December 2006

On The Value of Code Inspections for Software Project Management: An Empirical Analysis

Narayan Ramasubbu
Singapore Management University

Ramanath Subramanyam
University of Illinois

Sunil Mithas
University of Maryland

M.S. Krishnan
University of Michigan

On The Value of Code Inspections for Software Project Management: An Empirical Analysis

Narayan Ramasubbu
Singapore Management University
nramasub@smu.edu.sg

Ramanath Subramanyam
University of Illinois, Urbana-Champaign
rsubrama@uiuc.edu

Sunil Mithas
University of Maryland, College Park
smithas@rhsmith.umd.edu

M.S. Krishnan
University of Michigan
mskrish@bus.umich.edu

ABSTRACT

Code inspections continue to gain significance as a software verification scheme since Fagan introduced the concept. Software engineering researchers examining the value of code inspections have exclusively focused on defect removal benefits of inspections. In this paper we develop and test empirical models of both quality improvement and project management benefits realized because of effort spent on code inspections. We analyze data collected on 40 real world projects from a leading software corporation to provide rigorous empirical evidence for the value of code inspections. We find evidence for hitherto unexplored hypothesis that improved understanding gained during code inspection has project management benefit of better test planning and control that could eventually avoid project overruns. We provide a research framework that takes into account the sequential characteristics of waterfall software development model and the effects of rework generated by verification schemes to answer an important research question on the value of code inspections for project management.

Keywords

Code Inspections, Software Engineering, Software Testing, Software Quality, Estimation, Life Cycle, Process Metrics, Project Management

INTRODUCTION

Code inspection is a structured quality verification process to identify software defects by cognitive analysis of software code. Unlike other software verification schemes such as unit, integration or system testing, code inspection is a static verification process in the sense that software code is not executed but reviewed like any other document. The idea of examining software code in an organized manner to reduce defect rates was first published by Fagan (1976). Since then software inspection process has spurred lot of interest among both academic researchers and practitioners.

Past software inspection studies may be broadly classified as studies that highlighted the benefits of inspections (Ackerman et al., 1989, Russell, 1991, Weller, 1993), studies that compared different inspection methods (Hetzel, 1976, Myers, 1978, Basili and Selby, 1987, Wood et al., 1997); studies that posited different strategies for efficient inspection (Bisant and Lyle, 1989, Chaar et al., 1993, Rodgers and Dean, 1999); studies that explained the determinants of inspection effectiveness (Porter and Votta, 1997, Petersson, 2001, Emam et al., 2000) and studies that evaluated the cost effectiveness of defect removal through inspections (Collofello and Woodfield, 1989, Kusumoto et al., 1991, Kusumoto et al., 1992, Porter et al., 1997, Bianchi et al., 2001, Briand et al., 2000, Briand et al., 1998). Several researchers have also offered detailed inspection literature surveys (Fagan, 1986, Humphrey, 1989, Porter et al., 1996, Laitenberger, 2002).

In this paper we develop and test empirical models of quality improvement and test planning benefits of software inspections. Our quality improvement model evaluates the efficacy of software verification schemes such as code inspections and software testing in reducing in-process and delivered defects. We also consider the effect of rework induced by software verification schemes on software defects at various stages of project life cycle providing a nuanced understanding of the effect of verification schemes on final defects. In addition, we posit and test an important and hitherto unexplored project management benefit resulting from code inspections: test-planning accuracy. We empirically test the hypothesis that better

understanding of the software code and interrelationships among its various modules attributable to code inspection leads to improved estimation of resources needed for the testing stage. In turn, improved accuracy of estimation process may allow firms to prepare in advance for potential budget or time overruns.

This paper makes three main contributions. First, this paper considers business value benefits from software inspection process while explicitly accounting for the effect of rework and other complimentary processes such as software testing. This research approach allows us to compare the benefits and limitations of software inspections and software testing. Software managers can use the insights gained from this analytical approach in their choice of defect removal tools. Second, we consider the effects of software testing on both *in-process* and *final* software quality providing a more complete understanding of the effect of software verification processes on software quality. Finally, in addition to defect removal benefit from code inspections, we also highlight the beneficial effect of code inspections on test planning accuracy that may help managers in estimating their resource requirements more precisely and thus avoiding potential schedule or budget over-runs.

