent excluded	
	Significance Level	**Correlation**	**Significance Level**
% of Automated tests	≤ 0.10	-0.344	≤ 0.05
Ran regression test (1=Yes, 0=No)	≤ 0.05	0.197	> 0.10

Table 7-14. Summary of analysis with the exclusion of projects from HP-Agilent.

An explanation of the differences between the original productivity analysis and the analysis with the exclusion of HP-Agilent projects could be that automated tests are less productive and that running regression tests after code base changes increases productivity at the company relative to the other projects.

7.3 Data Analysis: Bugginess

Bugginess is measured in terms of bugs per month per thousand lines of code, calculated with Equation 3. For the data analysis, two outliers were removed from the sample. Table 7-15 provides a summary of bugginess levels of the samples, with and without outliers.

Bugginess	#Projects	Min	Max	Average	Std Dev	Median
With outliers	60	0	1111.11	93.6315	228.677	7.9206
Without outliers	58	0	701.754	60.4617	142.946	7.57558

Table 7-15. Summary of bugginess, with and without outliers.

Hypothesis 1: Project teams with higher percentages of experienced developers have lower levels of bugginess.

	Bugginess	*Statistical Significance*
% of Developers with 0-5 years experience	0.167	0.3039
% of Developers with 6+ years experience	-0.167	0.3039

Table 7-16. Pearson Product-Moment Correlation of bugginess and developers' experience.

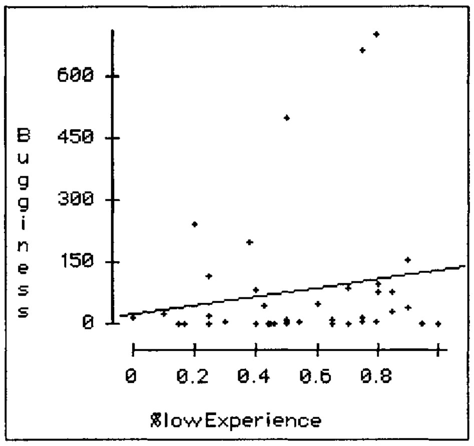

Figure 7-12a.

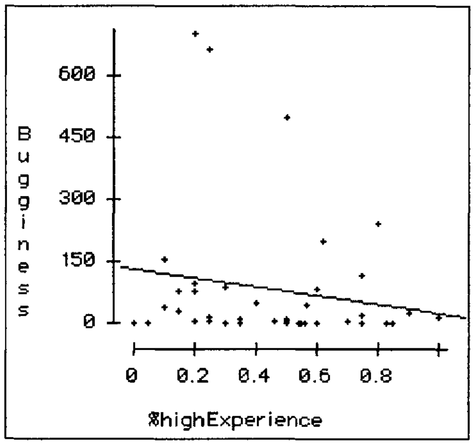

Figure 7-12b.

Figures 7-12. Scatterplots of bugginess vs. percentage of developers with different levels of experience. A total of 40 projects were in this sample.

c. Percentage of developers with 0-5 years of experience.
d. Percentage of developers with 6+ years of experience

The correlations between bugginess and the experience levels of the project teams are not statistically significant. The positive correlation between bugginess and project teams with low experience and the negative correlation between bugginess and more experienced project teams indicate that more experienced developers write more solid code. This is intuitive because a developer's coding ability improves over time by learning from past mistakes.

<u>Hypothesis 2</u>: Project teams that complete a higher percentage of specification documents before coding have lower levels of bugginess.

	Bugginess	*Statistical Significance*
% of Architectural specification finished	-0.135	0.3162
% of Functional specification finished	-0.108	0.4252
% of Detailed Specification Finished	-0.155	0.2492

Table 7-17. Pearson Product-Moment Correlation of bugginess and percentage of specifications completed before coding.

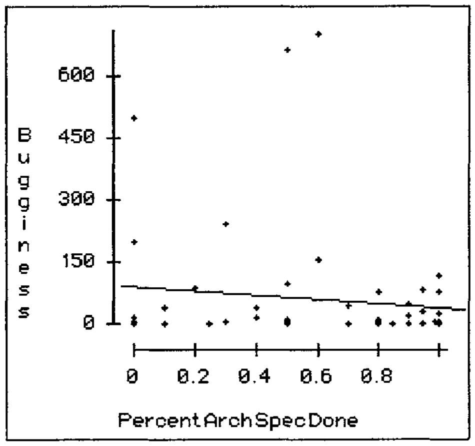

Figure 7-13a.

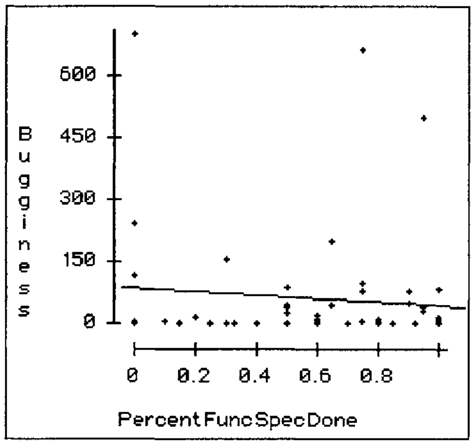

Figure 7-13b.

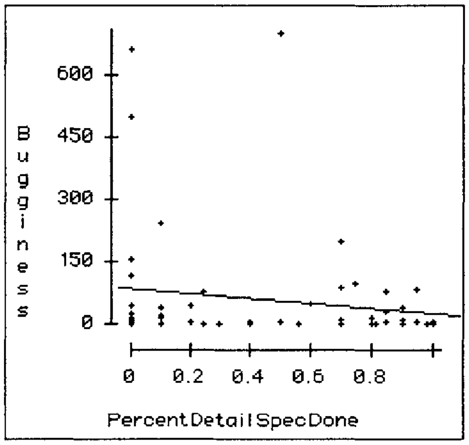

Figure 7-13c.

Figures 7-13. Scatterplots of bugginess vs. percentage of specification documents completion before coding. There were 57 projects in the sample.

a. Architectural specification
b. Functional (or requirements) specification
c. Detailed specification

The correlations between bugginess and completeness of the specification documents before coding are statistically insignificant. However, all three correlations are negative. When more time is spent on design, more of the design documents are completed before coding begins. This indicates that more time spent on planning the design before writing code leads to more robust code. Projects that do not spend much time on design may need to redesign later during development, which usually contributes more bugs to the project.

Hypothesis 3: Projects that receive early technical and market feedback have lower levels of bugginess.

	Bugginess	*Statistical Significance*
% of Final functionality in first prototype	0.166	0.2292
% of Final functionality in first integration	0.064	0.6468
% of Final functionality in first beta	-0.174	0.2094

Table 7-18. Pearson Product-Moment Correlation of bugginess and percentage of the final functionality present in early project events.

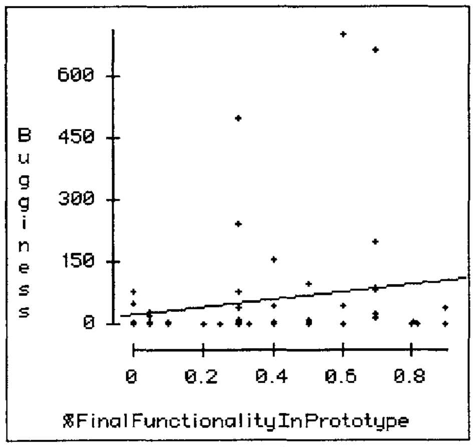

Figure 7-14a.

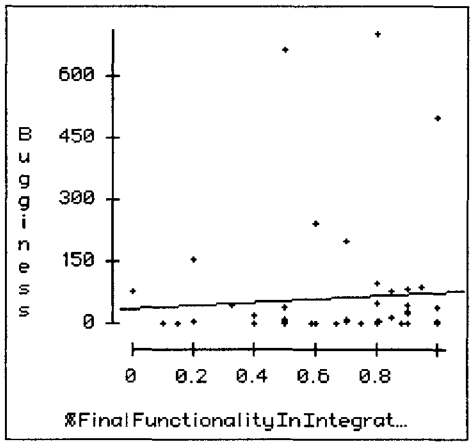

Figure 7-14b.

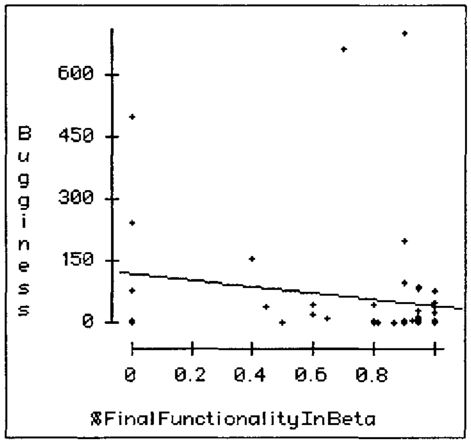

Figure 7-14c.

Figures 7-14. Scatterplots of bugginess vs. percentage of the final functionality present in early project events. There were 54 projects in the sample.

a. First prototype
b. First integration
c. First beta

The correlations between bugginess and the percentage of final functionality present in early project events are not statistically significant. No general conclusions about market and technical feedback regarding bugginess can be drawn. The difference between prototype and beta feedback could either be due to the fact that prototypes are usually released once and betas are released multiple times or the fact that prototypes are usually released before betas. If a project team releases its prototype late in the project, there is less time to incorporate feedback, which usually results in more bugs. Projects that release later betas may be less likely to incorporate feedback. Less incorporation means less rework and therefore fewer potential bugs.

Hypothesis 4: The practice of design reviews reduces bugginess levels.

	Bugginess	*Statistical Significance*
Conducted design reviews (1=Yes, 0=No)	0.029	0.8388
# Design reviews	-0.044	0.7529

Table 7-19. Pearson Product-Moment Correlation of bugginess and design reviews.

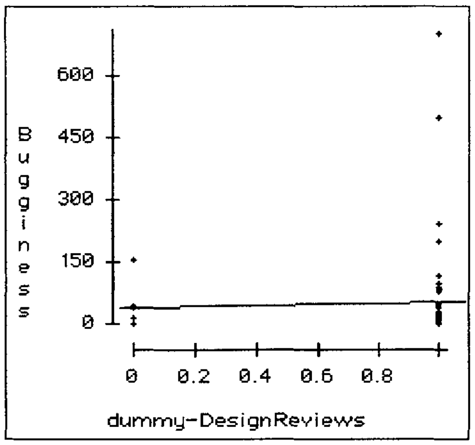

Figure 7-15a.

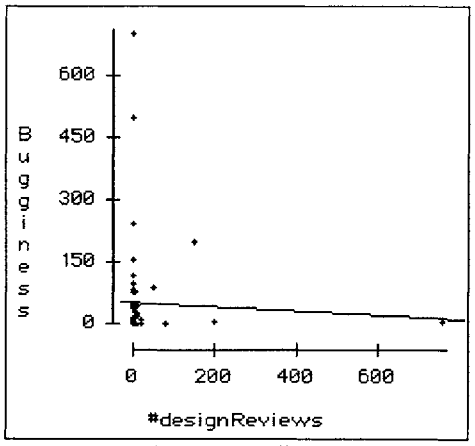

Figure 7-15b.

Figures 7-15. Scatterplots of bugginess vs. design reviews. There were 53 projects in the sample.
a. Dummy variable of design reviews
b. Number of design reviews conducted

A dummy variable was used for the practice of design reviews such that the variable was set to 1 if the team conducted design reviews and 0 if the team did not. The correlations between bugginess and design reviews are not statistically significant. Further analysis on the data cluster at the lower left-hand corner of Figure 7-15b did not yield statistically significant correlations.

Hypothesis 5: The practice of code reviews is associated with lower bugginess levels.

