Trade-offs between Productivity and Quality in Selecting Software Development Practices

Alan MacCormack, *Harvard University*
Chris F. Kemerer, *University of Pittsburgh*
Michael Cusumano, *Massachusetts Institute of Technology*
Bill Crandall, *Hewlett-Packard*

Choosing appropriate practices for a project can be hard, given the various dimensions of performance each claims to optimize. Using data from 29 projects, this study examines the impact of eight development practices on both productivity and quality. Results suggest managers should choose a coherent system of practices on the basis of the specific objectives the software must meet.

Given software's important role in business, the design of effective development processes has received great attention. Early attempts at standardizing a formal software development model culminated in what's now known as the waterfall model. Over time, this view of software development as a process that could be managed proactively led to the development of other models, each proposing improvements. However, although some of the practices in these newer models complement the waterfall model, others appear to contradict its main principles. Furthermore, the performance dimensions that each practice seeks to optimize often differ. Unfortunately, these potential conflicts are difficult to resolve because the data presented to support most models is anecdotal and biased toward reporting only successful projects.

Our objective here is to look at how certain practices, proposed by various development models, affect performance. By exploring different practices' associations with multiple dimensions of performance across a number of projects, we examine the trade-offs involved in designing a process to optimize specific performance criteria. We carefully selected the practices for investigation to explore some of the potential conflicts that arise between different development models.

0740-7459/03/$17.00 © 2003 IEEE

Software development models and practices

Once upon a time, software development was seen mainly as an art. Stories of legendary developers abounded, describing how fast they could write code and how "cool" the resulting functionality was. The development process most prevalent in the early days was an implicit one: the "code and fix" model. This model reflected more the absence of a formal development process than any predetermined thought as to how to manage the process. In essence, developers built systems with minimal input as to what they were supposed to do, put them in front of clients, then tried to remedy the defects that were found. The process relied on the "star power" of the programmers involved and, for any system of significant complexity, almost guaranteed poor results.

The waterfall development model was introduced in the late 1960s in response to the problems of managing large custom software development projects. Its aim was to bring control and discipline to what had previously been a rather unstructured and chaotic process. The waterfall model involves completing a series of stages, including requirements definition, specification, planning, design, implementation, and integration. It emphasizes the need to develop comprehensive and detailed specifications up front and thereafter to execute efficiently according to these specifications.

The waterfall model proved to be a somewhat successful response to the early problems that had plagued software development, so people began trying to improve on it. These efforts included both examining other practices that were complementary to the waterfall model (for example, formal design and code reviews) and exploring new model types that might be better suited to environments where the waterfall model proved ineffective. Many of these newer models focused on markets where customer needs were uncertain or rapidly changing. The processes they advocated were therefore based more on the need for speed and responsiveness than on the traditional objectives of control and discipline. At the heart of these more flexible models was the notion that a prototype or an actual working version of the software should be released to customers at an extremely early stage of development.[1,2] To facilitate this process, development was often broken into several subcycles, each geared to producing a subset of the final product's functionality.[3,4]

So, a large number of software development practices have been proposed over the past 40 years, many as part of more comprehensive development models. The problem is not so much that we lack "silver bullets" for a project manager's arsenal but that we have such an extensive array of differently colored bullets that choosing among them has become impossible. How should managers make these choices? Is it possible to find a set of practices that will improve performance on all dimensions? Or must managers tailor the design of their processes to each project's specific requirements? The fact that different practices often bring similar benefits further complicates the problem. For example, conducting formal code inspections helps identify defects at an early stage, as does conducting daily builds and integration tests on a design.[5–8] Should managers use all these practices, or would one suffice? Finally, and perhaps most important, there are instances where the principles of one model appear to contradict those of another. Take, for example, the waterfall model, which stresses the need to complete a product's specifications before beginning its development. Many of the more flexible models, which let product specifications evolve as a project proceeds, violate this principle. So, do flexible models incur a performance penalty as a consequence? Or do they overcome this potential trade-off through the use of other mechanisms? These were the questions we set out to answer.

The problem is not so much that we lack silver bullets but that we have such an array of differently colored ones.