THEORY AND HYPOTHESIS

In order to accurately measure the value of inspections, researchers have focused on metrics used for evaluating software inspection effectiveness. Early on, Fagan proposed inspection effectiveness metric as the proportion of defects identified in the inspection of an artifact to the total number of defects discovered in the artifact throughout the software life cycle. However, this metric does not account for the effort spent on inspections and thereby does not provide a cost-benefit view of inspections. To overcome this weakness of Fagan's metric, Collofello and Woodfield (1989) proposed cost-effectiveness metric as a measure of the worth of an inspection. Collofello measures worthiness of an inspection as the ratio of testing cost saved by code inspection to the total cost of inspection. Kusumoto (1991, 1992) refined Collofello's metric by accommodating a 'potential testing cost' factor that adjusts for the amount of testing cost that would be needed in the absence of any inspections

There are two limitations of a purely *metrics-based* approach for evaluating the benefits of code inspection. First, code inspection is one out of many software verification schemes and different verification schemes vary in their ability to identify different types of software defects. By assuming that testing could identify all defects, the inspection effectiveness metrics reviewed above tend to underestimate the actual benefits of inspection. Second, calculating 'potential testing cost' or the cost of testing in the absence of inspections may be problematic. It is difficult to reliably estimate the increase in number of defects in the absence of inspection and the cost associated with identifying such defects during testing. Although academic researchers have tried to overcome these difficulties by using experimental and ex-post simulation techniques (Briand et al., 1998, Emam et al., 2000), it may not be possible for software managers to routinely and easily assess value of specific verification schemes using such experimental and simulation techniques. This may be because commercial projects exhibit much more complexity as compared to laboratory experiments making it difficult to directly apply techniques from experimental projects. Also, deriving simulation parameters tailored to specific project scenarios is not straightforward and specific simulation designs for different development scenarios and processes are not readily available for practitioners.

In this paper we develop and test a *process based* empirical framework that evaluates quality improvement and project management benefits of code inspection by considering the processes involved in software development and verification. In this process based framework we account for the effect of rework generated by verification schemes to identify the benefits as well as limitations of individual verification schemes used in software projects. Also, our framework distinguishes the effect of inspections from the effect of other verification schemes such as software testing. We also make a conceptual distinction between *in-process* software quality and *final* software quality. While *in-process software quality* concerns number of defects identified in the code artifact before the software is delivered to the customer, *final software quality* relates to the number of software defects reported by customer after the software has been delivered to the customer. This conceptual distinction between two dimensions of software quality allows us to explain observed empirical relationships between software verification effort and software quality.

Figure 1 depicts our research model. We next discuss the theory and reasoning in support of our proposed research model.

