

Appendix O

Swords & Plowshares - *The Rework Cycles of Defense and Commerical Software*

Content

O.1 Introduction	O-3
O.2 The Rework Cycle	O-3
O.2.1 Observations	O-5
O.2.1.1 The Sample	O-5
O.2.1.2 The Problems	O-5
O.2.1.3 Rework Creation	O-6
O.2.1.4 Rework Discovery	O-6
O.2.1.5 Rework Execution	O-7
O.2.1.6 Progress Monitoring	O-7
O.3 Implications for Improvement	O-8
O.4 Conclusions	O-8
O.5 Notes	O-10
O.6 Acknowledgments	O-10
O.7 About the Authors	O-11

O.1 Introduction

A worldwide survey recently revealed that less than half of all development projects meet their targets for development time and cost.¹ Technological development projects dominated by software-based systems constitute increasingly significant portions of companies' new product and business plans. Traditionally defense-oriented firms in the post-Cold War period search, with varying degrees of aggressiveness and desperation, for how to make an effective transition toward commercial markets. Companies already firmly established in commercial software markets search, with varying degrees of frustration, for ways to bring new products to market faster and at lower cost. Indeed, the success of virtually all companies has never before been more dependent upon timely, low-cost development project execution (nor more threatened by its absence).

With the magnitude of the stakes involved, why does there seem to be so little progress in achieving better-managed software development projects? Why in such projects are cost and schedule problems so persistent and pervasive? What are the underlying sources of consistently “*surprising*” project overruns? Why are we so bad at estimating when developing products will be completed and ready for the market? How far must “defense” firms be prepared to go to become commercially competitive? What must both they and commercial firms alike do in order to achieve dramatic project performance improvement? Are there any fundamental lessons we can learn and transfer from one “*unique*” project to another?

Here we aim to provide some initial answers to these questions. To do so we draw upon over ten years of experience, shared with our colleagues, in developing and applying computer-based dynamic simulation models² of software system development projects. We have used such models to accurately recreate, forecast, diagnose, and improve the performance on dozens of major development programs and projects in aerospace, defense electronics, financial systems, construction, and telecommunications. Among these, many whole projects and significant segments thereof have been dedicated to software system development.

At the core of the structure of these models is a different but straightforward view of development project work — one which recognizes the rework cycle³. Indeed, what is most lacking in conventional methods for project planning and monitoring is any acknowledgment or measurement of *rework*. For all their utility, most planning tools treat a development project as being composed of individual, discrete tasks which are “*to be done*,” “*in process*,” or “*done*.” No account is taken of incomplete or imperfect task products, or the amount of rework needed. This is particularly inappropriate for naturally iterative development efforts; the dozens of analyses we've conducted show that rework can account for the majority of project work content and *cost*!

O.2 The Rework Cycle

Using dynamic simulation models to analyze and aid the management of software development compelled us to simulate the performance of actual projects as they really did occur — not just how they are planned to go, or how they “*should*” go. Hence, we had to design and employ a structure which could accurately recreate such projects. To do so we had to treat their substantial rework explicitly, as well as its causes, detection, and execution. We developed a core structure which proved to be universally applicable to development projects and project stages. We term this structure “*the rework cycle*.” See Figure O-1.