Empirical setting

Our results are based on a sample of Hewlett-Packard software development projects. (During this study, HP spun off Agilent as a separate firm, so this article reports data from both HP and Agilent.) HP is a large high-technology firm with operations in various businesses (for example, personal computers and peripherals, workstations and servers, measuring instruments and medical devices) across multiple geographic locations. The diversity of businesses and locations ensured wide variation in the practices we observed, while the use of a single firm helped control for extraneous factors that might have influenced development performance. This setting also helped facilitate data collection.

In most projects, coding began before the specification documents were complete.

We gathered data using a survey instrument hosted on an MIT Web server from the summer of 2000 through the winter of 2001. A company contact published the project's aims using HP's regular internal communication systems and asked project managers to complete the survey. We asked managers to use reference materials such as project data sheets, schedules, progress presentations, and resource plans. Although 35 managers responded, only 32 provided enough data to consider their project for analysis. After reviewing responses, we omitted three more projects due to their small size (they involved developing fewer than 1,000 new lines of code). So, the final sample for analysis consisted of 29 projects.

We collected data on two performance measures:

- *Productivity*: the number of lines of new code developed per person-day (an imperfect measure of productivity but one that could be measured consistently)
- *Quality* (also referred to as from here on as *defect rate*): the number of customer-reported defects per month per million LOC averaged over the first 12 months after launch

Software development practices

We collected data on eight software development practices. The first two measures captured the degree to which specifications were complete (as a percentage) before coding began in each project. We report data on two types of specification:

- *Functional* (or *requirements*) *specification*: a document that describes how features work but not the underlying structure of code or modules
- *Detailed design specification*: a document that describes the modules' structure and an outline of the algorithms where needed

Our next two measures captured whether formal design reviews and formal code reviews were used during development, as indicated by a binary variable. For example, the code review variable was set to 1 if one or more people typically reviewed another person's code before it could be checked into the system build.

The third pair of measures (as we defined them) related to the use of more flexible processes:

- *Subcycles*: Whether development was divided into separate subcycles, in which a subset of the final product's functionality was built and tested
- *Early prototype*: The extent to which an early prototype was shown to customers, as measured by the percentage of the product's final functionality that was complete when the first prototype was released

The final two measures captured the use of practices that provide rapid feedback on a design's performance: whether daily system builds were used during development and whether any type of integration or regression test was conducted at code check-in (binary variables were used in each case).

Descriptive statistics

Table 1 reports descriptive statistics for the sample. The median project involved developing 70,000 new LOC for a product totaling 170,000 LOC. The median project team size was nine people, with a median duration of 14 months. In terms of the type of software being developed, 59 percent of projects involved application software, 38 percent involved systems software, and 28 percent involved embedded software. (These numbers add up to more than 100 percent because some projects reported developing software of multiple types.)

In terms of performance, the mean defect rate of products was 18.8 defects per month per million LOC. The mean productivity achieved was 26 LOC per person-day. In terms of development practices, several points deserve comment. First, in most projects, coding began before the specification documents were complete; this was particularly notable for the detailed design specification, which was on average only 20 percent complete when coding began. Thus, in this sample, there were few pure waterfall models. Second, clear differences emerge in the penetration of different development practices: Around 30 percent of projects used daily builds throughout development, around half the projects used code reviews and integration or regression tests at check-in, and almost 80 percent of projects used design reviews. Finally, although 76 percent of projects

Table 1

Descriptive statistics for the sample's 29 projects

Variable	Description	Mean	Median	Standard deviation	Minimum	Maximum
Defect rate	Average no. of customer-reported defects per month per million lines of new code over first 12 months	18.8	7.1	23.1	0.0	80.0
Productivity	New LOC developed per person-day	26.4	17.6	24.0	0.7	85.0
Functional specification	Percentage of functional specification that was complete before team started coding	55%	55%	32%	0%	100%
Design specification	Percentage of detailed design specification complete before team started coding	20%	10%	26%	0%	80%
Design review	Binary: 1 if design reviews were performed during development, 0 if not	0.79	1	0.41	0	1
Code review	Binary: 1 if the number of people who typically reviewed another person's code was one or more, 0 if none	0.52	1	0.51	0	1
Subcycles	Binary: 1 if development was divided into separate subcycles, 0 if not	0.76	1	0.43	0	1
Early prototype†	Percentage of final product's functionality contained in the first prototype	38%	40%	24%	0%	90%
Daily builds	Binary: 1 if design changes were integrated into code base and compiled daily; 0 if not	0.32	0	0.48	0	1
Regression test	Binary: 1 if someone ran an integration or regression test when checking code into the build; 0 if not	0.55	1	0.51	0	1