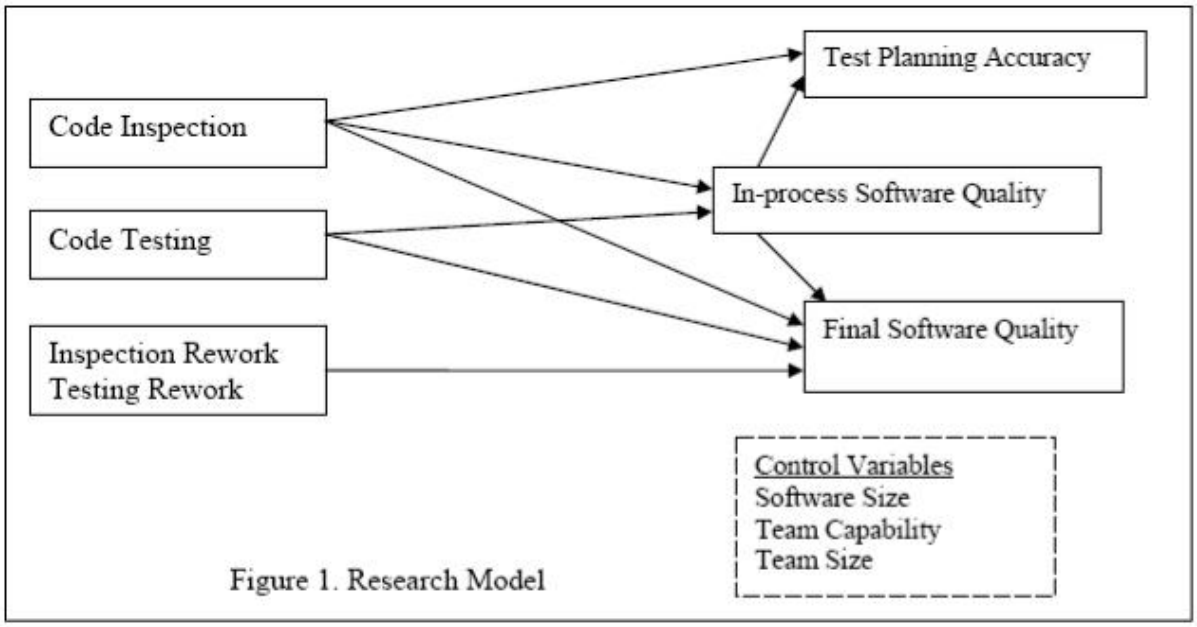

Figure 1. Research Model

Software Quality Model

Despite mixed findings on benefits and cost effectiveness of individual verification schemes in the academic research, there is a consensus among software managers that verification schemes in general improve software quality (Jones, 1997). The dominant logic in arriving at this consensus has been that when software artifacts are verified for their correctness, structural, modularity and descriptive properties before delivery, the probability that these attributes of software will contribute to a production failure decreases (Dromey, 1995). Several software quality initiatives such as ISO 9001, SEI-CMM have made verification schemes mandatory to achieve higher process maturity levels endorsing the view that verification schemes improve software quality. Based on these we propose our initial set of hypotheses:

H1: Code inspection effort is positively associated with final software quality.

H2: Code testing effort is positively associated with final software quality.

As noted earlier, while software engineering researchers conceptually agree on the value of software verification schemes in general, the real debate is about the relative benefits of individual static schemes such as software inspections vis-à-vis code testing. To understand the benefits and limitations of individual verification schemes we propose a *process-based* framework. This framework, while accounting for the direct effect of code inspection and testing effort on final software quality, also considers the effect of software correction process triggered by the verification schemes. These correction processes involve reworking software code after analyzing the defect reports from individual verification schemes such as inspection and testing activities. The nature of correction processes and effort spent on associated rework depends on the software development model followed in a firm. In this paper we focus on the waterfall model of software development.

Due to sequential nature of activities in the waterfall model of development, rework done on software code to correct the defects identified during different verification schemes undergo different levels of further quality checks. Defects identified during inspection lead to reworking of software code and new test cases are written to enhance test coverage of the modified code. Also, code walkthrough conducted during inspections helps developers in identifying structural, modularity and integration related defects that can be corrected during rework without further root-cause analysis. Moreover, during the inspection report analysis, if any proposed rework causes functionality and scope creep, managers have the discretion to drop such activities and focus on current requirements. Thus rework after inspections may be more time effective and focused on customer specifications and we expect that inspection triggered rework has a positive impact on final software quality.

In contrast to rework phase after inspections, time available for rework phase after testing is often limited because of pre determined delivery schedules. In addition, because of schedule pressure during the final stages of development, programmers may not spend adequate time on root cause analysis for defects that are identified during testing. This often leads to quick fixes and postponing thorough analysis for future service packs and maintenance activities after delivery. Thus even though testing might be an effective defect identification process, its impact on reducing delivered defects might be

offset because of the high defect prone nature of rework process conducted after testing. To test this process based framework for evaluating inspection effectiveness in waterfall model of software development, we propose the following set of hypotheses:

H1a: Code inspection effort is positively associated with in-process software quality.

H1b: Rework done after code inspection is positively associated with final software quality.

H2a: Code testing effort is positively associated with in-process software quality.

H2b: Rework done after code testing is negatively associated with final software quality.

Test Planning Accuracy Model

Importance of code inspection process in contributing to better software project management through better estimation has not been fully explored in previous inspection research. In this section we model this planning benefit to present a complementary view of the value of code inspections.