	Bugginess	*Statistical Significance*
Conducted code reviews (1=Yes, 0=No)	0.225	0.1249
% Code reviewed	0.204	0.1641

Table 7-20. Pearson Product-Moment Correlation of bugginess and code reviews.

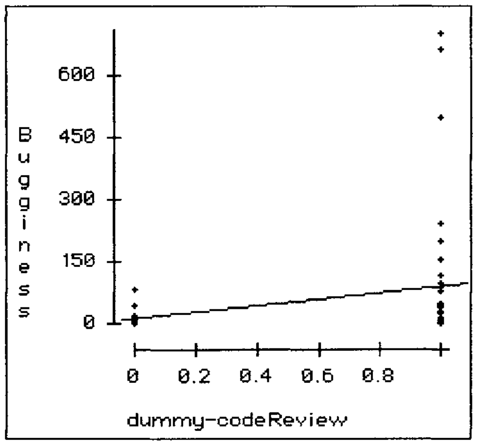

Figure 7-16a.

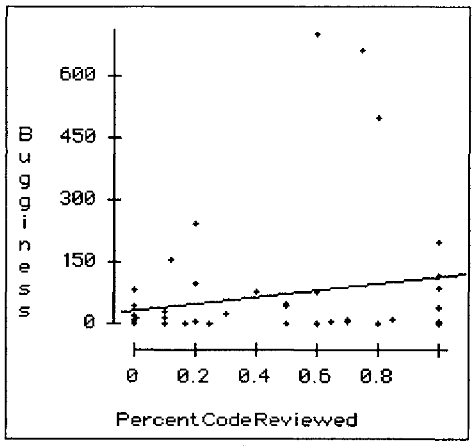

Figure 7-16b.

Figures 7-16. Scatterplots of bugginess vs. code reviews. There were 48 projects in the sample.
c. Dummy variable of code reviews
d. Percentage of code reviewed

A dummy variable was used to denote whether a project team conducted code reviews where the variable is set to 1 if the team had code reviews and 0 otherwise. There is a positive correlation between bugginess and the practice of code reviews. This may be explained by the calculation of bugginess (Equation 3) where fewer lines of code contribute to a higher level of bugginess. Since many code reviews are conducted to ensure that code is written concisely, the project teams that engaged in code reviews are more likely to have higher bugginess levels.

Hypothesis 6: The use of sub-cycles will result in lower levels of bugginess.

	Bugginess	*Statistical Significance*
Used sub-cycles (1=Yes, 0=No)	0.086	0.5225
# Sub-cycles	-0.069	0.6068
Sub-cycle length (% of project duration)	-0.053	0.6933

Table 7-21. Pearson Product-Moment Correlation of bugginess and sub-cycles.

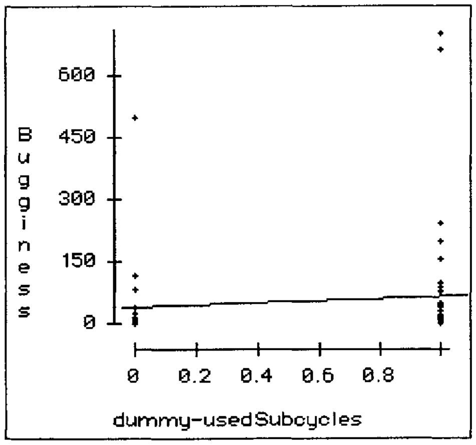

Figure 7-17a.

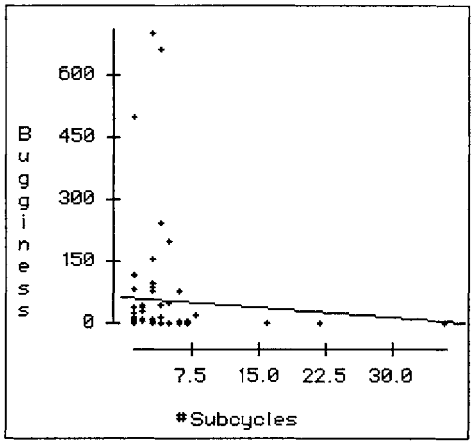

Figure 7-17b.

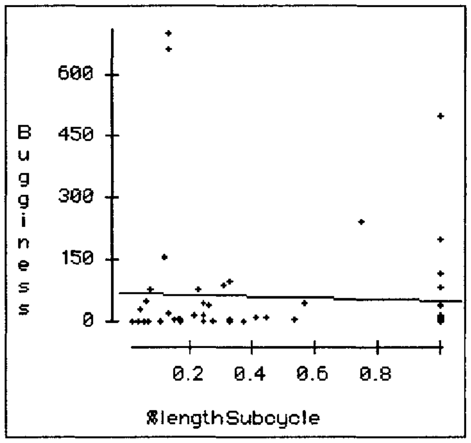

Figure 7-17c.

Figures 7-17. Scatterplots of bugginess vs. sub-cycles. There were 57 projects in the sample.

d. Dummy variable of sub-cycle use
e. Number of sub-cycles
f. Sub-cycle length as a percentage of the project duration

There are no statistically significant correlations between bugginess and the different aspects of sub-cycles. A possible explanation is that old bugs are fixed and new bugs are introduced in each iteration, regardless of length, such that the overall bugginess level is unaffected.

Hypothesis 7: A project team that runs regression tests after each modification to the code base will have lower levels of bugginess.

	Bugginess	*Statistical Significance*
Ran Regression Test (1=Yes, 0=No)	-0.247	0.0612

Table 7-22. Pearson Product-Moment Correlation of bugginess and the use of regression test after code base modifications.

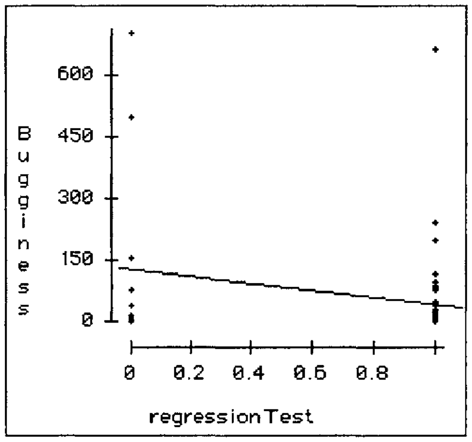

Figure 7-18. Scatterplot of bugginess and the practice of running regression tests after code base modifications. There were 58 projects in this sample.

There is a negative correlation between bugginess and the practice of running regression tests after each code base modification that is statistically significant at the 0.10 level but not at the 0.05 level. This practice prevents bugs from accumulating in the system. Assuming the original code base was mainly error-free, the bugs uncovered after the regression test can also be located more easily because the new code introduced the new bugs to the existing code. These advantages reduce the overall bugginess level of the project.

Hypothesis 8: More developer participation in testing contributes to lower levels of bugginess.

	Bugginess	*Statistical Significance*
Developers write test cases (1=Yes, 0=No)	0.089	0.5834
% of Code tested by developers	-0.192	0.2354

Table 7-23. Pearson Product-Moment Correlation of bugginess and the developer participation in testing.

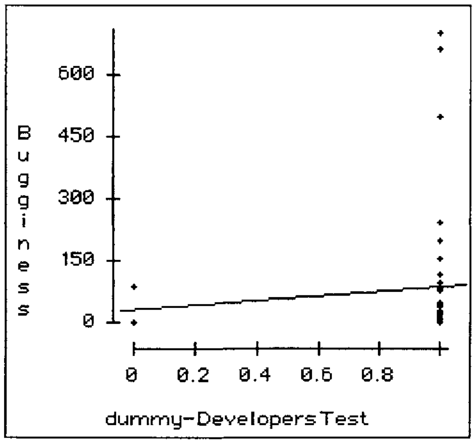

Figure 7-19a.

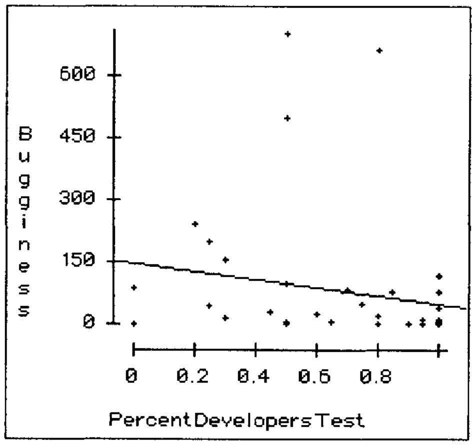

Figure 7-19b.

Figures 7-19. Scatterplots of bugginess and developer participation in testing. There were 40 projects in this sample.

a. Dummy variable for developer participation in testing.
b. Percentage of code tested by developers.

The correlations between bugginess and developer participation in testing are not statistically significant. There is a stronger negative correlation between bugginess levels and the percentage of code tested by developers than between bugginess levels and the practice of developer participation in testing. This suggests that bugginess level decreases as developers test a higher portion of the code. A possible reason is that developers are generally more efficient testers than the QA staff because they know the code, and therefore know what to test for and where bugs may occur.

<u>Hypothesis 9</u>: The level of bugginess decreases as a higher percentage of tests are automated.

	Bugginess	*Statistical Significance*
% of automated tests	-0.054	0.6889

Table 7-24. Pearson Product-Moment Correlation of bugginess and the percentage of automated tests.

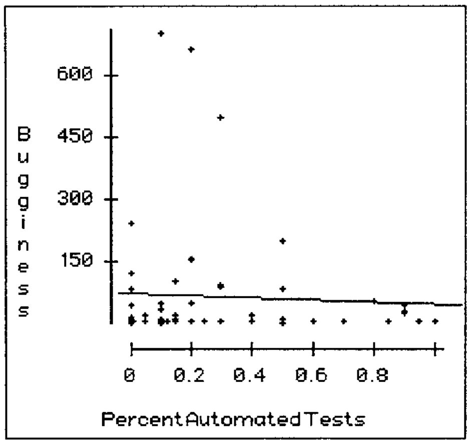

Figure 7-20. Scatterplot of bugginess the percentage of automated tests. There were 58 projects in this sample.

The correlation between bugginess and percentage of automated tests is not statistically significant. There is a slight negative relationship, which suggests that automated tests do help lower bugginess but not significantly.

Sensitivity Analysis

Since there were a large number of projects from HP-Agilent, a sensitivity analysis was run to see whether they had significant impact on the original analysis of bugginess. The results are shown in Table 7-25.

	Original	HP-Agilent excluded	
	Significance Level	**Correlation**	**Significance Level**
% of Functional specification finished	> 0.10	-0.362	≤ 0.05
% of Detailed specification finished	> 0.10	-0.342	≤ 0.10
Ran regression test (1=Yes, 0=No)	≤ 0.10	-0.437	≤ 0.05

Table 7-25. Summary of analysis with the exclusion of HP-Agilent projects.

Compared to other projects, the bugginess levels of the projects from HP-Agilent are less affected by functional and detailed specification completion and the practice of running regression tests after each change to the code base.

7.4 Data Analysis: Timeliness

Timeliness is measured by schedule estimation error as a percentage (see Equation 4). The smaller the error, the closer the project finished according to the original schedule.

Hypothesis 1: More experienced project teams have smaller schedule estimation errors.

	Timeliness	*Statistical Significance*
% of Developers with 0-5 years experience	0.206	0.1051
% of Developers with 6+ years experience	-0.206	0.1051

Table 7-26. Pearson Product-Moment Correlation of timeliness and developers' experience.