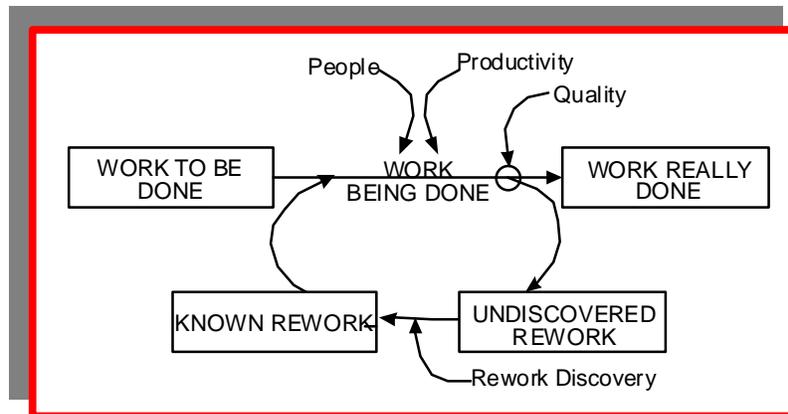


Figure O-1 The Rework Cycle

At the start of a project or project stage, all work resides in the pool of Work To Be Done. As the project begins and progresses, changing levels of People working at varying Productivity determine the pace of Work Being Done. But unlike all other program/project analysis tools and systems, the Rework Cycle portrays the real-world phenomenon that work is executed at varying, but usually less than perfect, “Quality.” A fraction that potentially ranges from 0 to 1.0, the value of Quality (as well as that of Productivity) depends on many variable conditions in the project and company. The fractional value of Quality determines the portion of the work being done that will enter the pool of Work Really Done, which will never again need redoing. The *rest* will subsequently need some rework, but for a (sometimes substantial) period of time the rework remains in a pool of what we term Undiscovered Rework — work that contains as-yet-undetected errors, and is therefore perceived as being done. Errors are detected by “downstream” efforts or testing; this Rework Discovery may occur months or even years later, during which time dependent work has incorporated these errors, or technical derivations thereof. Once discovered, the Known Rework demands the application of People, beyond those needed for completing the original work. Executed rework enters the flow of Work Being Done, subject to similar Productivity and Quality variations. Even some of the reworked items may then flow through the rework cycle one or more subsequent times.

Undiscovered rework plays a pivotal role in the propagation of problems through a project. Lurking undetected — for example, as a software “bug,” or design flaw — it causes productivity loss and work delays, and triggers rework cycles on downstream dependent tasks. The more tightly-scheduled and parallel the project tasks, the more of a “multiplier effect” on subsequent rework cycles. Undiscovered rework is the single most important source of project cost and schedule crises. To control undiscovered rework on software development projects, we must:

- **Acknowledge** its existence,
- **Plan** so as to allow for it,
- **Measure** it (once it is recognized as known rework),
- **Prevent** it (i.e., improve “quality,” as defined here) as much as possible, and
- **Seek** to find and identify it early, so as to reduce its propagation.

We offer the following observations in the hope of spurring companies and individual managers to do just that.

O.2.1 Observations

The full simulation models of these development projects employ thousands of equations. They explicitly portray the time-varying conditions which cause changes in productivity, quality, staffing levels, rework detection, and work execution, as well as the interdependencies among multiple project stages. All of the dynamic conditions at work in these projects and their models (e.g., staff experience levels, work sequence, supervisory adequacy, “*spec*” stability, worker morale, task feasibility, vendor timeliness, overtime, schedule pressure, hiring and attrition, progress monitoring, organization and process changes, prototyping, testing) cause changes — some more directly than others — in the performance of the rework cycle. Because our business clients require demonstrable accuracy in the models upon which they will base important decisions, we have needed to develop highly accurate measures of all these factors, especially those of the rework cycle itself. We do not, however, offer here a treatise on model design. Instead, we draw upon the many real business applications of our models to provide heretofore unavailable guidelines and benchmarks on the characteristics of the rework cycle.

O.2.1.1 The Sample

The observations come from seven defense and fourteen commercial software development efforts. They range from modest upgrades of existing systems that involve about ten people for a year, up to major first-of-a-kind system development efforts involving many hundreds of people for several years. The defense projects as a group differ from the commercial projects in many ways (average size, duration, customer for the product, etc.), so we’ve examined the averages and the ranges of values within these groups as well as across all the projects. Figure O-2 [*not available in this format*] shows the difference in planned size and duration, with the commercial projects averaging under 130,000 hours of planned work over the course of about a year’s schedule, versus 170,000 hours and a two-year-plus average planned duration for the defense projects.

O.2.1.2 The Problems

The defense projects’ performance against plans was substantially worse than those of commercial developments. See Figure O-3 [*not available in this format*]. On average, the defense projects took about three times the planned hours, versus about 1.4 times the planned hours for the commercial projects. Schedule performance was even worse. The commercial projects were poor, taking nearly twice as long to complete as planned. On average, though, defense projects took over four times as long to complete as was originally planned.

How could this be? These projects involved some of the brightest, most experienced, and hardest-working people in the field. They were managed by seasoned and conscientious managers using the standard planning tools, and yet they still cost many times their budgets. We acknowledge the bias caused by the fact that easy, smoothly-running projects rarely command the attention of external consultants. Still, these projects are not anomalies; recall that *most* projects fail to meet their targets. We believe there are systemic reasons why software development projects perform so poorly.

To be fair, *some* of the problems — especially on the defense side — are due to midstream scope and specification changes by the “*customer*,” be they internal or external to the company. These are not fully anticipated at the start of the project, and not reflected in the original budgets and schedules (although, given the history of large projects, such changes are to be expected). The difference in the typical magnitude

of midstream changes is part of the reason for the differences between defense and commercial project performance. However, even adjusted for those changes, the same patterns remain. We believe that our use of simulation modeling has helped identify the systemic causes of those patterns, through the characteristics of the Rework Cycle — rework creation, identification, and execution.