Estimating resource allocation for downstream software development activities like testing is difficult because reliable estimates of software size are not available at the planning stage. Managers, therefore, have to work with their initial estimates based on previous experience with similar projects. The key to reliably estimate resources needed for downstream activities is to review and update the initial estimation as soon as more information about actual code size becomes available. Since code inspections are carried out once the preliminary source code has been developed, it is possible to compare initial estimates with the actual software code at this stage. Code inspection process, thus, provides an opportunity mid-way in the software development life cycle to improve ex-ante estimations. An improved understanding of the processing logic by team members and the availability of actual size of software code may enable a better assessment of the time and resources needed for the next sequential stage in software lifecycle (i.e. software testing). Effort spent on code inspections could help team members to better understand the interconnections between different modules to design appropriate test data, test plans and to estimate the actual amount of test coverage needed. Software development managers may use such enhanced understanding of the software code and project parameters to reassess their testing strategy, renegotiate contractual deadlines or reformulate their personnel allocation strategy to avoid schedule overruns. Thus our next hypothesis is:

H3: Code inspection effort is positively associated with enhanced test planning accuracy.

One way to evaluate the test-planning benefit of the code inspections effort is to assess the accuracy of software test estimation done after code inspections. If managers gain beneficial information because of the inspection process then their estimates on the effort needed for testing should be more accurate and schedule overruns be minimized. The difference between actual testing effort and the estimated effort for testing calculated after inspections can indicate if code inspection effort contributes to test planning accuracy.

RESEARCH SITE AND ANALYSIS

Research Site

We collected data from the information technology and consulting services division of a leading software company that employs over seventeen thousand people world-wide and has annual revenue of about seven hundred and fifty million dollars. We collected planning, effort and quality related data on each of these projects from the central quality services of the division. Personnel related data was obtained from individual project managers who had tracked such data using a centralized, web based resource allocation and tracking tool. The quality department had audited all of these projects for CMM level five compliance and certified the integrity of project data.

Variables

Code Inspection Effort: This is measured as the number of person hours spent on all code inspections conducted during the coding stage of the project.

Testing Effort: This variable is measured as the total number of person hours spent on software testing before delivery of the project to the customer.

Software Size: We measure software size in terms of Function Points (IFPUG, 1999). Function points measurement of code size indicates the magnitude of application functionality as well as the complexity of implementation involved in those functionalities (Dreger, 1989).

Software Quality: Software quality is measured in terms of the defect density of the software. We explicitly classify defects into two categories of *in-process* and *delivered* defects. Defects identified by internal verification schemes such as inspections and testing are *'in-process defects'* and defects reported by the customer after the delivery of the project are *'delivered defects'*. Delivered defects are the errors that went unnoticed by the verification schemes employed by the project team. Using these counts of in-process and external defects, we calculate the defect density of software before and after delivery. Defect density before delivery is the *in-process software quality or* software quality as seen internally by the development team. Defect density of software after delivery of the project is the *final software quality* or software quality as reported by the customer.

Team Capability: Software development teams with capable team members are likely to produce greater output in terms of software code (measured in function points). Hence, we measure team capability in this research as the average amount of software code (in function points) produced by members of the software development team per person per month. Since software programmers may be part of several software teams for different projects, this measure was obtained form the Human Resources division of the software firm because HR managers collected this data in measuring performance across teams and for use in planning team based incentives.

Team Size is the count of number of personnel involved in the project.

Test Planning Accuracy: We calculate test planning accuracy using the *magnitude of relative error* (MRE) measure originally used by Kemerer (1987):

MRE = (Actual testing hours – estimated testing hours) / Actual testing hours

Higher MRE signifies that the deviance of the estimate from the actual is higher and hence test-planning accuracy is lower.

Test Rework: Effort spent in person hours to fix errors identified during software testing.

Inspection Rework: Effort spent in person hours on reworking software code to rectify errors and incorporate comments received from the code inspection reviewers before proceeding to the next stage (i.e. testing).

Table 1 presents the summary statistics for the above variables. Table 2 reports the correlations among variables.