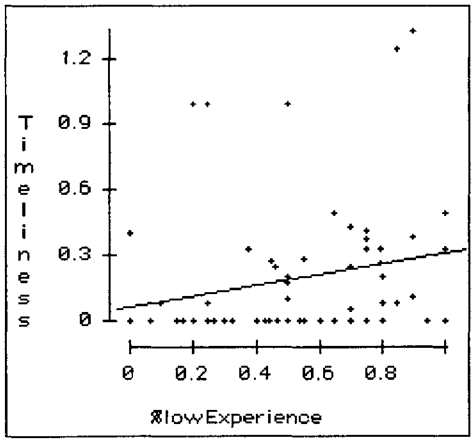

Figure 7-21a.

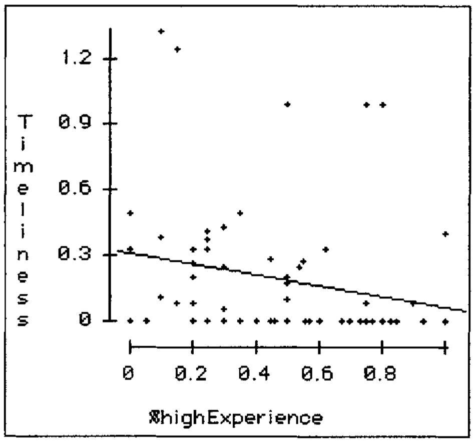

Figure 7-21b.

Figures 7-21. Scatterplots of timeliness vs. percentage of developers with different levels of experience. A total of 64 projects were in this sample.

a. Percentage of developers with 0-5 years of experience
b. Percentage of developers with 6-10+ years of experience

The correlation between timeliness and the percentage of project teams with less than 6 years of experience is 0.206, and the correlation between timeliness and the percentage of project teams with 6 and more years of experience is –0.206. Neither of the correlations is statistically significant. The positive correlation of timeliness and less experienced teams indicates that less experienced teams are more prone to scheduling errors than more experienced teams, which had a negative correlation to timeliness. An interesting point for further investigation is the managers' years of experience, which should have a direct effect on schedule estimation error.

Hypothesis 2: Completing higher percentages of specification documents before coding contributes to smaller schedule estimation errors.

	Timeliness	*Statistical Significance*
% of Architectural specification finished	0.090	0.4200
% of Functional specification finished	0.009	0.9331
% of Detailed specification finished	-0.261	0.0170

Table 7-27. Pearson Product-Moment Correlation of timeliness and percentage of specifications completed before coding.

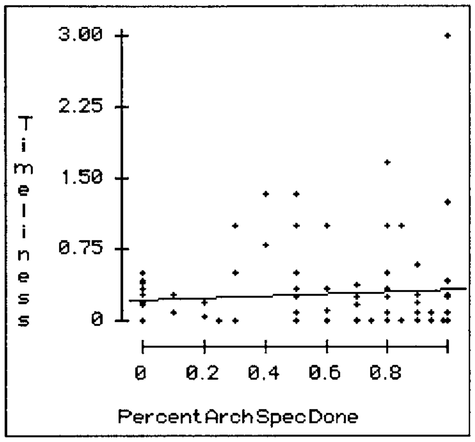

Figure 7-22a.

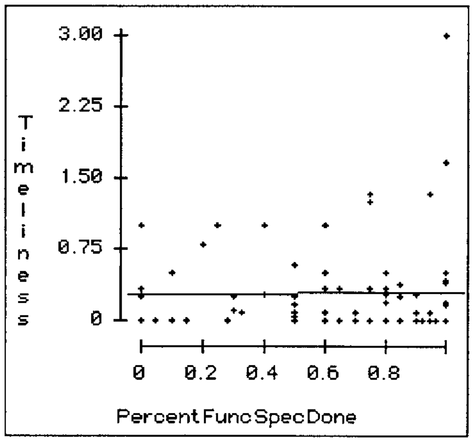

Figure 7-22b.

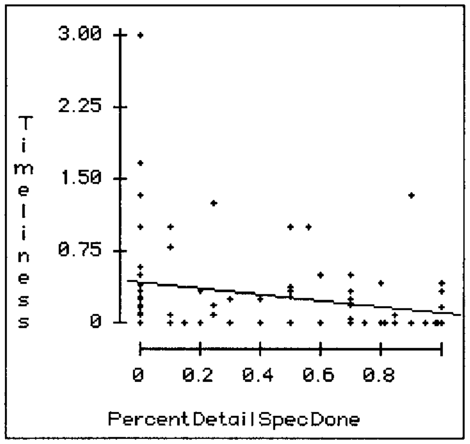

Figure 7-22c.

Figures 7-22. Scatterplots of timeliness vs. percentage of specification documents completion before coding. There were 83 projects in the sample.

a. Architectural specification
b. Functional (or requirements) specification
c. Detailed specification

Out of the three correlations, only the correlation between timeliness and percentage of detailed specification documents completed before coding is statistically significant at the

0.05 level. The other two correlations are statistically insignificant. Well-planned detailed designs contribute to more efficient coding because developers can concentrate on coding once most of the design is finished. This efficiency, in turn, helps the project team stay on schedule.

Hypothesis 3: Projects that receive early technical and market feedback have smaller schedule estimation errors.

	Timeliness	*Statistical Significance*
% of Final functionality in first prototype	-0.116	0.3107
% of Final functionality in first integration	-0.205	0.0705
% of Final functionality in first beta	-0.035	0.7587

Table 7-28. Pearson Product-Moment Correlation of timeliness and percentage of the final functionality present in early project events.

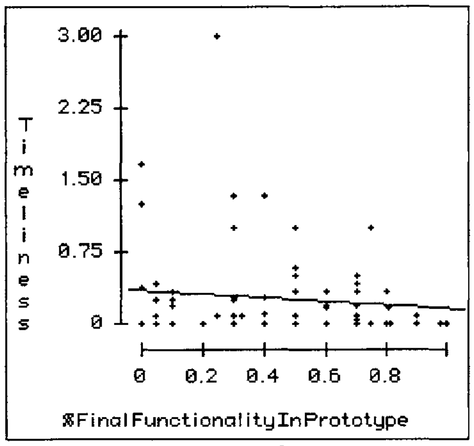

Figure 7-23a.

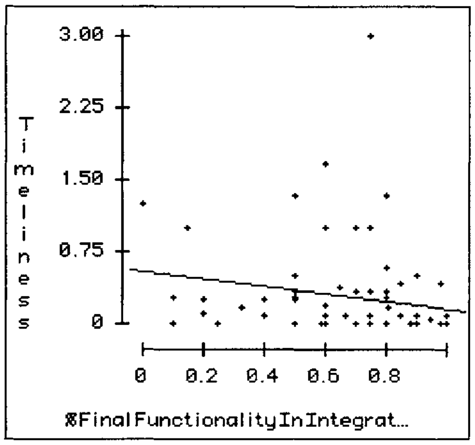

Figure 7-23b.

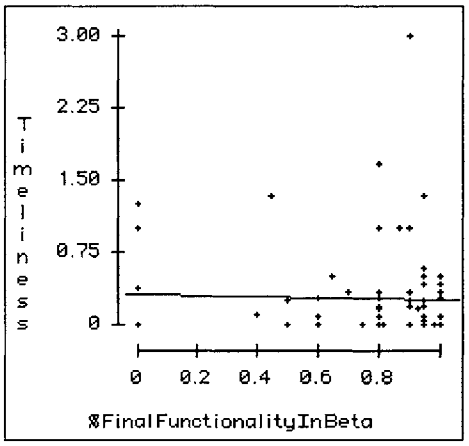

Figure 7-23c.

Figures 7-23. Scatterplots of timeliness vs. percentage of the final functionality present in early project events. There were 79 projects in the sample.

a. First prototype
b. First integration
c. First beta

The correlations between timeliness and feedback are negative but not statistically significant. This may be because project teams that receive early feedback continuously incorporate feedback throughout the development process, which causes schedule delays. There is a time cost to modifying functionality.

Hypothesis 4: The practice of design reviews reduces schedule estimation error.

	Timeliness	*Statistical Significance*
Conducted design reviews (1=Yes, 0=No)	-0.090	0.4250
# Design reviews	-0.106	0.3491

Table 7-29. Pearson Product-Moment Correlation of timeliness and design reviews.

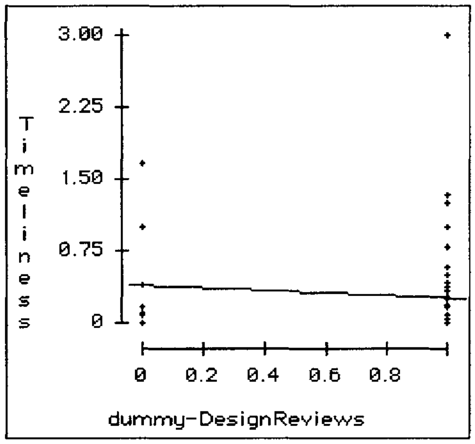

Figure 7-24a.

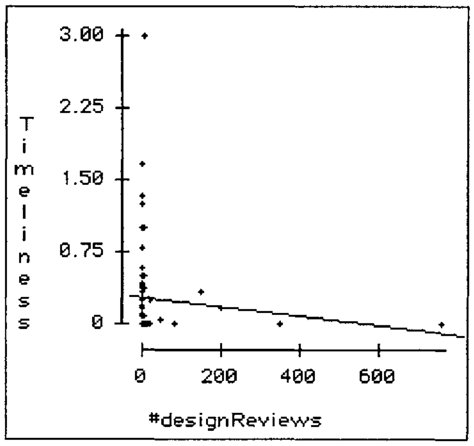

Figure 7-24b.

Figures 7-24. Scatterplots of timeliness vs. design reviews. There were 80 projects in the sample.

a. Dummy variable of design reviews
b. Number of design reviews conducted

A dummy variable was used for the practice of design reviews. The correlations between timeliness and design reviews are negative but not statistically significant. Further analysis on the data cluster at the lower left-hand corner of Figure 7-24b did not yield a statistically significant correlation (-0.155 with p=0.1821) between timeliness and the number of design reviews. However, the negative correlations of the overall and the cluster samples suggest that more design reviews reduce schedule estimation error. This is probably because a well-planned design requires less rework and backtracking later in the development process, which helps project teams stay on schedule.

Hypothesis 5: The practice of code reviews means reduces schedule estimation error.

	Timeliness	*Statistical Significance*
Conducted code reviews (1=Yes, 0=No)	-0.260	0.0242
% Code reviewed	-0.195	0.0928

Table 7-30. Pearson Product-Moment Correlation of timeliness and code reviews.

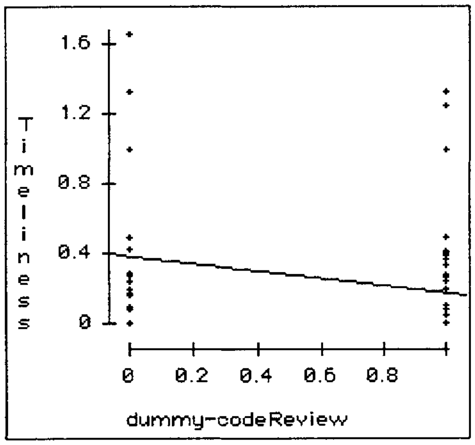

Figure 7-25a.

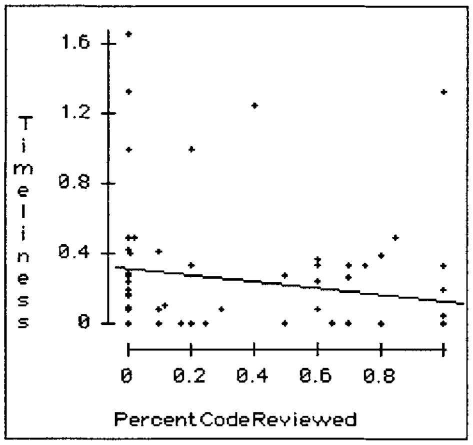

Figure 7-25b.