O.2.1.3 Rework Creation

In the rework cycle, “*quality*” is the fraction of work being done at any point in time which will not eventually need to be reworked; the lower the quality, the more rework. The average levels of quality on defense projects was half that observed on commercial projects, 0.34 versus 0.68. See Figure O-4 [*not available in this format*]. In other words, only about 1/3 of the work products being executed on a defense software development project will not need reworking (or *another* round of reworking), as opposed to 2/3 in commercial projects.

The rework cycle quality on the worst of commercial projects was nearly as low as that of the average defense software project, but the best was nearly “*perfect*.” The best among defense efforts exhibited a 0.55 quality, but the worst was near 0.10.

O.2.1.4 Rework Discovery

We know of no company which routinely monitors or measures the amount of time elapsed between the commission of design or coding errors and their detection. And yet it is undoubtedly one of the most critical determinants of “*time to market*” and of the true quality (in the conventional sense) of the delivered product. Think for a moment. On projects of this nature, what would you say is the typical amount of time between making and finding an error? A week? Two? A month? The actual average across all these projects is about *nine months*. See Figure O-5 [*not available in this format*].

On this measure we find little absolute difference between defense and commercial projects, on average. In fact, the worst of commercial software projects exhibited longer rework discovery times than the worst defense projects. Perhaps this is due to the extraordinary amount of oversight, testing, and review endemic to defense projects. Regardless of the cause, it is troublesome to note that for commercial efforts the average scheduled duration of work, eleven months (versus 29 for defense), is *only three months longer* than the time to detect the need for a single round of rework.

Only the *relatively* high levels of quality in commercial software developments prevent consistent disasters. Even so, rework discovery times which are nearly the length of the planned work explain why firms so consistently overpromise on software product introduction dates (open any business or software journal for an example). Rework keeps being found near “*the last minute*,” and revised announcements sheepishly admit to the delay. The more aggressively the projects are scheduled in response to competitive pressures to “*be first*,” or to respond fast, the worse this tendency. No “*productivity*” gain will help — only improvements (reductions) in the rework discovery time will.

And when you consider that a project with a 0.34 average quality (the defense average) will require seven cycles of rework to surpass 95% real completion, for a rework discovery time of 9-10 months for each cycle would add over five years to an effort for which rework was not planned. Interestingly (and not coincidentally), that is near the amount by which real completion of defense software development projects in our sample exceeded their schedule targets.

O.2.1.5 Rework Execution

As the rework cycle structure indicates, *some work may cycle through multiple times before it is complete and correct*. Figure O-6 [not available in this format] shows the average number of full revisions⁵ of each task (e.g., specifications, modules). The average task product on defense projects was revised fully three times. On commercial projects, only 40% of the tasks were revised, on average. This is the major reason for the cost and schedule performance differences. Some commercial software projects saw over one full round of revisions, and defense projects as many as seven.

So where was all the time spent? On defense projects, more time was spent on rework than was spent to do tasks for the first time. See Figure O-7 [not available in this format]. An average of nearly one and a half hours were spent fixing, for every one hour spent to do it the first time. Even on commercial projects, over forty minutes were spent on rework for every hour spent on the first iteration. This is even more astonishing when put in perspective: about half of the time spent on all these software development projects was for work not even tracked in most planning and monitoring systems!

No wonder managers of software projects have difficulty — they're using tools that only let them see half of the job. Even worse, in the later stages of a project, 70, 80, even 90 % and more of the work is “invisible” to their planning systems. Managers and staff on these projects aren't incompetent, they aren't (usually) deceitful, and they aren't lazy — *they are misled by their planning tools*.

Experienced managers are certainly aware of rework, and make allowances for it. However, this means that all of their “sophisticated” planning tools become nearly worthless at the end of the project, when the bulk of the work is rework. Instead that effort is being forecasted purely on the basis of instinct. For most project managers, a handful of projects constitutes a career. By the time some managers work up to a really difficult project, they may have had little experience with “major rework” on which to base their planning (and promises). Under tremendous pressure to deliver (from senior management, the customer or the marketplace, and dependent efforts), managers' well-intended plans and promises will be thwarted by the rework cycle.

