Iterative Rework: The Good, the Bad, and the Ugly

In software development, some rework is both inevitable and beneficial. But how much is too much and how little is too little? Understanding the nature of rework and why it occurs can help answer these questions.

Richard E. Fairley
Oregon Health and Science University

Mary Jane Willshire
Colorado Technical University

Because the creative processes in developing and modifying software are subject to myriad external and changeable forces, it is impossible to get all but the simplest products right in one pass. Long experience has shown the advantages of iterative development, in which each iteration subsumes the previous iteration's software and adds capabilities to the evolving product to produce a next version. As developers build and validate the next version's capabilities, they rework the previous version to enhance its capabilities and fix defects discovered while integrating the new and old versions.

Iterative development can take many forms, depending on the project's goals: *Iterative prototyping* can help evolve a user interface. *Agile* development is a way to closely involve a prototypical customer in a process that might repeat daily. *Incremental build* lets developers produce weekly builds of an evolving product. A *spiral model* can help the team confront and mitigate risk in an evolving product.

Each iteration involves a certain amount of rework to enhance and fix existing capabilities (the good). However, excessive rework could indicate problems in the requirements, the developers' skills and motivation, the development processes or technology used, or all of the above (the bad). Exorbitant levels of rework result in truly untenable situations (the ugly).

On the other hand, too little rework could indicate insufficient review and testing or too little anticipation of the product features needed to support the next version (bad that can turn ugly). Understanding and correcting the root causes of problems that result from too much or too little rework can significantly increase productivity, quality, developer morale, and customer satisfaction.

ITERATIVE DEVELOPMENT MODELS

All forms of iterative development provide a way to

- continuously integrate and validate the evolving product,
- frequently demonstrate progress,
- alert developers early on about problems,
- deliver subset capabilities early on, and
- systematically incorporate the inevitable rework that occurs in software development.

Agile and incremental-build are two commonly used iterative development models. In both models, significant changes to requirements, design constraints, or environmental factors such as changes to middleware application programming interfaces (APIs) or hardware features can require significant rework of the design and existing code.

Agile development

Although the agile theme has several variations, most agile process models adhere to five principles (http://agilemanifesto.org/principles.html):

- Continuously involve a prototypical customer.

Published by the IEEE Computer Society

- Develop test cases and test scenarios before implementing the next version.
- Implement and test the next version.
- Demonstrate each version to the customer and elicit the next requirements.
- Periodically deliver evolving subsets of the product into the operational environment.

The customer's role is to review progress and provide the story line that determines the requirements for new capabilities and revisions to demonstrated capabilities (www.martinfowler.com/articles/newMethodology.html).

Figure 1 depicts an iterative model for agile development. Some agile process models require developers to produce a next version of a running system within one workday. Some use pair programming, in which pairs of developers share a computer terminal to develop their software.

Experience with agile models indicates that the resulting products rate high in customer satisfaction and have low defect levels. Customer satisfaction, however, depends critically on involving a knowledgeable prototypical customer. Some critics maintain that products from an agile process may have functional structures, which are hard to modify, and no design documentation. Others note the approach's lack of scalability.

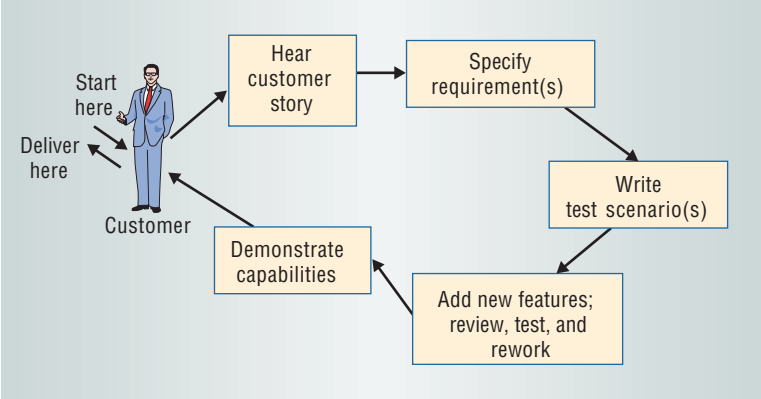

Figure 1. Steps in the agile development process. The process from customer story to demonstrated capabilities is iterative in nature, with Iterations typically occurring in cycles of one day or less. Developers may periodically deliver versions to users.