Table 1. Summary Statistics

Variable	Mean	Std. Dev.	Min	Max
Code Size	2280.0	2973.8	33.0	18247.0
Team Size	11.4	6.5	2.0	30.0
Team Capability	48.8	28.7	3.1	118.9
Inspection Hours	15.7	38.3	0.0	175.3
Inspection Rework	800.5	819.8	121.0	3910.0
Testing Hours	82.0	65.9	3.0	234.0
Estimated Testing Hours	343.6	301.8	11.0	1194.0
Testing Rework	366.2	407.2	48.0	2416.0
Delivered Defect Density	404.5	726.5	20.8	3306.0
Internal Defect Density	8.3	11.5	0.7	65.4
Estimation Error(MRE)	7.6	14.2	0.1	83.5

Table 2. Correlation among Variables

		1	2	3	4	5	6	7	8	9	10
1	Delivered Defect Density	1.00									
2	Internal Defect Density	0.29	1.00								
3	Inspection Hours	0.15	-0.07	1.00							
4	Testing Hours	-0.41	-0.71	0.22	1.00						
5	Team Capability	0.41	0.60	-0.24	-0.52	1.00					
6	Code Size	0.24	0.71	0.11	-0.35	0.70	1.00				
7	Team Size	0.16	0.25	0.35	0.13	0.07	0.61	1.00			
8	Testing Rework	-0.43	-0.65	0.15	0.85	-0.44	-0.34	0.09	1.00		
9	Inspection Rework	0.04	0.46	0.20	-0.04	0.24	0.74	0.69	-0.01	1.00	
10	Estimation Error(MRE)	0.10	0.52	-0.29	-0.59	0.09	0.18	0.06	-0.58	0.05	1.00

Data Analysis

Based on our discussion in section 2.1, the empirical model to test the quality improvement benefit of code inspection is shown in equation 1.

Final Software Quality (Delivered defect density) = F (Code Inspection Effort, Testing Effort, Team Capability, Code Size, Team Size, In-Process Quality) ……. *Equation (1)*

Past research indicates that the effects of size, effort and cycle time on software quality are not linear because of the presence of scale economies in software development (Banker and Kemerer, 1989). Consistent with this approach we use a log linear model of quality improvement as shown in equation (2).

$$Log(\text{Delivered Defect Density}) = \beta 0 + \beta 1 * Log(\text{Code Inspection Effort}) + \beta 2 * Log(\text{Testing Effort}) + \beta 3 * Log(\text{Team Capability}) + \beta 4 * Log(\text{Code Size}) + \beta 5 * Log(\text{Team Size}) + \beta 8 * Log(\text{In-Process Quality}) + \varepsilon \quad \text{…….. Equation (2)}$$

In order to empirically validate the defect identification effects of testing and inspection effort, and to confirm the impact of rework done after inspection and testing on delivered defects, we propose the log-linear multiplicative models shown in equations 3 and 4.

$$Log(\text{Internal Defect Density}) = \beta 0 + \beta 1 * log(\text{Inspection hours}) + \beta 2 * log(\text{testing hours}) + \beta 3 * log(\text{team capability}) + \beta 4 * log(\text{Code Size}) + \beta 5 * log(\text{team size}) + \delta \quad \text{…….Equation (3)}$$

$$Log(\text{Delivered Defect Density}) = \beta 0 + \beta 1 * log(\text{Inspection hours}) + \beta 2 * log(\text{testing hours}) + \beta 3 * log(\text{Team capability}) + \beta 4 * log(\text{Code Size}) + \beta 5 * log(\text{Team Size}) + \beta 6 * log(\text{Testing Rework}) + \beta 7 * log(\text{Inspection Rework}) + \beta 8 * log(\text{In- Process Quality}) + \zeta \quad \text{…….Equation (4)}$$

Based on our discussion in section 2.2, the test planning accuracy model is shown in its multiplicative specification in equation 5.

$$Log(\text{Test Planning Accuracy}) = \beta 0 + \beta 1 * Log(\text{Inspection Effort}) + \beta 3 * Log(\text{Team Capability}) + \beta 4 * Log(\text{Code Size}) + \beta 5 * log(\text{Team Size}) + \beta 8 * Log(\text{Internal Defect Density}) + \varepsilon \quad \text{….Equation (5)}$$

Equations 2-5 are estimated using classical linear regression. All models were subject to empirical tests of normality, multicollinearity, heteroskedasticity and the effect of outliers and we did not notice any violations. The results of our estimations are presented in Table 3.