Table 2

Simple correlations with performance

	Defect rate	Productivity
Control variables		
Systems projects	**0.436****	0.030
Applications	–0.058	–0.018
Embedded	0.014	0.066
Size (Ln{LOC})	**–0.562*****	**0.514*****
Process variables		
Functional specification	0.035	**0.473****
Design specification	–0.403*	–0.014
Early prototype†	**0.742******	**–0.624*****
Subcycles	–0.418*	0.021
Daily builds	–0.026	0.316
Regression test	–0.383*	0.180
Design review	**–0.545****	–0.227
Code review	–0.255	0.104

†: *Implies less functionality in the first prototype*
* p < *10%*, ** p < *5%*, *** p < *1%*, **** p < *0.1%*
Correlations in bold are significant (p < 0.05)

Table 3

Best-fit multivariate models of performance

Model	Defect rate	Productivity
Constant	16.36	34.9****
Systems	14.87**	
Early prototype†	0.48***	–0.42***
Daily builds		16.89**
Regression test	–12.64**	
Design review	–19.65**	
R-squared (adjusted)	74.6%	52.8%
F-ratio	15	11.1
Df	15	16

†: *Implies less functionality in the first prototype*
* p < *10%*, ** p < *5%*, *** p < *1%*, **** p < *0.1%*

divided development into subcycles, they varied greatly in how early they showed a prototype to customers. In some projects, coding had not begun at this point, whereas in others, the product's functionality was 90 percent developed.

Results

Our approach was first to assess the performance impact of each development practice, as well as control variables for the type of software and the size of the product, using simple correlations (see Table 2). For each performance dimension, we then developed a best-fit multivariate model using stepwise backward regression (see Table 3). To ensure that the results were robust, we removed from the analysis projects that were outliers on each performance dimension on a case-by-case basis.

Developers are more productive to the degree that a more complete functional specification exists prior to coding.

Simple correlations with defect rate and productivity

We found a significant relationship between systems software projects and defect rate, implying that these types of projects result in higher levels of customer-reported defects (per LOC) than application or embedded software projects. (All references to statistical significance in this article indicate relationships that are significant at the usual confidence levels ($p < 0.05$).) We also found a significant relationship between software size and both productivity and defect rate, implying that larger projects are more productive and have lower levels of customer-reported defects (per LOC).

Regarding the use of specifications, there was a significant relationship between the completeness of the functional specification and productivity. There was a weak relationship between the completeness of the detailed design specification and defect rate ($p = 0.078$). The former result suggests that developers are more productive to the degree that a complete functional specification exists prior to coding. This is intuitive, given that the functional specification outlines the features that developers must complete. To the degree that these are stated up front, developers can focus solely on "executing" these features in code. The latter result suggests that the existence of a more complete design specification up front contributes to a product with fewer defects. Again, this is an intuitive finding. Having a detailed description of each feature's design presupposes that some thought has gone into selecting which among the available alternatives are most robust.

With regard to using formal reviews, the relationship between design reviews and a lower defect rate was significant, but code reviews were not significant in predicting either outcome. The former result is intuitive, given that design reviews are likely to uncover errors in the overall development approach that might otherwise remain hidden until a product is operating in the field. An association between code reviews and lower defect rates might also be expected. However, our measure of defect rate captures only the bugs that customers report—meaning bugs that might otherwise be found through code reviews may be discovered through other development practices (for example, integration or regression tests) prior to product launch.