Incremental build

In contrast to agile models, in which requirements and architecture evolve, the incremental-build model is based on stable requirements and an architectural design. As Figure 2a shows, the model partitions requirements and architecture into a prioritized sequence of builds. Each build adds capabilities to the incrementally growing product. Developers typically produce a next version of a demonstrable system each week. Each version integrates, tests, and demonstrates the progress that all developers have made.

The incremental-build process works well when each team consists of two to six developers plus a team leader. Team members can work as individuals or perhaps in pairs using an agile process. Each individual or pair will typically produce several unofficial builds during each development cycle using a copy of the current official version as a testbed.

The incremental-build model scales well to large projects. Developers partition the architecture into well-defined subsystems and allocate requirements and interfaces to each. The team can independently test and demonstrate subsystems, perhaps using

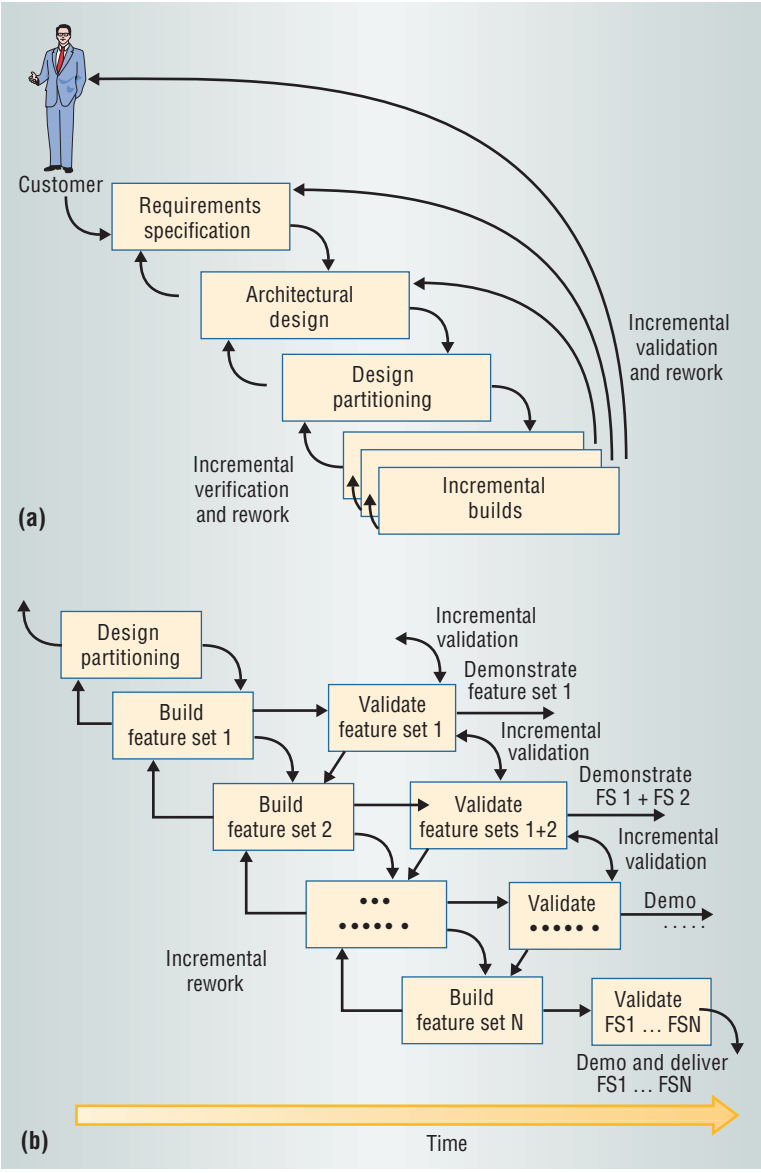

Figure 2. An incremental-build model: (a) partitioning of the design into prioritized build sequences and (b) build-validate-demonstrate iterations.

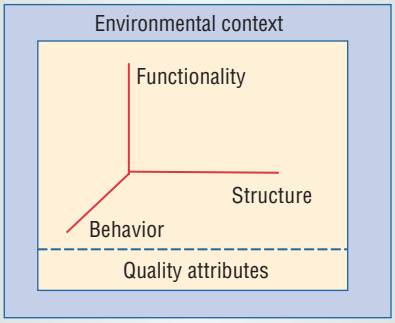

Figure 3. Four dimensions of software: functionality, structure, behavior, and quality attributes. All four dimensions exist within the software's environmental context.