Figures 7-25. Scatterplots of timeliness vs. code reviews. There were 75 projects in the sample.

a. Dummy variable of code reviews
b. Percentage of code reviewed

The correlation between timeliness and the practice of code reviews is statistically significant at the 0.05 level. Project teams that review code can uncover and fix errors earlier in the development process, which takes less time than realizing and correcting bugs later during development. There is a positive yet statistically insignificant correlation between timeliness and the percentage of code reviewed. More bugs are uncovered and corrected earlier and more quickly as more code is reviewed, which helps a project stay on schedule. Early bug fixing is easier and less time-consuming than bug fixing later in the development phase.

Hypothesis 6: Projects that use sub-cycles have smaller schedule estimation error.

	Timeliness	*Statistical Significance*
Used sub-cycles (1=Yes, 0=No)	0.195	0.0760
# Sub-cycles	0.134	0.2226
Sub-cycle length (% of project duration)	-0.161	0.1440

Table 7-31. Pearson Product-Moment Correlation of timeliness and sub-cycles.

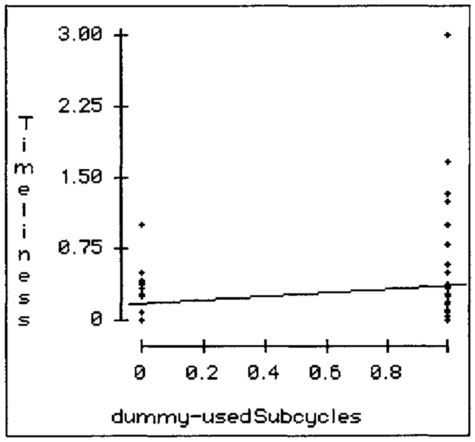

Figure 7-26a.

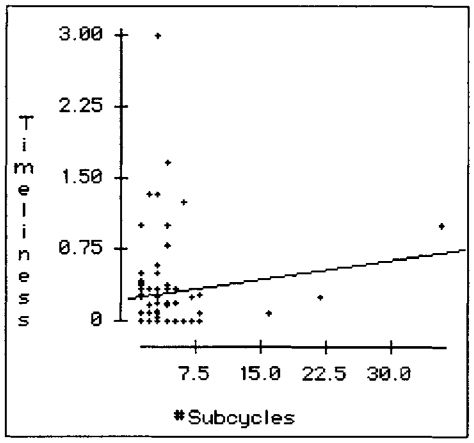

Figure 7-26b.

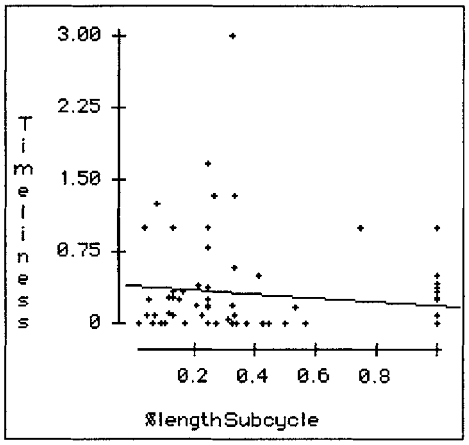

Figure 7-26c.

Figures 7-26. Scatterplots of timeliness vs. sub-cycles. There were 84 projects in the sample.

a. Dummy variable of sub-cycle use
b. Number of sub-cycles
c. Sub-cycle length as a percentage of the project duration

The correlations between timeliness and sub-cycles are not statistically significant. There is a positive correlation between timeliness and the use of sub-cycles. Using sub-cycles allows a project team to continuously evaluate their situation and adjust their development schedule at the beginning of each iteration to finish by deadline. There is a positive correlation between timeliness and the number of sub-cycles and a negative correlation between timeliness and sub-cycle length. This is because a project team that develops with more sub-cycles usually has shorter sub-cycles. Frequent iteration means smaller schedule estimation error because schedule estimation is more accurate with each iteration and as the project approaches the final deadline.

Hypothesis 7: A project team that runs regression tests after each modification to the code base will have a smaller schedule estimation error.

	Timeliness	*Statistical Significance*
Ran regression test (1=Yes, 0=No)	-0.222	0.0410

Table 7-32. Pearson Product-Moment Correlation of timeliness and the use of regression test after code base modifications.

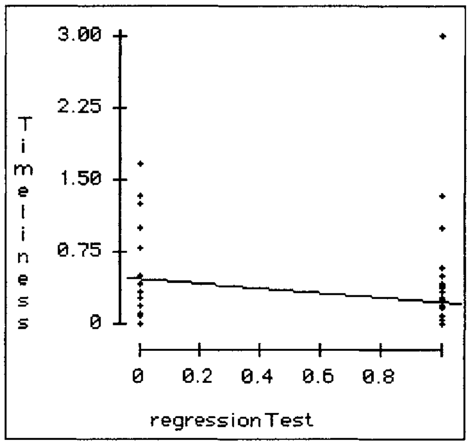

Figure 7-27. Scatterplot of timeliness and the practice of running regression tests after code base modifications. There were 85 projects in this sample.

The negative correlation between timeliness and running regression tests after changes to the code base is statistically significant at the 0.05 level, which means that this practice helps project teams stay close to schedule. This is because these regression tests find and fix bugs early in the development process so that less time is spent on bug fixing, which is a more time-consuming task later in development.

Sensitivity Analysis

A sensitivity analysis was run to evaluate whether projects from a certain company or country had a significant impact on the original timeliness analyses. The tests were conducted for projects from HP and Agilent. The results are shown in Table 7-33.

	Original	HP-Agilent excluded	
	Significance Level	**Correlation**	**Significance Level**
% High/Low experience	> 0.10	-0.358/0.358	≤ 0.05
% of Architectural specification finished	> 0.10	-0.251	≤ 0.10
% of Detailed specification finished	≤ 0.05	-0.106	> 0.10
Conducted code reviews (1=Yes, 0=No)	≤ 0.05	0.022	> 0.10
Ran regression test (1=Yes, 0=No)	≤ 0.05	-0.185	> 0.10

Table 7-33. Summary of analysis with the exclusion of HP-Agilent projects.

The timeliness of the projects from HP-Agilent are not affected by project team experience levels or architectural specifications compared to the other projects. However, the HP-Agilent are strongly affected by detailed specifications, code reviews, and regression tests after code base modifications.

7.5 Data Analysis: Budget Error

Over-expenditure level is measured by budget estimation error (see Equation 5). The values of budget error are percentages.

Hypothesis 1: More experienced project teams are less likely to exceed their budget.

	Budget Error	*Statistical Significance*
% of Developers with 0-5 years experience	0.075	0.5926
% of Developers with 6+ years experience	-0.075	0.5926

Table 7-34. Pearson Product-Moment Correlation of budget error and developers' experience.

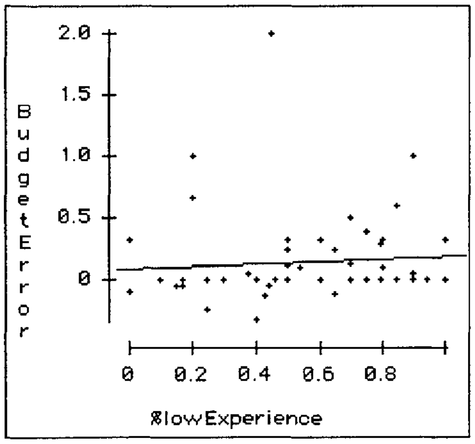

Figure 7-28a.

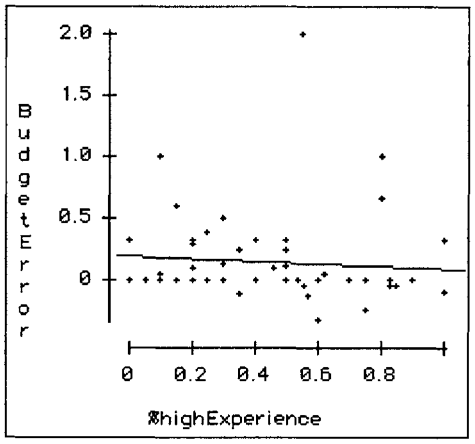

Figure 7-28b.

Figures 7-28. Scatterplots of budget error vs. percentage of developers with different levels of experience. A total of 54 projects were in this sample.

a. Percentage of developers with 0-5 years of experience
b. Percentage of developers with 6+ years of experience

The correlation between budget error and the percentage of team members with less than 6 years of experience is 0.075, and the correlation between budget error and the percentage of team members with 6 and more years of experience is –0.075. Both of these relationships are statistically insignificant. However, from Figure 7-29b, it can be observed that some projects spent less than their budgets (negative budget errors) after the 30% mark. Along with the positive correlation, this suggests that more experienced teams better manage their budgets.

Hypothesis 2: Project teams with higher percentages of specification documentation completion before coding are less likely to exceed budget.

	Budget Error	*Statistical Significance*
% of Architectural specification finished	-0.277	0.0212
% of Functional specification finished	-0.330	0.0056
% of Detailed specification finished	-0.116	0.3431

Table 7-35. Pearson Product-Moment Correlation of budget error and percentage of specifications completed before coding.

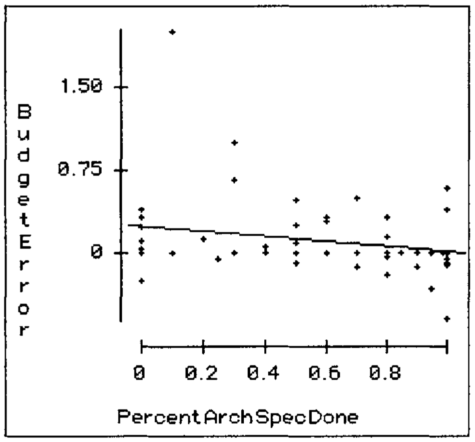

Figure 7-29a.

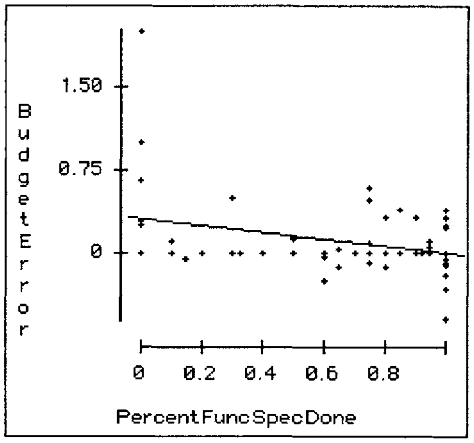

Figure 7-29b.

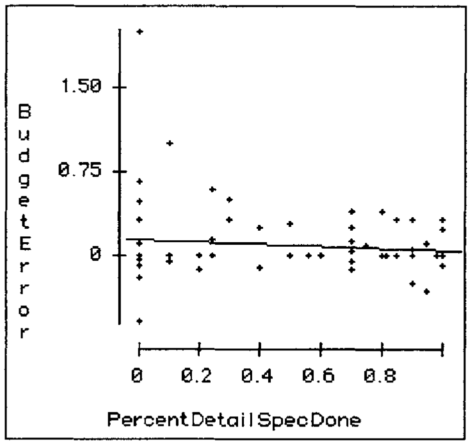

Figure 7-29c.