With regard to measuring flexible processes, we found a significant relationship between early prototyping and both a lower defect rate and higher productivity. We found a weak relationship ($p = 0.060$) between splitting development into subcycles and a lower defect rate. The former result suggests that getting early feedback from customers contributes to multiple dimensions of performance, mirroring the findings of previous empirical studies of product development in the software industry.[9] The latter result suggests that breaking a project into subcycles might contribute to a lower defect rate but has little effect on productivity. These results are intuitive: breaking development into subcycles and releasing an early prototype enable developers to identify and correct defects at an earlier stage in a project. Furthermore, early prototyping brings the promise that subsequent work will continue only on the features customers value most, with a subsequent positive impact on productivity.

Regarding practices that provide rapid feedback on design performance, there was a weak relationship ($p = 0.087$) between integration or regression testing at code check-in and a lower defect rate. The use of daily builds was not significant in predicting either performance measure, although it does appear in multivariate models, as we'll see later. These results suggest that a project's defect rate doesn't depend on whether it compiles daily builds of the evolving design but rather on whether the code that is checked goes through integration or regression tests that examine its performance. This makes intuitive sense, given that the tests themselves, not the mere submission of code, generate feedback on the design. With regard to daily builds, however, many managers reported varying build frequency according to project stage and integration level—for example, module versus system builds. Our measure, which reflects only the projects that conducted daily system builds throughout development, does not capture the subtleties of these approaches.

Multivariate models of defect rate and productivity

Table 3 shows the final multivariate regression models for both defect rate and productivity. We obtained these through a stepwise backward procedure that progressively eliminated variables that were not significant in

predicting the outcome. In the final model predicting defect rate, three development process measures and one control variable (for systems software projects) were significant. Together, these measures explain over 74 percent of the variation in defect rate. The significant development process measures are the use of

- An early prototype
- Design reviews
- Integration or regression testing at check-in

When we compare theses results to those of simple correlations, we can make several observations. First, controlling for size is no longer a significant predictor. Similarly, the weak relationships observed between defect rate and both the completeness of the detailed design specification and breaking development into subcycles are no longer present. By contrast, the measure of integration or regression testing is a stronger predictor in this model than when considered alone.

To illustrate the magnitude of the effects we observe in this model, consider the example of an applications software project that releases a prototype to customers when 40 percent of the functionality is complete (the sample median). The model suggests that such a project typically has a defect rate of 35.6 defects per million LOC per month. (We get this figure by substituting appropriate numbers into the equation implied by the model.) Now, releasing a prototype when only 20 percent of the functionality is complete is associated with a 27 percent reduction in the defect rate, to 26.0. Adopting the practice of integration or regression testing at code check-in is associated with a 36 percent reduction in the defect rate, to 22.92. Finally, adopting the practice of design reviews is associated with a 55 percent reduction in the defect rate, to 15.9. We conclude that the benefits from adopting such practices are significant.

In the final model predicting productivity, two development process measures are significant. Together, these measures explain over half the variation in productivity. The significant development process measures are the use of

- An early prototype
- Daily builds

Comparing these results to the results using simple correlations, we note first that the measure of functional-specification completeness is no longer a significant predictor in this model. It appears that the observed trade-off between specifying less of a product's requirements up front and productivity disappears in the presence of other process variables. Second, the use of daily builds is a significant predictor in this model, although it was not when considered alone. This suggests that this practice interacts with others in the model in a way that explains the remaining variance left only after more powerful predictors have taken effect.

To illustrate the magnitudes of the effects we observe in this model, we again consider an applications software project in which a prototype is released to customers when 40 percent of the functionality is complete. The model suggests that such a project typically has a productivity of 18.1 LOC per person-day. Now, releasing a prototype when only 20 percent of the functionality is complete is associated with a 35 percent productivity increase, to 26.5. Adopting the practice of daily builds is associated with a 93 percent productivity increase, to 35.0. Again, we conclude that the benefits from adopting such practices are significant.

Discussion

Our findings shed light both on the relationships between specific practices and performance and on the nature of the interactions among the various practices that form part of a development model.