What Is Refactoring?

Refactoring improves the structure of software so that developers can more easily understand, modify, evolve, document, and test it. It can also improve the quality attributes of software components and subsystems or enhance their potential for reuse.

The goal is to make it easier to incorporate reusable elements, add new elements, or accommodate future changes. Developers perform refactoring in small steps. At each step, they perform tests to ensure that structural transformations do not alter functionality or behavior or degrade necessary quality attributes.

Figure A illustrates refactoring in object-oriented development, which can involve

- moving an attribute or a method from one class to another;
- consolidating common attributes or methods in two different classes into a parent class;
- splitting a class into two classes;
- adding an adapter to allow two components with incompatible interfaces to work together; or
- modifying the relationships among classifiers (for example, changing an inheritance relationship into a composition relationship).

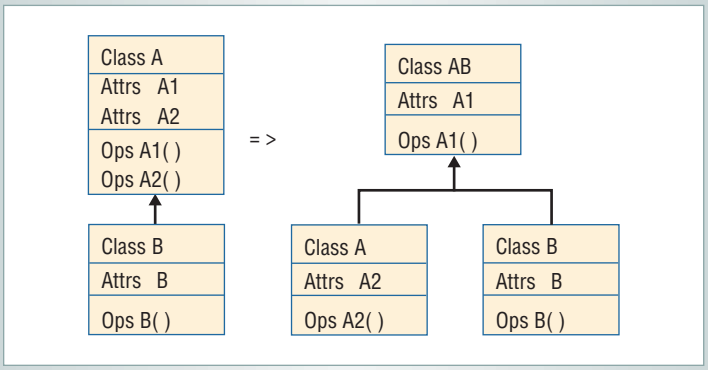

Figure A. Refactoring to avoid inheritance of unneeded attributes and operations in Class B (Attrs2 and Ops2( )). By placing shared capabilities (Attrs1 and Ops1()) in an abstract parent class, developers can avoid inheriting unneeded attributes and operations in subclasses.

stubs and drivers as interfaces, or perhaps using early, incremental versions of other evolving subsystems. System integration can proceed incrementally as intermediate versions of the various subsystems become operational.

As Figure 2b shows, successive incremental builds can overlap. Developers could, for example, start the next version's detailed design while validating the present version. In contrast to the agile approach, the customer is not in the loop, except perhaps to observe some of the weekly demonstrations.

TYPES OF REWORK

Both software development and maintenance involve new work and rework. New work is concerned with adding capabilities to the previous version, while rework involves modifying a previous version.

The basic premise of iterative development is that rework is inevitable because events that occur during software development make once-through development impossible in all but the simplest cases. Acknowledging and accommodating this premise avoids the well-known pitfalls of massive integration, testing, and rework that proponents of the waterfall model—a once-through, sequential process—encounter. Iterative processes, such as agile development and incremental build, provide the ability to gracefully modify the current version of an evolving product while adding capabilities to produce the next version.

As Figure 3 shows, four dimensions characterize software: functionality, structure, behavior, and quality attributes. Software exists in an environmental context from which it receives stimuli and to which it delivers responses.

The software's computational features provide functionality. Structure is both static—programs and data as designed and coded—and dynamic—the interconnections among software entities that come and go at runtime. The behavioral dimension reflects how a system's state changes over time in response to computations and external events. Quality attributes include performance, reliability, safety, security, dependability, ease of modification, and reusability.

Rework results in changes to one or more of these four dimensions. A well-known form of rework is refactoring, in which developers perform semantics-preserving structural transformations, usually in small steps.[1] The "What Is Refactoring?" sidebar gives an example of this practice. Software developers must take care to ensure that structural transformations do not alter functionality or behavior or degrade necessary quality attributes.

Table 1. An iterative rework taxonomy.