O.2.1.6 Progress Monitoring

Just how thwarted those plans can be is illustrated in Figure O-8 [not available in this format]. This “progress ramp” displays the accuracy of progress monitoring in the sampled commercial software developments. For the range of commercial efforts modeled, the display charts the *perceived* progress, as it was reported at different points in time, against the *real* progress (which excludes undiscovered rework) at that time. Perfectly accurate project progress monitoring would yield a straight 45° diagonal (hence the triangular ramp shape): at a perceived/reported condition of 20% complete, the *actual* % complete would be 20%, and so on. Instead, real progress is typically less than reported progress.⁶

Note that the “best” of the sample achieved nearly perfect monitoring. The lower the quality, and the longer the rework discovery time (hence the more undiscovered rework), the larger the gap between real and reported progress, and the longer that gap persists: the “worst” of the commercial efforts reported 90% completion when as little as 60% was really complete. With 40% of the effort really left, it is easy to see why that last “10%” can seem to take so (unexpectedly) long, with the attendant delays in product introduction dates. Indeed, the product is often delivered when perceived “complete,” leaving it to customers to find the as-yet-undiscovered rework. Before moving on to a comparison with defense software projects, imagine the difference in ease of management and accuracy of projections just between these two commercial

extremes. When considering best-practice/TQM/process re-engineering, consider the differences displayed here — all caused by variations in “quality” and rework discovery time.

Figure O-9 [not available in this format] overlays the same kind of envelope for progress monitoring in the defense software efforts modeled. Generally longer rework discovery times, and notably lower levels of “quality,” produce a much more bowed shape overall, reflecting less accurate progress monitoring. The best of the defense projects is near the typical commercial project in progress monitoring accuracy. The worst among the defense efforts, with quality levels near 0.10, reports 75% completion *when less than 15% is really done*. After reaching 90% reported completion, the line goes nearly vertical — meaning a long time was spent thinking the end was near, only to discover more and more rework that extended the project and increased its cost far beyond the original and interim plans. By these measures again, the distance that defense firms must move in order to be viable commercial competitors is a long one. And, again, “productivity” improvement is not the answer. Instead, the answer lies in measures that are rarely monitored, let alone being the focus of control and improvement efforts — quality and rework discovery times.

O.3 Implications for Improvement

The nature of the performance improvement that can be achieved if efforts are successfully focused on rework cycle quality is illustrated in Figure O-10 [not available in this format]. The individual data points from all the software development efforts chart (a) their average quality achieved versus (b) their costs incurred, as a ratio to their budgets. As an example, we’ve circled one data point for a project that achieved an average of 70% quality, and saw a cost/budget ratio of about 1.5 (a 50% overrun).

Of course, the accuracy of the original budgets causes some deviation among the plotted points. Nevertheless, the pattern is clear: higher quality, lower cost. Most projects with quality levels in excess of 0.70 achieve costs comparatively near their budgets. In contrast, projects with a quality less than 0.40 exceeded their budgets by factors of 3, 4, ... 5!

We overlaid on this chart a line that plots from several *simulations* the resulting costs versus budget for one identical project which varies among the simulations only in the average quality achieved. We do so to drive home the extent to which quality improvement translates to cost performance improvement. With no productivity change whatsoever, a change in rework cycle quality from 0.35 to 0.55, for example, would eliminate most of the project cost overrun, reducing costs by an amount equal to the original budget.

O.4 Conclusions

We need to improve our fundamental understanding of how development projects really work. In order to avoid the persistent cost and schedule performance problems so closely associated with software development efforts, we must take a more strategic and realistic view of project work content and processes. While the products and technical steps may indeed be unique, we need to recognize that there are common structures and processes, and common problem causes. Only then is it possible to extract lessons and to implement changes that achieve radical improvements in project performance and business success.

Our experience in simulating software development projects indicates that conventional methods and systems are inadequate to support the management of such projects. Further, the improvement in project or product development performance sought by most companies is frustrated by the prevailing managerial mindset. It

is a mindset encouraged by the use of systems which treat projects as the sum or sequence of purely discrete tasks.

We need to recognize the flows of work in software development projects, flows in which there are multiple *rework cycles*. Managerial systems which ignore rework and its cycles are deficient, misleading, and constitute a roadblock to achieving breakthrough improvements in project and product development performance.

Indeed, we see from the reported results a clear indication of just how dramatic a breakthrough is required in order for traditionally defense-oriented firms to transfer their substantial expertise to compete effectively in commercial markets. Their cost performance against targets must improve by at least *a factor of 2*. Schedule performance versus targets must improve by *a factor of 3*, lest faster and more nimble commercial competitors thwart the attempted transition to “*plowshares*.” Rework content in their projects must be cut to *1/3 to 1/2* of currently prevailing levels. The “*quality*” of on-going work must *double*. Rework *detection* must be encouraged, to avoid the snowballing effects of undiscovered rework. Both better quality and rework detection are required to improve radically managers’ ability to: (a) contain costs; (b) assess accurately the true state of ongoing projects’ work progress; and (c) foretell dependably the time at which developing products will be completed and delivered to the market.