Practices and performance

The results illustrate that different software development practices are often associated with different dimensions of performance. For example, the use of integration or regression tests as code is checked in appears in our final model predicting defect rate but not in the model predicting productivity. Conversely, the use of daily builds appears in our final model predicting productivity but not in the model predicting defect rate. For practitioners, this result implies that the choice of development model for a particular project—and hence the set of practices used—should differ according to the specific performance dimensions that must be optimized. Projects optimized for productivity will use different combinations of practices than projects optimized for quality. This conclusion is particularly rel-

Different software development practices are often associated with different dimensions of performance.

Development models should be considered as coherent systems of practices.

evant for practices that incur significant costs (in terms of time, money, or resources).

Our results also indicate that some software development practices are in fact associated with multiple dimensions of performance. For example, releasing a prototype earlier in development appears to contribute to both a lower defect rate and higher productivity. This finding complements other studies, which have shown that an evolving product design's early release to customers is associated with a final product that is better matched to customer requirements.[9] Given that this measure consistently predicts several dimensions of performance in different samples of projects, it represents a uniformly good software development practice.

Finally, some practices that are not correlated with performance when considered in isolation appear as significant predictors in our multivariate models. The use of daily builds follows this pattern, appearing in the final model predicting productivity despite not being correlated with this performance dimension on an individual basis. This suggests there is a "pecking order" in terms of the relative importance of various development practices. Some appear to explain the residual variance left unexplained only after more dominant process parameters (for example, early prototyping) have taken effect.

Coherent systems of practices

Some practices that are correlated with performance when considered in isolation do not appear as significant predictors in multivariate models. For example, we initially found that having a less complete functional specification when coding begins is associated with lower productivity, lending support to the waterfall model of development. When we accounted for the variance explained by releasing early prototypes and using daily builds, however, this relationship disappeared. (A similar pattern exists for the relationship between a more complete detailed design specification and a lower defect rate.) In a sense, these other practices appear to "make up" for the potential trade-offs arising from an incomplete specification. This suggests that development models should be considered as coherent systems of practices, some of which are required to overcome the potential trade-offs arising from the use (or absence) of others.

To illustrate our argument, consider that without a complete functional specification when coding begins, a development team would have to evaluate what features should be in the product as they went along. As a result, they would likely be less productive than in projects having a complete specification. However, to overcome this trade-off, they might organize development tasks so they could release a prototype early in the project. The aim would be to solicit feedback on the features that should be in the final product, providing a surrogate for the information that a specification would normally contain. In essence, early prototyping provides an alternative mechanism for obtaining the benefits that a complete specification would normally bring.

Importantly, these dynamics help shed light on the debate about whether more flexible development models are likely to suffer along traditional performance dimensions such as productivity and quality. These doubts arise in part from the fact that many of these models appear to lack the control mechanisms that are built into the waterfall model. Our results suggest, however, that more flexible models compensate for such problems by using alternative practices geared to overcoming the potential trade-offs. This finding highlights the danger in assuming that you can implement a more flexible process piecemeal by "picking and choosing" among the many practices claimed to support greater flexibility. To the degree that such a process relies on a coherent system of practices, a piecemeal approach is likely to lead to disappointment.

Overall, our results suggest that at the beginning of each project, practitioners should establish the primary performance objectives for the software deliverable, given that these will largely drive the type of development model and mix of practices they should use. In our sample of projects, only one practice—releasing an early prototype during development—was associated with both higher productivity and a lower defect rate.

At a higher level, our results shed light on

potential conflicts between traditional and more flexible development models. Indeed, they provide a way to reconcile the seemingly opposite viewpoints that these approaches are founded on. More flexible processes do not have to incur a performance penalty vis-à-vis models such as the waterfall model, even though they appear to violate the principles of such models. This is because they are based on coherent systems of practices, in which some practices are geared to overcoming the potential performance trade-offs from the use of a more flexible process. Our findings highlight why future researchers should gather data on multiple dimensions of performance as well as on alternative sets of practices that might provide similar benefits to the ones they are investigating. Only with such an approach will the subtleties we observed in this study become more apparent.