Type of rework	Characteristics	Good, bad, or ugly?
Evolutionary	Work performed on a previous version of an evolving software product or system to enhance and add value to it	Good—if it adds value without violating a cost or schedule constraint Bad—if it violates a cost or schedule constraint Ugly—if it smacks of "gold plating"
Avoidable Retrospective	Work performed on a previous version of an evolving software product or system that developers should have performed previously	Good—small amounts are inevitable; better now than later Bad—if it occurs routinely Ugly—if excessive, it indicates a need to revise work processes
Avoidable Corrective	Work performed to fix defects in the current and previous versions of an evolving software product or system	Good—if total rework is within control limits Bad—if it results in patterns of special-cause effects Ugly—if it results in an out-of-control development process

As Table 1 shows, rework can be either evolutionary or avoidable,[2] and avoidable rework can be retrospective or corrective.

Evolutionary rework

Evolutionary rework adds value to an evolving product by modifying one or more of the current version's four dimensions (structure, functionality, behavior, and quality) to provide new capabilities in the next version. Evolutionary rework typically occurs in response to changes in user requirements, design constraints, environmental factors, or other conditions that the developers were not aware of or could not have foreseen when developing the software's previous version. Evolutionary rework is unavoidable because the developers could not have known about or foreseen the changes that necessitate it.

Avoidable rework

In principle, avoidable rework is work that no one would have to do had the previous work been correct, complete, and consistent. That means the previous work satisfies its requirements, is fit for its intended use, and does not contain defects (unlike hardware, software does not break from repetitive use; all defects in software are traceable to human error).

In practice, some amount of avoidable rework is inevitable—even desirable—because insufficient rework could mean that the software developers are not doing enough refactoring, reviewing, or testing. An excessive amount of avoidable rework, however, reduces productivity, increases costs, delays schedules, and demoralizes the development team. It also erodes customer satisfaction because customers tend to doubt the quality of software that has high levels of rework, thinking it might mean the delivered software will have too many defects.

Because a certain percentage of defects will escape the development process, high defect levels during development indicate the likelihood of high levels of customer-discovered defects (one of software engineering's many counterintuitive results).

Retrospective rework. In retrospective rework, developers modify the previous version's functionality, structure, behavior, or quality attributes, or some combination of these because the developers of that version (likely themselves) failed to implement it in a manner that provided foreseeable capabilities needed in the next version. In contrast to evolutionary rework, retrospective rework occurs because developers *knew* about the needs but didn't accommodate them for whatever reason (perhaps because of excessive schedule pressure).

Suppose, for example, that rework involved adding an interface to the next version. The rework would be evolutionary if the customers added the requirement for such an interface after the developers had finished the previous version. The rework would be retrospective (and thus avoidable) if the developers knew the capabilities the new version would require and did not add the interface in anticipation of needing it later on.

Corrective rework. Corrective rework occurs when developers correct defects that result from mistakes previous developers (most likely themselves) made. Defects can be due to mistakes of commission—doing something incorrectly—or mistakes of omission—not doing something they or others should have done. Defects encountered at runtime can result in failures, which could cause system crashes or produce incorrect results or unexpected behavior.

Corrective rework fixes defects detected in the new capabilities or defects in a previous version that the new capabilities expose. In iterative development, mistakes count as defects if developers find them

- while integrating the new capabilities of the next version with the current version or

The line between evolutionary and retrospective rework can be fuzzy.

- while verifying and validating the next version.

Verification is the process of determining that a work product is complete, consistent, and correct with respect to its requirements. Validation determines if a work product is suitable for its intended use. In iterative development, common verification techniques include traceability, peer reviews, and testing, while testing and demonstrations are common validation techniques.

Mistakes that developers make and correct while developing the next versions of their work products do not count as defects. Defects count only in work products that have satisfied their acceptance criteria and are then checked into the official build directory.

IS IT GOOD OR BAD?

In many cases, rework is obviously done in response to changes in user requirements, design constraints, or the operational environment (evolutionary rework). In other cases, rework addresses the failure of the previous version to provide the capabilities needed in the next version (retrospective rework). In still others, it is in response to detected defects (corrective rework).

Fuzzy lines

The line between evolutionary and retrospective rework—and even whether one is good and the other bad—can be fuzzy.

Suppose, for example, that unforeseeable refactoring is required to modify the previous version's structure to make it a suitable basis for the capabilities of the next version; the rework effort is evolutionary and thus unavoidable (good). If developers had foreseen future needs but chose not to include them in the previous version, the rework effort is retrospective and thus avoidable (bad).

The fuzzy part is in the interpretation. Customers might say that they are only clarifying requirements the developers should have understood (retrospective rework), while the developers might maintain that the customer changed the requirements (evolutionary rework).