Figures 7-29. Scatterplots of budget error vs. percentage of specification documents completion before coding. There were 69 projects in the sample.

a. Architectural specification
b. Functional (or requirements) specification
c. Detailed specification

Attention to higher level design is strongly correlated with minimizing budget error whereas detailed design is not. The correlation between budget error and the percentage of completeness of architectural specification documentation before coding is statistically significant at the 0.05 level, and the correlation of the budget error and the percentage of completeness of functional specification documentation before coding is statistically significant at the 0.01 level. Although the correlation between budget error and the percentage of detailed specification completed before coding is not statistically significant, the negative correlation is consistent with the other two documentation relationships with budget error. These results indicate that spending sufficient time to create a good design helps a project team stay within their budget because starting with a good design helps a team avoid the need to redesign later in the development phase. Redesigns are expensive both monetarily and temporally.

Hypothesis 3: Earlier technical and market feedback lead to a lower likelihood of exceeding budget.

	Budget Error	*Statistical Significance*
% of Final functionality in first prototype	-0.061	0.6246
% of Final functionality in first integration	-0.227	0.0663
% of Final functionality in first beta	-0.093	0.4586

Table 7-36. Pearson Product-Moment Correlation of budget error and percentage of the final functionality present in early project events.

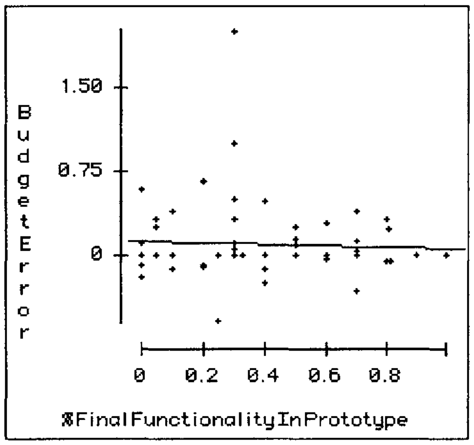

Figure 7-30a.

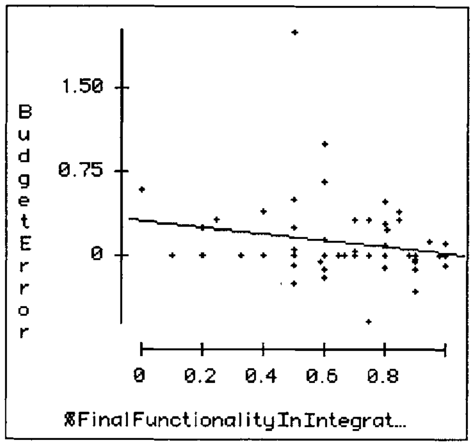

Figure 7-30b.

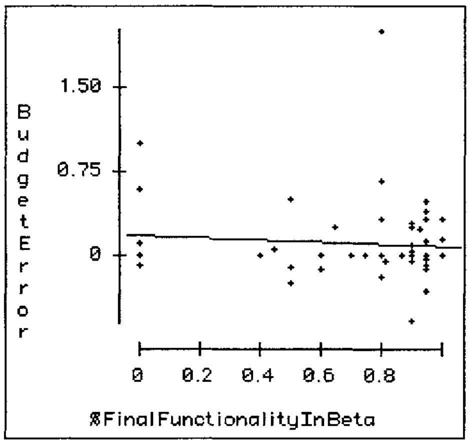

Figure 7-30c.

Figures 7-30. Scatterplots of budget error vs. percentage of the final functionality present in early project events. There were 66 projects in the sample.

a. First prototype
b. First integration
c. First beta

The correlations between budget error and feedback are negative but not statistically significant. This suggests that project teams that receive earlier feedback have a higher likelihood of overspending. This could be because the projects that receive early feedback are more likely to incorporate feedback throughout development. Constant feature change during development is costly because implemented features may be replaced or redesigned to accommodate the addition of new features. To add new features, the project team may need to backtrack and change the initial design to integrate the new features, which costs both time and money. Projects that do not incorporate much feedback do not go through this expensive process of changing functionalities.

The correlation of budget error and technical feedback (% of final functionality in the first integration) is stronger than the correlations of budget error and market feedback (% of final functionality in the first prototype and % of final functionality in the first beta). This suggests that technical feedback has a greater impact on project expenditure than market feedback.

Sensitivity Analysis

Since a large number of the submitted projects were from one company (HP and Agilent are treated as one company), the analysis was rerun with the exclusion of these projects.

The results of the reanalysis are shown below in Table 7-37

	Original	HP-Agilent excluded	
	Significance Level	**Correlation**	**Significance Level**
% Low/High experience	> 0.10	0.294/-0.294	≤ 0.10
% of Architectural specification finished	≤ 0.05	-0.290	≤ 0.10
% of Functional specification finished	≤ 0.01	-0.346	≤ 0.05

Table 7-37. Summary of analysis with the exclusion of HP-Agilent projects.

An explanation of the correlation difference for experience levels is that experience does not greatly affect overspending budget at HP-Agilent. Regarding the percentages of architectural and functional specification completed before coding, their degrees of completeness affect expenditure more at HP-Agilent compared to the other projects.

7.6 Data Analysis: Customer satisfaction perception rating

Customer satisfaction perception rating was measured on a 5 point scale where 1 means the project's perceived customer satisfaction rating was significantly below its expectations and 5 means its perceived customer satisfaction rating was significantly above its expectations.

Hypothesis 1: Projects that receive earlier feedback have higher customer satisfaction perception ratings.

% of Final functionality in first prototype	#Projects	Min	Max	Average	Std Dev	Median
0-20%	22	1	5	3.22727	0.922307	3
21-40%	17	2	5	3.35294	0.86177	3
41-60%	17	2	5	3.52941	0.799816	4
61-80%	11	2	5	3.09091	0.94388	3
80-100%	8	2	5	3.875	0.991031	4

Table 7-38. Summary of customer satisfaction perception ratings and the percentage of final functionality in the first prototype.

There is generally an increasing trend between the percentage of final functionality in the first prototype and customer satisfaction perception ratings. The 61-80% group seems to be an anomaly because its average is a lot lower compared to the average ratings of the other four percentage groups. This table shows that the more complete the first prototype, the higher the customer satisfaction perception rating.

% of Final functionality in first beta	#Projects	Min	Max	Average	Std Dev	Median
0-20%	10	2	5	3.6	1.07497	3
21-40%	0	-	-	-	-	-
41-60%	6	3	4	3.33333	0.516398	3
61-80%	13	2	5	3.23077	1.16575	3
80-100%	46	1	5	3.36957	0.826201	3

Table 7-39. Summary of customer satisfaction perception ratings and the percentage of final functionality in the first beta.

Projects that have 0-20% of the final functionality present in their first beta had the highest overall average for customer satisfaction perception ratings. The reason could be that these projects developed incrementally and incorporated more customer feedback, which explains the low percentage of final functionality. Therefore these projects have more customer interaction than the other projects.

% of Final functionality in first integration	#Projects	Min	Max	Average	Std Dev	Median
0-20%	5	3	5	4	1	4
21-40%	5	2	4	3	1	3
41-60%	19	2	5	3.31579	0.945905	3
61-80%	22	1	4	3.13636	0.83355	3
80-100%	24	2	5	3.58333	0.829702	4

Table 7-40. Summary of customer satisfaction perception ratings and the percentage of final functionality in the first integration.

The projects that received the earliest technical feedback (0-20% final functionality in the first integration) have a higher average customer satisfaction perception rating than the other projects. However, only 5 projects fall in this group so the data may not be an accurate representation. The projects that had 81-100% of the final functionality in the first integration also had a high average customer satisfaction perception rating. These could be projects that use the waterfall model for development, which explains the large percentage of the final functionality implemented before the first integration. If the requirements were well defined before development and do not change much as the project progresses, which is characteristic of most waterfall methods, the customer will get what they initially asked for. This may explain why the customer satisfaction perception rating is high.

Hypothesis 2: More beta releases increase customer satisfaction perception ratings.

# of Betas	#Projects	Min	Max	Average	Std Dev	Median
0	24	2	5	3.33333	0.916831	3
1	17	2	5	3.52941	0.799816	4
2	19	1	5	3.42105	1.01739	4
3	11	2	4	3.27273	0.786245	3
4+	8	2	5	3.375	0.916125	3

Table 7-41. Summary of customer satisfaction perception ratings and number of beta releases.

There were a large number of projects in the sample that did not have a beta release. Their average customer satisfaction perception rating was 3.33333, which is lower than the average satisfaction perception ratings of the projects that released one, two, and four more betas. However, the customer rating for three beta releases was lower than the customer ratings for no beta releases. This may be an anomaly in the data. The table shows that customer satisfaction perception ratings are generally higher if projects release betas. This is probably because betas increase customer involvement in the projects.

Hypothesis 3: Projects that use sub-cycles have higher customer satisfaction perception ratings.

# of Sub-cycles	#Projects	Min	Max	Average	Std Dev	Median
1	30	2	5	3.46667	0.819307	4
2 -3	19	2	4	3.15789	0.60214	3
4 -5	17	2	5	3.23529	0.970143	3
6+	12	1	5	3.83333	1.19342	4

Table 7-42. Summary of customer satisfaction perception ratings and the number of sub-cycles.

Sub-cycle Length	#Projects	Min	Max	Average	Std Dev	Median
0-25%	29	1	5	3.41379	0.982607	4
26-50%	14	2	4	3.14286	0.662994	3
51-75%	3	2	4	3	1	3
76-100%	30	2	5	3.43333	0.8172	3.5

Table 7-43. Summary of customer satisfaction perception ratings and sub-cycle length as a percentage of the project duration.

Out of the projects that developed in sub-cycles, those that used more sub-cycles had higher customer satisfaction perception ratings. The projects that used the waterfall model (i.e., those with one sub-cycle) have a higher average rating than the projects with two to five sub-cycles but have a lower average rating than the projects with six or more sub-cycles.

The projects with sub-cycle lengths between 76% and 100% of their actual project duration are most likely projects that use the waterfall method. With this assumption, the projects with shorter sub-cycles have higher customer satisfaction perception ratings.

This is probably because these projects have more interaction with the customer, which increases customer satisfaction perception rating.

7.7 Summary of Results

This section shows a quick summary of the study's results. The table below presents the outcome variables and the process variables that affect the outcome variables. The significance levels, denoted by *p*, of each correlation between outcome and process variables are provided. A lower significance level indicates a stronger relationship between the variables.

Outcome Variables	**Process Variables**
Budget Error	Percent of architectural specification completed before coding. There is a negative correlation ($p \leq 0.05$). Overspending levels reduce as more architectural specifications are documented before coding begins.
	Percent of functional specification completed before coding. There is a strong negative correlation ($p \leq 0.01$). The degree of overspending is greatly reduced as more functional specifications are documented before coding begins.
	Percent of final functionality present in the first integration. There is a weak negative correlation ($p \leq 0.10$). Early technical feedback helps projects reduce their levels of overspending.
Bugginess	Running regression tests after each code base modification. There is a weak negative correlation ($p \leq 0.10$). Bugginess levels reduce with the practice of running regression or integration tests (and not compile and link tests) after each change to the code base.
Customer Satisfaction	Waterfall model. Projects that use the waterfall model had high customer satisfaction perception ratings.

	Incremental development. Projects with short and frequent sub-cycles had higher customer satisfaction perception ratings than the projects that use the waterfall model. Projects that use long and few sub-cycles had lower customer satisfaction perception ratings than the projects that use the waterfall model.
Productivity	Percent of architectural specification completed before coding. There is a weak positive correlation ($p \leq 0.10$). Productivity increases when more architectural specification are documented before coding begins.
	Running regression tests after each code base modification. There is a positive correlation ($p \leq 0.05$). The practice of running regression or integration tests after changes to the code base increases overall productivity levels.
	Percent of automated tests. There is a weak negative correlation ($p \leq 0.10$). Productivity levels decrease as more tests are automated.
Timeliness	Percent of detailed specification completed before coding. There is a negative correlation ($p \leq 0.05$). Schedule estimation error decreases as more detailed specifications are documented before coding begins.
	Percent of final functionality present in the first integration. There is a weak negative correlation ($p \leq 0.10$). Early technical feedback reduces schedule estimation error.
	Use of code reviews. There is a negative correlation ($p \leq 0.05$). The practice of code reviews reduces schedule estimation error.