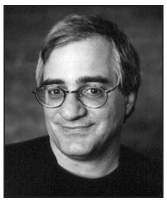
References

1. J.L. Connell and L. Shafer, *Structured Rapid Prototyping: An Evolutionary Approach to Software Development*, Yourdon Press, 1989.
2. B. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, vol. 21, no. 5, May 1988, pp. 61–72.
3. C. Wong, "A Successful Software Development," *IEEE Trans. Software Eng.*, vol. 10, no. 6, Nov. 1984, pp. 714–727.
4. T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, 1988.
5. A. Porter et al., "An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development," *IEEE Trans. Software Eng.*, vol. 23, no. 6, June 1997, pp. 329–346.
6. M. Cusumano and C.F. Kemerer, "A Quantitative Analysis of US and Japanese Practice and Performance in Software Development," *Management Science*, vol. 36, no. 11, Nov. 1990, pp. 1384–1406.
7. M.A. Cusumano and R.W. Selby, *Microsoft Secrets*, Simon & Schuster, 1998.
8. S. McConnell, *Rapid Development*, Microsoft Press, 1996.
9. A. MacCormack, "Product-Development Processes that Work: How Internet Companies Build Software," *Sloan Management Rev.*, vol. 42, no. 2, Winter 2001, pp. 75–84.

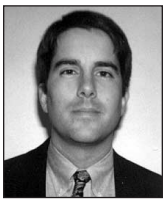
About the Authors

Alan MacCormack is an associate professor of business administration at the Harvard Business School. His research explores the management of technology and product development in high-tech industries, particularly the software industry. He received his doctorate in business administration from the Harvard Business School, where he received the George S. Dively award for distinguished research. Contact him at Morgan Hall T39, Harvard Business School, Soldiers Field Park, Boston, MA 02163; amaccormack@hbs.edu.

Chris F. Kemerer is the David M. Roderick Professor of Information Systems at the Katz Graduate School of Business, University of Pittsburgh. His research interests include software engineering measurement and modeling, economic issues in information systems, and technology adoption and diffusion. He received his PhD in systems sciences from Carnegie Mellon University. Contact him at 278A Mervis Hall, KGSB, Univ. of Pittsburgh, Pittsburgh, PA 15260; ckemerer@katz.pitt.edu.



Michael Cusumano is the Sloan Management Review Distinguished Professor at MIT's Sloan School of Management. He specializes in strategy, product development, and entrepreneurship in the software business. He received his PhD in Japanese studies from Harvard University and a postdoctoral fellowship in production and operations management at the Harvard Business School. His most recent book, *The Software Business: What Every Manager, Programmer, and Entrepreneur Must Know, in Good Times and Bad*, is forthcoming in 2004. Contact him at the MIT Sloan School of Management, 50 Memorial Dr., Rm. E52-538, Cambridge, MA 02142-1347; cusumano@mit.edu.



Bill Crandall is director of product generation services at Hewlett-Packard. His team is responsible for developing and disseminating new and better ways of developing products, services, and solutions across HP and for delivering shared engineering services across HP. He holds an MS in computer science and an MS in management from MIT, where he was a fellow in the Leaders for Manufacturing program. He is a member of the ACM. Contact him at Hewlett-Packard, 1501 Page Mill Rd., MS 1229, Palo Alto, CA 94304-1126; bill.crandall@hp.com.

SOFTWARE ENGINEERING GLOSSARY

Peer reviews

peer review: A semiformal to formal evaluation technique in which a person or group other than the originator examines software requirements, design, or code in detail to detect faults, violations of development standards, and other problems; sometimes called *walkthrough* or *inspection*.

inspection: A type of peer review in which a group of the developer's peers checks product documents at specific points in the development process to find errors in the product.

author: The person responsible for the software product meeting its inspection criteria, contributing to the inspection based on special understanding of the software product, and for performing any rework required. [IEEE Std. 1028-1997]

moderator: The person responsible for planning and preparing for an inspection, ensuring that it is conducted in an orderly way and meets its objectives, collecting inspection data (if appropriate), and issuing a report. [IEEE Std. 1028-1997]

California State University, Sacramento ■ thayer@csus.edu

■ Richard H. Thayer