RESULTS AND DISCUSSION

Table 3. Results for Software Quality and Test Planning Accuracy Models
(p values are in parentheses)

Models→		(1)	(2)	(3)	(4)
Dependent Variable→		Final Software Quality	In-process Software Quality	Final Software Quality	Test Planning Accuracy (MRE)
Inspection Hours	β_1	0.326**	-0.026	0.328**	-0.388**
		(0.017)	(0.716)	(0.016)	(0.020)
Test Hours	β_2	-0.609**	-0.569***	-0.216	---
		(0.032)	(0.000)	(0.553)	---
Team Capability	β_3	1.521***	-0.193	1.699***	-0.838*
		(0.001)	(0.396)	(0.001)	(0.099)
Code Size	β_4	-1.153***	0.644***	-1.383***	-0.183
		(0.009)	(0.003)	(0.009)	(0.717)
Team Size	β_5	1.389***	-0.128	1.407***	0.213
		(0.007)	(0.632)	(0.006)	(0.714)
Testing Rework	β_6	---	---	-0.605*	---
		---	---	(0.094)	---
Inspection Rework	β_7	---	---	0.227	---
		---	---	(0.467)	---
In-process Software Quality	β_8	-0.028	---	-0.122	1.241***
		(0.928)	---	(0.696)	(0.000)
Constant	β_0	6.714***	0.264	8.16***	3.613**
		(0.000)	(0.783)	(0.000)	(0.036)
Observations		40	40	40	40
R-squared (Adjusted)		0.379	0.711	0.399	0.394
F		4.96***	20.23***	4.23***	6.07***
		(0.001)	(0.000)	(0.002)	(0.000)

* significant at 10%; ** significant at 5%; *** significant at 1%

Results: Software Quality

Model 1 in Table 3 presents the results of estimation of our quality improvement model. We find that effort spent on code inspection has a positive and significant impact on final software quality ($\beta1$=0.326, p=0.017). This means that holding all else constant, a one percent increase in effort spent on inspections leads to 0.33% increase in final software quality. This result establishes the value of code inspections in improving final software quality.

Empirical result for the effect of testing effort on final software quality is surprising. We had expected that an increase in testing effort would improve final software quality. However, our analysis indicates that one percent increase in effort spent on testing decreases final software quality by 0.60% ($\beta2$=-0.609, P=0.032). To further analyze the effects of testing verification scheme, we refer to our second and third models. In the second model we test for the ability of inspection and testing in capturing defects before delivery. In the third model, we account for sequential nature of software development processes by considering the effect of code inspections and testing on in-process software quality associated rework. The results of these analyses are presented in Models 2 and 3 of Table 3 and discussed below.

As Model 2 in Table 3 shows, effort spent on testing does help in capturing more internal defects before delivery. Specifically in this sample our analysis shows that a one percent increase in testing effort leads to a decrease in internal defect density by 0.57% ($\beta2$=-0.569, P=0.000). It is useful to note that a *decrease* in internal defect density means an *increase* in the

number of in- process defects identified before delivery. However it is important to analyze why even though testing enables developers to identify more internal defects, it does not eventually lead to an improvement in final software quality in this sample (i.e., decrease in defects after delivery). We had hypothesized that this may be primarily because of error proneness of rework effort spent on fixing defects identified during black box testing. We find empirical support for this assertion. Specifically, our analysis indicates that a one percent increase in effort spent on rework after testing deteriorates final software quality by 0.61% (see Model 3 in Table 3, $\beta 6$=-0.605, P=0.09). In contrast to testing verification scheme, code inspections do not significantly contribute to in- process defects (refer model 2 in table 3, $\beta 1$=-0.026, P=0.716) and any rework done after code inspections does not deteriorate final software quality (refer model 3 in table 3, $\beta 7$=0.227, P=0.467).

Our analysis highlights the benefits and limitations of two software verification schemes i.e. code inspection and testing. While effort spent on code inspections may not be the most effective way to identify more internal defects, it does have a positive and significant impact on improving final software quality. On the other hand testing is very effective in identifying in- process defects but might have a deteriorating effect on final software quality once we consider the rework effects arising from sequential nature of waterfall model of development. These relationships highlight the tradeoffs involved in the choice of software verification schemes. Software development managers thus have to judiciously choose their verification strategy by balancing effort spent on inspections and testing.