	Percent of code reviewed. There is a weak negative correlation ($p \leq 0.10$). Schedule estimation error decreases as more code is subjected to review.
	Use of sub-cycles. There is a weak positive correlation ($p \leq 0.10$). Incremental development increases schedule estimation error.
	Running regression tests after each code base modification. There is a negative correlation ($p \leq 0.05$). This practice reduces schedule estimation error.

Table 7-44. Summary of the data analysis results.

7.8 Data Analysis: Country

This section briefly summarizes the software industry of India, Japan, and the United States.

India

The Indian software industry gained global visibility in the mid-1980s when three large American organizations (Citibank, Texas Instruments, and Hewlett-Packard) established subsidiaries in India [19]. In the 1990s, outsourcing software activities became popular. With a large English speaking population and a skilled labor force, India was well positioned to be the receiving end of the outsourcing contracts, many from American firms. As a result, professional services dominate Indian software exports.

According to the Software Engineering Institute, 43 of the 66 organizations that have reached CMM Level 5 since January 2002 are located in India [25]. Software developers in India pursue quality relentlessly because "It's almost shameful for them to admit they are a [CMM] Level 2 company..." says Satish Bangalore who is a managing director of Phoenix Global Solutions in Bangalore [2]. And according to Deepandra Moitra, a general manager at Lucent Technologies India, Indian companies use these ratings to signal quality when competing for outsourcing contracts [18].

Japan

The software industry in Japan has been heavily influenced by the manufacturing industry, which has a long history in Japan [21]. Many Japanese software organizations use Total Quality Control, a technique for total performance improvement borrowed from the manufacturing industry.

Computer equipment manufacturers are at the top of the Japanese software industry hierarchy and small software companies are at the bottom level of the hierarchy. The term "software factory" describes how software development occurs in the computer manufacturing companies. In these firms, software development is the cooperative effort of large teams that focus on quality, productivity, and process standardization. In these factories, software reuse is a major component of quality and productivity improvement and development follows the waterfall model. These organizations also keep meticulous records of its organizational history, e.g. individual worker productivity, such that schedule and costs can be more accurately calculated for future projects [1, 8]. Most of the software development in Japan occurs in the lower levels of the hierarchy. The developers at these small software companies are poorly educated [21] and poorly managed [12].

The software industry in Japan suffer from the following problems [21]: old-fashioned software technologies and development styles, delays in international technology adaptation due to the language barrier, remote and relatively closed software organizations, lack of creativity, and the weak connection between academia and the industry. To address these issues, in 1997, the Software Engineering Association (SEA) in Japan initiated a movement to improve software process. SEA translated the Capabilities Maturity Model manual to Japanese, which was published in 1999. This translation actuated development improvements in the industry.

United States

In the summer of 1969, IBM announced the decision to unbundled its software from its hardware. Previously, customers who purchased the hardware received complimentary software. The independent software vendor sector emerged from this move. Because of the first mover advantage, the American software industry holds strong positions in domestic and international markets. The United States continues to maintain this lead because of its established software infrastructure [29].

The United States is also a leader of software innovations. However, foreign software industries and not the American software industry adapt and integrate the many software technologies, processes, and models that originated from the United States. The CMM is an example of an American theory embraced in another country-India. Most of the US companies that have been assessed for CMM certification are at Level 2 [14].

Results
This section investigates how software development practices differ across countries. Only specific comparisons of the countries of India, Japan, and the USA were possible due to the insufficient response numbers from the other countries. The other countries, which include Bahrain, Canada, China, France, Hong Kong, Switzerland, and Taiwan, are grouped together for the comparisons under the name "Other".

Code generation
A dummy variable for code generation was used such that the dummy variable was set to 1 if a project had a non-zero percentage of code generated automatically, else 0. It is assumed that code generation is not practiced if no answer was provided. Table 7-45 shows the breakdown among India, Japan, the USA, and across the sample.

Code generation	India	Japan	USA	Other	All
Yes	6	10	7	6	29
No	6	17	32	3	58
Total	12	27	39	9	87

Table 7-45. Summary of code generation practice across selected countries compared to the entire sample.

Approximately one-third of the projects practiced code generation. A high percentage of the projects received from India (50%) and Japan (~ 37%) used the practice of code generation compared to the USA (~ 18%). Two-thirds of the sample from the remaining countries automatically generated code.

Specification Documents
Since the CMM model advocates consistent documentation of activities during the development cycle, a high percentage of the Indian sample should have the practice of formal specification documentation. The breakdown of the three specification documents-architectural, functional, and detailed-is shown in Tables 7-46 to 7-48 below.

Architectural	India	Japan	USA	Other	All
Yes	9	19	31	8	67
No	3	8	8	1	20
Total	12	27	39	9	87

Table 7-46. Summary of architectural specification documentation across selected countries compared to the entire sample.

About 77% of the entire sample formally documented architectural specifications. The US leads the three countries with ~79% of its sample writing architectural documentation, followed by India with 75% and Japan with ~70%. Approximately 89% of the projects from the other countries wrote formal architectural specification documentation.

Functional	India	Japan	USA	Other	All
Yes	11	24	33	9	79
No	1	3	6	0	15
Total	12	27	39	9	87

Table 7-47. Summary of functional (or requirements) specification documentation across selected countries compared to the entire sample.

Across the entire sample, about 91% formally documented functional or requirements specifications. In comparison, ~92% of the Indian projects, ~89% of the Japanese projects, and 100% of the other projects had functional specification documents. The US lags behind with only ~85% of the surveyed projects with formal documentation of functional specifications.

Detailed	India	Japan	USA	Other	All
Yes	12	23	19	7	61
No	0	4	20	2	26
Total	12	27	39	9	87

Table 7-48. Summary of detailed specification documentation across selected countries compared to the entire sample.

About 70% of the sample formally documented detailed design specifications. India and Japan have higher percentages of projects that had detailed design documentation, 100% and ~85% respectively, compared to the overall average and the other countries' average.

About 78% of the sample from the other countries used formal detailed specification documentation. The US, however, follows behind the overall average and all the other countries with ~49% of its projects that engaged in detailed specification documentation.

Among the three selected countries, the projects from India lead in terms of formally documenting specifications. This Indian sample is consistent with the current state of the Indian software community where there is a constant strive for CMM Level 5 status.

Sub-cycles

Table 7-49 shows the breakdown of sub-cycle usage across the three countries compared to the whole sample.

Sub-Cycles	India	Japan	USA	Other	All
Yes	9	13	24	9	55
No	3	14	15	0	32
Total	12	27	39	9	87

Table 7-49. Summary of the use of sub-cycles during development across selected countries compared to the entire sample.

Sub-cycles are a sign of flexible design because they allow software to be developed incrementally such that requirements can be modified during development. The majority of the sample (~63%) engaged in the practice of sub-cycles. 75% of the projects from India and ~62% of the US projects developed in sub-cycles while less than half of the Japanese projects used sub-cycles for development. 100% of the other countries developed software in sub-cycles.

The Japanese sample trails the other countries in the practice of incremental development. This observation is consistent with the software factory approach to development of the large computer equipment manufacturers, which is to follow the waterfall method.

Design Reviews

The breakdown of the practice of design review is seen in Table 7-50.

Design Reviews	India	Japan	USA	Other	All
Yes	12	26	31	8	77
No	0	1	8	1	10
Total	12	27	39	9	87

Table 7-50. Summary of design reviews across selected countries compared to the entire sample.

High percentages of the projects from India, Japan, and the other countries conducted design reviews as part of the development process with India at 100%, Japan at ~96% and the other countries at ~89%. The majority of the US sample, about 79%, also practiced design reviews but this percentage is lower than the overall average of ~89%.

Code Reviews

Table 7-51 shows the breakdown of the practice code reviews.

Code Reviews	India	Japan	USA	Other	All
Yes	12	19	23	8	62
No	0	8	16	1	25
Total	12	27	39	9	87

Table 7-51. Summary of code reviews across selected countries compared to the entire sample.

Like design reviews, 100% of the Indian sample and ~89% of the other countries' sample conducted code reviews. Compared to design reviews, fewer projects in the Japanese (~70%) and US (~59%) samples conducted code reviews. The overall average was lower, at ~71%, than the overall average of design reviews which was ~89%.

It is not surprising to see that all the projects from India conducted both design and code reviews. This result is consistent with the Indian effort to produce quality.

Build Frequency

Table 7-52 shows the number of projects for each country category that conducted daily builds during three different times of development-the first third, the middle third, and the last third.

Daily Builds	India	Japan	USA	Other	All
Beginning	1	6	11	0	18
Middle	2	7	10	3	22
End	1	10	10	3	24
Total Projects	12	27	39	9	87

Table 7-52. Summary of daily builds during the different time periods of development across selected countries compared to the entire sample.

Overall, the percentage of projects that conducted daily builds increases as development progresses. Only Japan shows the same increasing trend. On average, the Japanese sample leads with ~28% that had daily builds followed by ~26% of the USA sample, ~22% of the other countries, and ~11% of the Indian projects.

Integration or Regression Test

The practice of running an integration or regression test after changes are made to the code base is broken down as shown in Table 7-53.

Regression Test	India	Japan	USA	Other	All
Yes	10	26	22	7	65
No	2	1	17	2	22
Total	12	27	39	9	87

Table 7-53. Summary of integration or regression tests conducted across selected countries compared to the entire sample.

Overall, approximately 75% of the projects ran integration tests after code changes. The Indian, Japanese, and other countries' samples have higher percentages of projects that used this practice, with ~83% for the Indian sample and ~96% for the Japanese sample and ~78% for the other countries. The US sample has a lower percentage of ~56% compared to the overall average.

Productivity

There is a wide range of productivity levels in each country, as shown by Figure 7-31 below. The plot does not include outliers.

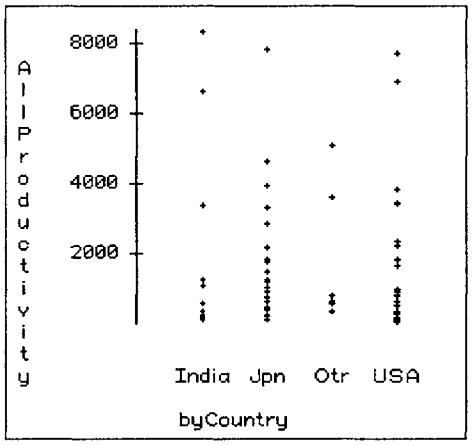

Figure 7-31. Dot plot of productivity (without outliers) broken down by country.

Most of the projects have productivity levels below 2000 LOC/person-month. Each country category has a few extremely productive projects, despite removing the outliers, relative to the group average. Table 7-54 gives a more detailed view of the difference in productivity levels across different countries.

Country	Count	Average	Std Dev	Median
India	11	2010.01	2910.1	531.168
Japan	25	1752.95	1709.02	1204.92
Other	8	1502.84	1811.31	582.387
USA	30	1362.14	1977.45	542.993

Table 7-54. Summary of productivity (without outliers) across selected countries compared to the entire sample.

India has the highest average out of the four country categories. However, taking the median value and Figure 7-31 into account, Indian projects have similar productivity levels as the other projects. With a high average and a high median value, the Japanese projects are the most productive overall out of the four categories.