And all this presumes that commercial development efforts themselves do not improve — that they represent a standing target. This would be imprudently optimistic, for there is substantial room for improvement in most complex commercial software developments as well. Managerial, technical, and procedural improvements which could increase work quality by just 10 points would cut cost overruns in half. Most significant for commercial projects, reductions in the rework discovery time would yield a faster “*time to market*,” and more dependable estimates thereof. But we cannot control by mandate the “*levers*” of quality and rework detection in the rework cycle. Instead we need to influence them through that which we **can** control, or more directly influence — interim schedule targets, staffing, monitoring systems, coding techniques, testing practices.

The role of a simulation model on a specific project is to portray accurately the project and to aid its managers in evaluating potential actions through “*What if*” analyses. However, employed on several development projects, the model also helps to illuminate the underlying structure of such projects. In the “*rework cycle*” we have developed a near-universal structure which facilitates both roles. The resulting improved understanding helps managers to identify transferable lessons. In turn, these do lead to significant performance improvements. Indeed, without some more complete and realistic “*model*” than is now the norm, the seemingly intractable cost and schedule problems of software development projects will continue to plague defense and commercial firms alike.

O.5 Notes

1. “Strategic Management of Technology: Global Benchmarking,” Dr. Edward B. Roberts, December 10, 1992, Cambridge, MA.
2. These project models were built using the dynamic continuous simulation language DYNAMO; see DYNAMO User’s Manual, Pugh-Roberts Associates, and Introduction to System Dynamics Modeling with DYNAMO, Richardson, George P. and Pugh III, Alexander L.
3. The concepts and workings of the rework cycle model were explained in “The Rework Cycle: Benchmarks for the Project Manager”, Cooper, K. G., from which some introductory description here is excerpted, and first published in Project Management Journal, March 1993.
4. Do the math: Really complete after 1st release= .34; after 1st rework cycle= prior complete + (quality • remainder)= .34 + (.34 x .66)=.56; after 2nd cycle= .56 + (.34 x .44)=.71; 3rd= .71 + (.34 x .29)=.81... after 7th rework cycle= .96
5. Revisions reported here are normalized to be equal in effort to a full re-execution of the first release of the work product, so as to correct for significant variations in the effort content of different revisions.
6. With the benefit of data on a completed project or project stage, one may construct one’s own “*progress ramp*” chart by plotting for the completed project: (1) the historically reported “% complete,” versus (2) a retrospective computation of the % really complete then (you should compute the % really complete based on hours spent to that point, relative to the total hours eventually spent).

O.6 Acknowledgments

Many years of work by our colleagues and clients have gone into making possible these observations and findings. Without naming the dozens of individuals and companies with whom we have worked, we wish to acknowledge here their invaluable contribution. We wish to thank explicitly our colleagues, Dr. Thomas G. Kelly and Alexander L. Pugh, for their substantial and timely help in preparing the material on which this article is based. The interpretation of the assembled information remains the authors’ responsibility.

O.7 About the Authors

Kenneth G. Cooper is Director of the Management Simulation Group and Senior Vice President of Pugh-Roberts Associates, a division of PA Consulting Group. Mr. Cooper's management consulting career spans twenty years, specializing in the development and application of computer simulation models to a variety of strategic business issues. His clients include AT&T, Aetna, Arizona Public Service, Hughes Aircraft, IBM, Litton, MasterCard, McDonnell-Douglas, Northrop, Rockwell, and several law firms. Mr. Cooper has directed over a hundred consulting engagements, among them analyses of sixty major commercial and defense development projects. Mr. Cooper is an original author of the program management model introduced in this article. His group's offices are in Cambridge, Massachusetts and Oxford, England. Mr. Cooper received his bachelor's and master's degrees from M.I.T. and Boston University, respectively.

Thomas W. Mullen is a Senior Manager in the Management Simulation Group of Pugh-Roberts Associates, a division of PA Consulting Group. Mr. Mullen has managed and participated in over thirty consulting projects in his eight years with the firm. He has concentrated on the use of simulation models to analyze and aid major commercial and defense development programs. Mr. Mullen has worked with many clients in the aerospace, software, and financial services industries. He has also managed the development of several simulation software products. Mr. Mullen received both his bachelor's and master's degrees from M.I.T.'s Sloan School of Management.