Results: Test Planning Accuracy

Model 4 in Table 3 presents the results of test planning accuracy model. Our regression results show that effort spent on inspection has a significant and positive impact on increasing the accuracy of test estimations developed after conducting code inspection process. A one percent increase in the effort spent on code inspection decreases the magnitude of estimation error by 0.39% ($\beta 1$=-0.388, P=0.02). This shows that managers could use the improved understanding gained during inspections to choose their testing resources appropriately and possibly prevent slippage and schedule overruns. This result has important implications for both practitioners and researchers. Academic researchers and practicing managers tend to overemphasize the cost effectiveness of verification schemes solely based on the defect capturing capacity of a particular verification scheme. Since code inspections could also be a source of other benefits that improve overall project performance, focusing on defect capture alone may not provide a fair comparison among verification schemes. Researchers concerned with inspection cost effectiveness metrics have also focused explicitly on quantifying the defect removal benefits realized through inspections. Without the inclusion of the entire set of project management benefits realized by effort spent on inspections, these metrics tend to underestimate the business value of code inspections. Researchers should be cautious in comparing different verification schemes on a defect removal based cost effectiveness metrics as each of these schemes have unique advantages and limitations.

The effects of control variables in the test planning accuracy model are as expected. Our results indicate that the estimation error decreases with respect to team capability ($\beta 3$=-0.838, P=0.099). We also find that estimating required testing time for an artifact with larger levels of defect density is more difficult. In our sample, one percent increase in defect density increases estimation error by 1.24% ($\beta 8$=1.241, P=0.000). One way to mitigate this effect is to gauge development quality level using code inspections before testing and revisit original testing plans to assess their relevance and assess the required test coverage. As Dijkstra (1971) noted, 'any sophisticated testing can never prove the absence of bugs but only their presence'. Thus it is not possible for a manager to dynamically decide the required levels of test coverage and testing time using results from past testing experience alone. Our analysis indicates that the estimates for testing parameters when validated and corrected using experience through complimentary verification schemes such as code inspections leads to improved estimation accuracy.

CONCLUSION

Past research analyzing the value of code inspections has presented a paradox. While experimental studies and theoretical intuition adequately demonstrate the value of inspections, there has been mixed evidence from empirical studies. Researchers attempting to demystify this paradox have encountered difficulties in using currently available inspection metrics for their empirical analysis. In this research, we developed a process-based empirical framework that overcomes this difficulty. We use software lifecycle data collected at a fine granular level to assess the limitations and benefits of code inspections and testing. Also our empirical analysis goes beyond the defect removal framework to demonstrate project management benefits realized because of code inspections.

Our analysis highlights the trade offs present in the choice of verification schemes for a project. We showed that the choice of verification scheme solely based on defect removal may not be judicious and overall improvement in project performance can be achieved by balancing the effort spent on testing and other complementary verification schemes such as code inspections. One reason for the realization of benefits by balancing resource allocation to multiple verification schemes could be because of the sequential nature of activities in waterfall model of software development. We show that in such a sequential mode of development, rework effects in later stage of lifecycle might offset the benefits realized through higher defect identification capabilities of testing verification scheme.

There are several limitations of this research that may be overcome in future. First, we have focused on the value of code inspection in waterfall mode of software development. Though we are confident of the usefulness of our empirical framework in other software development models, generalization of our results to other models of software development such as extreme programming needs further examination. Second, we have limited our analysis to inspections done at the coding stage. While our empirical framework may be applicable to analyze value of inspections in requirements and design stages, we have not validated our empirical model with data specific to these stages. Future research could explore these. Future inspection studies could also analyze the effectiveness of inspections at different levels of code structure, logical and data complexities. The benefits and limitations of different verification schemes will become clearer with such an analysis. Even after three decades of research on software inspections, there are still promising avenues for research that will tremendously benefit both practice and our understanding of this important process in the software development life-cycle.