Bugginess

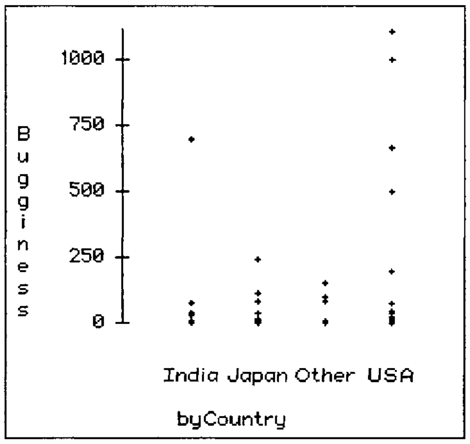

Figure 7-32. Dot plot of bugginess broken down by country.

Most of the projects are around similar levels of bugginess, with the exception of a few extremely buggy projects in India and the USA. Figure 7-32 above shows the quality range across different countries. Table 7-55 lists the numerical qualities of bugginess across the countries.

Country	Count	Average	Adj. Average	Std Dev	Median	Adj. Median
India	6	144.015	32.4677	274.658	37.0833	32.5
Japan	19	29.0032	29.0032	61.2365	2.56312	2.56312
Other	7	53.2098	53.2098	61.3019	13.5135	13.5135
USA	30	127.676	21.2496	293.304	6.77778	4.88095

Table 7-55. Summary of bugginess across selected countries compared to the entire sample. Two averages and medians are provided: one with the outliers and one without the outliers.

The five outliers shown in Figure 7-32 swayed the average measurements for India and the USA by one order of magnitude. With the omission of those extreme data points, the bugginess averages are similar. It is unclear, however, whether the outliers are typical to their respective country.

An interesting observation is that the median and the adjusted median value for India are the highest of the four categories. This does not seem to be consistent with their focus on quality. This may be that the samples in our analysis are not representative of the average projects in India. The analysis without the outliers indicates that Japan and the US are comparable in terms of quality.

8 Conclusion

Current State of Development Practices

The sample in this study is international in scope and diverse in their development strategies. The following is a summary of the current state of software development practices based on the data:

- Design and code reviews are common practice. About 89% of the sample conducted design reviews and approximately 71% of the projects conducted code reviews.
- The majority of the sample, about 63%, developed incrementally using sub-cycles.
- Overall, about 33% of the sample automatically generated code. However, nearly half the non-US sample used code generation. Less than 20% of the US projects generated code.
- Formally documenting specifications is common practice across all groups. About 77% of the sample wrote formal architectural specification documentation, about 91% wrote formal functional or requirements specification documentation, and about 70% wrote formal detailed specification documentation.
- Daily builds is not a common practice with only an average of 25% of the sample conducting builds daily at some time during development.
- About 75% of the sample conducted regression or integration tests after changes to the code base.
- For customer feedback, about 90% of the sample had some form of a prototype and about 72% released betas. Of the projects that released betas, most released one or two betas and about 16% had four or more betas.

Future Considerations

Survey implementers almost always realize potential enhancements to facilitate the data collection process after the survey has gone live. Although the survey from the 2000 HP-

Agilent study was improved upon, there are still areas of improvement such as the following:

Implementation

- The participant's ability to save the survey on his browser so that he can work on it thoroughly before submission. Logins is another method that allows the participants to answer the survey completely before submission.
- Before submission, the answers are checked for accuracy. This will let the participant correct mistakes before submitting his survey and increases the accuracy of the data received. For example, several questions in the survey ask for a series of percentages that sum to 100%. However, it is easy for a participant to mistype his answer. If there is a quick check that alerts the participant of the calculation error, he can correct his answers before the survey is submitted.

Content

- Whether a feasibility test was conducted before the project began. Project teams often evaluate the feasibility of their potential project before officially starting the project. The resources they used for this preliminary assessment should also be taken into consideration when analyzing software projects.
- Duration of development phases such as testing, design, and coding. Participants generally know the time frame of the development phases better than their precise start and end dates.
- Date of the last integration. It was not possible to know how long the integration period lasted without this information.
- Ratings of financial returns and market share change. Since financial information is confidential, many participants did not feel comfortable submitting these numbers. Also, at the time of response, many participants did not have exact numbers regarding market share. Therefore, instead of asking for specific financial and market share data, more participants should be able to rate their financial and market performance. One caution to this method is that the data is the participant's perception and may not accurately reflect the true performance.

- The amount of rework on architectural and functional design. Rework costs the project team both time and money. With this data, an analysis of the factors that contribute to rework would uncover valuable lessons for project teams to avoid rework.

Appendix A: Survey

SOFTWARE DEVELOPMENT PROCESS STUDY

by

MIT Sloan School of Management

Katz Graduate School of Business, University of Pittsburgh

Harvard Business School

This survey has two fundamental objectives:

- To identify and document best-known methods for increasing performance in software development, such as speed, flexibility, and quality
- To identify and understand what types of approaches to software development work best in different types of projects.

As an appreciation for your participation in this survey, you will receive a package of recently published materials about software development.

As an appreciation for your participation in this survey, you will receive a package of recently published materials about software development.

Please note that all project-specific data will be kept confidential by researchers; only summary results and project data that cannot be matched to a specific project will be included in published results.

Contact Information
Academic Contacts

- Prof. Michael Cusumano (MIT Sloan School of Management), cusumano@mit.edu
- Prof. Chris F. Kemerer (Katz Graduate School of Business, University of Pittsburgh), ckemerer@katz.pitt.edu
- Prof. Alan MacCormack (Harvard Business School), amaccormack@hbs.edu

Main Contact (responsible for maintaining the research questionnaire and data collection)

- Pearlin Cheung (MIT School of Engineering), pearl@mit.edu

Some reference material that may be helpful in filling out the survey:

- Project data sheets
- Project schedules
- Project resource plans
- Project results
- Project checkpoint presentations

Company Name	
Name of the project you are describing in this questionnaire (including version number, if any)	
Today's date (e.g. 25 November 2001)	
Name of the person filling out this form	
Your role on the project (e.g., project manager, lead architect, developer, etc.)	
Your email address (in the event that there are questions)	
Your phone number (in the event that there are questions)	
If you wish to be provided with a summary of the results of this research, please indicate that here (select one)	[✓ Yes] No

---

Part 1: Product and Project Description

In this survey you will be answering questions about the main software deliverable from the project.

- A project here is the entire effort devoted toward delivering a specific software deliverable where the activity was both separately managed and tracked from other software deliverables.
- The software deliverable from a project might be a product or a service. In particular, it might be a new release of a previously existing piece of software.

Throughout this survey the focus will generally be on the software *project*, but some questions will ask about the software deliverable, the *product* or *service*. When questions ask about the product or service, they are referring only to the version created by this project.

1.1 Product Description

1.1.1 What type of software is the deliverable? (check one only)

- Systems Software (e.g., OS, DB, network, tools, languages)
- General-Purpose Applications Software(e.g., Office, SAP standard package)
- Custom or Semi-Custom Application (including system integration services)
- Embedded Software

1.1.2 What level of mission-critical reliability does the software deliverable require? (check one only)

- High (e.g., real-time control software or enterprise OS, where system failures are extremely rare, costly, or dangerous)
- Medium (e.g., enterprise software but occasional system failures are acceptable)
- Low (e.g. individual such as Word, Excel)

1.1.3 What was the target hardware platform?

- Mainframe
- Workstation (e.g., Unix, NT)
- PC
- other

1.1.4 To whom is the software *primarily* sold? (check one only)

Individuals	Enterprises	In-House Use

1.1.5 Outline briefly the main functions of the software:

1.1.6 What programming language (e.g. C, C++, HTML, Assembly) was the software primarily written in?	
1.1.7 Please estimate the size of the delivered software in source lines of code:	lines
1.1.8 Does this figure include comments? (select one)	[✓ Yes] No
1.1.9 If "yes," estimate the percentage of comments:	%

1.1.10 What was the origin of the software code in the finished release according to the following categories?

Category	Percentage of Code
Existing code retained from the previous version of this product	%
Existing code taken from other sources (e.g., code libraries)	%
New code developed for this product in other project team(s) (e.g. core code, components, code outsourced to geographically separate organizations)	%
New code developed for this product in this project team	%
[redacted]	[redacted]

1.1.11 Please estimate the percentage of new code developed for this product in this project team (i.e., the figure in box four, above) that was generated automatically. %

1.2 Project Description

Project Budget

1.2.1 What was the software development budget for the project (in $million)? $ million

1.2.2 What was the actual outcome expenditure for the project (in $million)? $ million

Project Team Composition

For the following questions, please refer only to the project team members responsible for the new code identified in box four of table **1.1.10**.

1.2.3 What was the structure of the primary software development team?

Position	Average Staff (number of people)	Peak Staff (number of people)	Total Staff Resources (person-years)
Project Management (includes project managers and directors, but not team or technical leads)			
Architecture and Design			

Development/Programming			
Testing (QA/QE & Integration)			
Project Support (e.g., software engineering, project management support, configuration management, documentation, etc.)			
Other:			

1.2.4 What is the software development experience of the project team (give percentage of team) as described in each category?

Years of Software Development Experience	**Percentage of Software Team**
0-2 years	%
3-5 years	%
6-10 years	%
10+ years	%

1.2.5 What percentage of the development team had worked on the previous version of this product? %

1.2.6 In which country was your project team primarily based?

Project Schedule

1.2.7 What was the original software development schedule (duration in calendar months)? months

1.2.8 What was the actual duration of the project (in calendar months)? months

Please consider the following general phases in a software development project (your organization may not track all these steps, or may use slightly different terminology.)

Functional (or Requirements) Design: Phase that outlines the functional description of the product, which includes product features and market requirements.
Architectural Design: Phase that outlines the high level system design.

Detailed Design and Development: Phase that covers detailed design, coding, unit-level testing, and debugging.
Integration and System Testing: Phase that integrates and stabilizes modules, and tests the performance of the whole system.

1.2.9 Please fill in the dates for the following events on your project in the format MM/YY (e.g. February 2001 is denoted as 2/01). Note: Events do not have to be sequential.

Activity Number	Activity Description	Activity Date
1	Project start date	
2	Functional (or Requirements) design start date	
3	Architecture design start date	
4	Detailed design and development start date	
5	Last addition of new functionality, excluding bug fixes (e.g. code complete)	
6	First system integration date	
7	Project end date	

1.2.10 When was the first prototype of any sort shown to customers (e.g. even if only a mock-up of the user interface)? MM/YY

1.2.11 When was the first beta version released to customers? (A ***beta*** is defined as an early *working* version of the system that is released to selected customers.) MM/YY

1.2.12 How many beta versions did you release to customers?

Part 2: The Development Process

2.1 Planning and Specification

2.1.1 Did the team write an <u>*architectural specification*</u> (i.e. a document that provided a high level description of [✓ Yes] No the subsystems and interfaces of the eventual product or service)? Select one:

2.1.2 If "yes," what percentage of the architectural specification was completed before the team started coding?	%
2.1.3 If "yes," how long were the architectural specifications for this system or product in terms of pages?	pages
2.1.4 If "yes," on which date (MM/YY) was the architecture specification document first available?	MM/YY
2.1.5 If "yes," on which date (MM/YY) was the last major change made to the architectural design specification?	MM/YY

2.1.6 Did the team write a *functional (or requirements) specification* (i.e., a document that described how features worked but not the underlying structure of the code or modules)? Select one: [✓ Yes] No

2.1.7 If "yes," what percentage of the functional (or requirements) specification was completed before the team started coding?	%
2.1.8 If "yes," how long was the functional (or requirements) specification for this system or product in terms of pages?	pages
2.1.9 If "yes," on which date (MM/YY) was the functional (or requirements) specification document first available?	MM/YY
2.1.10 If "yes," on which date (MM/YY) was the last major change made to the functional (or requirements) design specification?	MM/YY

2.1.11 Did the team write a *detailed design specification* (i.e. a document that provides the structure of the modules and an outline of algorithms where needed)? Select one: [✓ Yes] No

2.1.12 If "yes," what percentage of the detailed design specification was completed before the team started coding?	%

2.1.13 If "yes," how long was the detailed design specification for this system or product in terms of pages?	pages
2.1.14 If "yes," on which date (MM/YY) was the detailed design specification document first available?	MM/YY
2.1.15 If "yes," on which date (MM/YY) was the last major change made to the detailed design specification?	MM/YY
2.1.16 What percentage of the features in the *original* functional (or requirements) specification were contained in the final product?	%
2.1.17 What percentage of the features in the final product were contained in the *original* functional (or requirements) specification?	%

2.2 Project Structure

Sub-cycles

The following questions ask about sub-cycles, in contrast to code builds. A sub-cycle typically includes the following phases: design, implementation, build, testing, integration, and release (internal or external). For example, we consider the pure waterfall method to be one sub-cycle. A 12-month project that is divided into four three-month sub-projects or phases, each ending with a formal release at least within the company, would contain four sub-cycles.

2.2.1 Did you divide the development phase of the project into separate development sub-cycles that built, tested, and released a *subset* of the final product's functionality?	[✓ Yes] No
2.2.2 If "yes," how many separate development sub-cycles were there on this project?	
2.2.3 If "yes," how long was the average sub-cycle, in terms of weeks?	weeks
2.2.4 If "yes," after which sub-cycle was the first beta version released?	

Project Events

2.2.5 Estimate the percentage of the final product's functionality which existed in the design at the following project events (assume the functionality in the design is 0% at the start of the project and 100% at the time the product is launched):

Project Event	**Percentage of Final Product Functionality**
The first prototype shown to customers (even if only a mock-up)	%
The first system integration (even if modules only partially complete)	%
The first beta version (the initial full version for external customer use)	%

2.3 Detailed Development Practices and Tools

Coding Practices

2.3.1 Was each developer paired up with a tester who would test their code (i.e., a "private release") prior to checking in to the master build? [✓ Yes] No

2.3.2 Did the project team engage in "pair programming" where developers work in pairs such that one developer codes or tests while the other reviews? [✓ Yes] No

2.3.3 Was the code *collectively* owned (i.e., developers can modify any part of the code), *selectively* owned (i.e., developers can alter only a subset of the code such as specific modules), or *individually* owned by the developer who wrote it (e.g., developers are assigned responsibility to certain classes)?

[✓ Collective]
Selective
Individual

2.3.4 Did the project team follow a uniform coding style or a set of coding rules? [✓ Yes] No

2.3.5 If "yes," please name or briefly describe this standard.

Reviews

2.3.6 Were there any design reviews done? [✓ Yes] No

2.3.7 If "yes," how many?

2.3.8 Were there any code reviews done? [✓ Yes] No

2.3.9 If "yes," approximately what percentage of all the newly developed code was reviewed by another team member? %

System Builds

2.3.10 *During the development phase*, how frequently was the system "built" on average (i.e., how often were design changes, including bug fixes, integrated into the code base and then recompiled)?

	Daily	**2-3x a week**	**Weekly**	**Bi-Monthly**	**Monthly or less**
First third of development					
Middle third of development					
Last third of development					

Testing

2.3.11 Did developers help write test cases? [✓ Yes] No

2.3.12 If "yes," what percentage of the code did the developers test? %

2.3.13 Was any type of integration or regression test (as opposed to a simple compile and link test) run each time developers checked changed or new code into the system build? [✓ Yes] No

2.3.14 If "yes," how long did the integration test usually take to run? hours

2.3.15 When the product was "built," how long did it take to get feedback on the performance of the system using the most comprehensive set of system tests assembled during the project? hours

2.3.16 Approximately what percentage of the test cases run on the product or system were automated? %

Tools

2.3.17 What were the most useful development tools or methods for each activity?

Requirements Design	
Architectural Design	
Detailed Design	
Coding	
Testing	
Configuration Management	
Project Management	
Other	

Part 3: Performance

The following questions refer to data on sales, market share, and reported bugs for the 12 month period following the product's launch. If less than 12 months of data are available, please report the number of months where indicated.

3.1 Financial Performance

3.1.1 If you sold your product in the market, please estimate the total dollar revenues that the product generated in the first 12 months after shipment of the final release, including extra charges for design changes, if applicable. If your product included charges for hardware, please estimate *revenues solely attributable to the software part* of the product (for example, as tracked by your internal accounting procedures).

$ million

Note: If less than 12 months of data are available, please report data only for those months you have data and note the number of months here:

months

3.2 Market Performance

3.2.1 If you sold your product in the market, please estimate the increase or decrease in market or user share of your product in the first 12 months after shipment of the final release (e.g. if your share increased from 10% to 20%, this is a 10% increase):

% points

Note: If less than 12 months of data are available, please report data only for those months you have data and note the number of months here: months

3.3 Product Quality

Software quality is often thought of as the relative absence of defects, or 'bugs'. Most organizations have mechanisms in place for testers and customers to report bugs, (e.g. 'software problem reports'). The following questions ask about these bugs in terms of volume and timing.

3.3.1 Please estimate the number of bugs reported by customers or by field technicians in the first 12 months after the system was shipped: bugs

Note: If less than 12 months of data are available, please report data only for those months you have data and note the number of months here: months

If there was a beta release, please answer the following question:
3.3.2 Of the bugs discovered AFTER the first beta release, please estimate the percentage of these bugs that came from the following sources. (Numbers should sum to 100%)

Source	**Percentage of Bugs Found After Beta**
Bugs found by development engineers themselves	%
Bugs found by QA and test engineers during testing activities	%
Bugs found by customers using the beta release	%

Please answer the following questions regarding project expectations using a 5-point scale, where 1=Significantly below, 2=Below, 3=Met expectations, 4=Above, 5=Significantly above

3.3.3 Please indicate the extent to which you perceive the project met expectations in terms of customer satisfaction with the end-product.

3.3.4 Please indicate the extent to which the product met expectations in terms of sales, relative to the forecasts which were made **prior** to the product's release.

Please note below the number of any questions for which you have low confidence in your answers.
(E.g., My numbers are good except for 2.2.5, which I estimated.)

Many thanks for completing this survey. We will email you with a link to the results of this survey as soon as they have been compiled. If you have any comments on the survey, or suggestions for future questions we should ask, please leave them below.

Please provide your mailing address below.

References

[1] Aaen, Ivan, Peter Bøtcher and Lars Mathiassen. "The Software Factory: Contributions and Illusions." *Proceedings of the Twentieth Information Systems Research Seminar*. Scandinavia, Oslo, 1997: 411-413.

[2] Anthes, Gary H. and Jaikumar Vijayan. "Lessons from India Inc." *Computerworld*. April 2, 2001. <http://www.itworld.com/Tech/2418/CWD010402STO59083/>.

[3] Blackburn, J.D.; Scudder, G.D.; Van Wassenhove, L.N. "Improving speed and productivity of software development: a global survey of software developers." *IEEE Transactions on Software Engineering*. December 1996: 875 –885.

[4] Boehm, Barry W. "Industrial Software Metrics Top 10 List." *IEEE Computer*. September: 43-57.

[5] Boehm, Barry W. "A Spiral Model of Software Development and Enhancement." *IEEE Computer*. May 1988: 61-72.

[6] Boehm, Barry W., et al. "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0." *Annals of Software Engineering, Special Volume on Software Process and Product Measurement*. J.D. Arthur and S.M. Henry, eds. Amsterdam: J.C. Baltzer AG, Science Publishers.

[7] Boehm, Barry W., et al. *COCOMO II Model Definition Model*. University of Southern California. Version 1.4. <http://my.raex.com/FC/B1/phess/coco/Modelman.pdf>.

[8] Cusumano, Michael A. *Japan's Software Factories: A Challenge to U.S. Management*. Oxford University Press. 1991.

[9] Cusumano, Michael A. and Chris F. Kemerer. "A Quantitative Analysis of U.S. and Japanese Practice and Performance in Software Development." *Management Science*. November 1990: 1384-1406.

[10] Cusumano, Michael A. and Richard W. Selby. *Microsoft Secrets*. New York: Simon & Schuster. 1998.

[11] Dunn, Robert H. *Software Defect Removal*. New York: McGraw-Hill. 1984.

[12] Duvall, Lorraine M. "A Study of Software Management: The State of Practice in the United States and Japan." *The Journal of Systems and Software*. January 1993. <http://www.dacs.dtic.mil/techs/management/node2.html>.

[13] Highsmith, Jim. "Extreme Programming." *e-business Application Delivery*. February 2000. <http://www.cutter.com/ead/ead0002.html>.

[14] HL Global Parters, L.L.C. "Why Offshore Development?" <http://www.hlglobalpartners.com/WhyOS.htm>.

[15] Humphrey, Watts S. "Characterizing the Software Process: A Maturity Framework." *IEEE Software*. March 1998: 73-79.

[16] Jones, Capers. *Assessment and Control of Software Risks*. Englewood Cliffs, N.J.: Yourdon Press. 1994.

[17] Jones, Capers. "Programming Languages Table." Software Productivity Research, Inc. March 1996. <http://www.theadvisors.com/langcomparison.htm>.

[18] Keuffel, Warren. "A Software Superpower?" *Software Development*. March 2001. < http://www.sdmagazine.com/documents/s=733/sdm0103o/0103o.htm>.

[19] Lateef, Asma. "Linking up with the global economy: A case study of the Bangalore software industry." *International Labour Organization Discussion Papers*. No. 96. 1997. <http://www.ilo.org/public/english/bureau/inst/papers/1997/dp96/ch2.htm>.

[20] Linger, Richard C. "Cleanroom Process Model." *IEEE Software*. March 1994: 50-58.

[21] Matsubara, Tomoo. "Japan: A Huge IT Consumption Market." *IEEE Software*. Sep/Oct 2001. <http://www.computer.org/software/homepage/2001/05CountryReport/>.

[22] Maxwell, K.D.; Van Wassenhove, L.; Dutta, S. "Software development productivity of European space, military, and industrial applications." *IEEE Transactions on Software Engineering*. October 1996: 706 –718.

[23] McConnell, Steve. *Rapid Development*. Redmond, WA: Microsoft Press. 1996.

[24] Myers, Glenford J. *The Art of Software Testing*. New York: John Wiley & Sons. 1979.

[25] Paulk, Mark. "List of Maturity Level 4 and 5 Organizations." Jan 2002. <http://www.sie.cmu.edu/cmm/high-maturity/HighMatOrgs.pdf>.

[26] Royce, Winston W. "Managing the Development of Large Software Systems: Concepts and Techniques." *Proceedings of IEEE WESCON*. 1970.

[27] Spangler, Alan. "Cleanroom Software Engineering." *IEEE Potentials*. October/November 1996: 29-32.

[28] Standard &Poor. *Industry Surveys, Computers: Software*. Vol. 170. No. 4. January 24, 2002.

[29] Steinmueller, W. Edward. "The U.S. Software Industry: An Analysis and Interpretive History." 1995. <http://www-edocs.unimaas.nl/files/mer95009.pdf>.

[30] Upadhyayula, Sharma. "Rapid and Flexible Product Development." Master of Science thesis in the MIT System Design and Management Program. MIT, 2001.

[31] Vosburgh, J.B., et al. "Productivity Factors and Programming Environments." *Proceedings of the 7th International Conference on Software Engineering*. Los Alamitos, CA: IEEE Computer Society. 1984: 143-152.

[32] Yourdon, Edward. *Structured Walk-Throughs*, 4th ed. New York: Yourdon Press. 1989.