REFERENCES

1. Ackerman, A. F., Buchwald, L. S. and Lewski, F. H. (1989) *IEEE Software,* 6**,** 31-36.
2. Banker, R. D. and Kemerer, C. F. (1989) *IEEE Transactions on Software Engineering,* 15**,** 1199-1205.
3. Basili, V. R. and Selby, R. W. (1987) *IEEE Transactions on Software Engineering,* 13**,** 1278-1296.
4. Bianchi, A., Lanubile, F. and Visaggio, G. (2001) In *Seventh International Software Metrics Symposium*, pp. 42-50.
5. Bisant, D. B. and Lyle, J. R. (1989) *IEEE Transactions on Software Engineering,* 15**,** 1294-1304.
6. Briand, L. C., Emam, K. E., Laitenberger, O. and Fussbroich, T. (1998) In *20th International conference on software engineering*, pp. 340-349.
7. Briand, L. C., Freimut, B. and Vollei, F. (2000) In *11th International symposium on software reliability*, pp. 124-135.
8. Chaar, J. K., Halliday, M. J., Bhandari, I. S. and Chillarege, R. (1993) *IEEE Transactions on Software Engineering,* 19**,** 1055-1070.
9. Collofello, J. S. and Woodfield, S. N. (1989) *Journal of Systems and Software,* 9**,** 191-195.
10. Dijkstra, E. W. (1971) In *Structured Programming*(Ed, Hoare, C.) Academic Press, pp. 1-82.
11. Dreger, B. J. (1989) *Function Point Analysis,* Prentice-Hall, Englewood Cliffs, New Jersey.
12. Dromey, G. R. (1995) *IEEE Transactions on Software Engineering,* 21**,** 146-162.
13. Emam, K. E., Laitenberger, O. and Harbich, T. (2000) *Journal of Systems and Software,* **54,** 119-136.
14. Fagan, M. E. (1976) *IBM Systems Journal,* 15**,** 182-211.
15. Fagan, M. E. (1986) *IEEE Transactions on Software Engineering,* SE-12, 744-751.
16. Hetzel, W. (1976) University of North Carolina, Chapel Hill.
17. Humphrey, W. S. (1989) *Managing the software process,* Addison-Wesley, Reading, Massachusetts.
18. IFPUG (1999) *Function Point counting practices manual,* International Function Points Users Group.
19. Jones, C. (1997) *Software Quality: Analysis and Guidelines for Success,* Thomson Computer Press, London.
20. Kemerer, C. F. (1987) *Communications of the ACM,* 30**,** 416-429.
21. Kusumoto, S., Matsumoto, K.-i., Kikuno, T. and Torii, K. (1991) In *15th Computer Software and Applications Conference*, pp. 424-429.
22. Kusumoto, S., Matsumoto, K.-i., Kikuno, T. and Torii, K. (1992) *IEICE Transactions on Information and Systems,* E-75-D**,** 674-680.
23. Laitenberger, O. (2002) In *Handbook of software Engineering and knowledge engineering,* Vol. 2 (Ed, Chang, S.-K.) World Scientific, River Edge, New Jersey.

24. Myers, G. J. (1978) *Communications of the ACM,* 21**,** 760-768.
25. Petersson, H. (2001) In *13th Australian Software Engineering Conference,* Canberra, Australia, pp. 160-170.
26. Porter, A. A., Siy, H. and Votta, L. G. (1996) *Advances in computers,* 42**,** 40-76.
27. Porter, A. A., Toman, C. A. and Votta, L. G. (1997) *IEEE Transactions on Software Engineering,* **23,** 329-346.
28. Porter, A. A. and Votta, L. G. (1997) *IEEE Software,* 14**,** 99-102.
29. Rodgers, T. L. and Dean, D. L. (1999) In *32nd Hawaii International Conference on System Sciences,* Vol. 3, pp. 13.
30. Russell, G. W. (1991) *IEEE Software,* 8**,** 25-31.
31. Weller, E. F. (1993) *IEEE Software,* 10**,** 38-45.
32. Wood, M., Roper, M., Brooks, A. and Miller, J. (1997) In *Fifth ACM SIGSOFT International Symposium on Foundations of Software Engineering,* Vol. 21 Zurich, Switzerland, pp. 262 - 277.