In other cases, retrospective rework might be the better choice. Suppose reworking version 5 would take less effort than attempting to accommodate version 5's needs while building version 4. In this case, rework to version 4 while developing version 5 is arguably delayed evolution, not retrospective rework.

Collecting and analyzing rework data

The development team and organization must understand the kinds of rework that occur and why, which requires collecting and analyzing rework data. This activity is possible even in ambiguous situations. Because the goal is to identify major trends and to determine the rework's root causes, the rework data does not have to be highly accurate to indicate problem areas.

Collected rework data fits the three categories in Table 1:

- *Evolutionary*. Rework caused by external factors, such as changed requirements, design constraints, environmental factors, or other unforeseeable external events.
- *Retrospective*. Rework to improve structure, functionality, behavior, or quality attributes of a previous version to accommodate the needs of the current version while building the current version.
- *Corrective*. Rework to fix defects discovered in the current version and previous versions during reviews, tests, and demonstrations of the current version.

Four common techniques—informal anecdotes, observation, record keeping, and automated version control—are suitable for collecting rework data.

Informal anecdotes and observation are useful indicators for small agile development teams and incremental-build subsystem teams. Record keeping is essential, although software developers tend to avoid it because they feel it takes time that detracts from their productivity and because they fear that the organization will tie rework to performance reviews.

Publishing sanitized rework data in summary form and basing process improvements on the data will show developers that the time spent in record keeping results in more efficient and effective work processes, thus increasing productivity. Organizations can ameliorate the fear of tying rework to performance by having a nonthreatening person such as a clerical worker ask each developer what percentage of the previous day was spent on new work and evolutionary, retrospective, and corrective rework. That person then records the composite data for the team or project, thus preserving individual anonymity.

Most iterative development projects require some kind of automated version-control system because of the many interrelated, ongoing work activities

and work products. With automated version control, an organization can easily collect data on the effort spent (new work and each kind of rework) for each product version by adding electronic forms that developers complete when they check in the next version's components. The collected data is not associated with the person who enters it, thus ensuring anonymity.

Developers should be willing to participate in collecting rework data once they receive composite, sanitized feedback, see improvements based on that feedback, and learn through experience that managers will not use rework data in performance evaluations (which is impossible if the data is anonymous). Training classes, mentoring, and discussion groups can help developers learn to categorize rework data consistently.

HOW MUCH IS ACCEPTABLE?

For several years, our rule of thumb has been that total rework (evolutionary plus both types of avoidable) is acceptable at 10 to 20 percent of the total effort for each reporting period in an iterative development process. The reporting period typically varies from a week to a month. Weekly analysis of rework data is desirable in a project's early stages. Less frequent reporting and analysis is appropriate once rework stabilizes and remains within the desired range.

In agile development, 10 to 20 percent of the effort is about one to two hours per workday for each developer (or developer pair in paired programming). Rework of 10 to 20 percent in a weekly incremental-build process is roughly a half to a full day of effort per developer during integration, review, testing, and demonstration of the weekly builds.

Spending more than 20 percent of total effort on rework (evolutionary and avoidable) indicates problems in the work processes, work products, or both. Excessive evolutionary rework typically indicates problems in requirements, design, or environmental factors, such as an unstable operating system or changing hardware specifications. Excessive avoidable rework indicates problems in the development process, the tools, the methods, the design strategy, or the developers' skills or motivation. It could also be the result of excessive schedule pressure.

Less than 10 percent rework, on the other hand, could indicate insufficient reviewing, revising, and testing. Robert Cringely addresses the possibility of too much refactoring (www.pbs.org/cringely/pulpit/pulpit20030508.html), but Martin Fowler and Kendall Scott advise organizations to be suspicious if developers refactor less than 10 percent of the code during each iteration.[3]

The purpose of reviewing and testing is to find defects. Software developers are only human; humans make mistakes that result in defects that must be corrected. As a result, the best organizations rarely achieve less than 5 percent corrective rework, so it is reasonable to be suspicious if evolutionary, retrospective, and corrective rework total less than 10 percent of the overall effort.

DEALING WITH THE UGLY

The "How Control Charts Work" sidebar describes the control-chart methodology that quality initiatives such as total quality management and six sigma use. The methodology suggests conducting a root-cause analysis and initiating corrective action when software rework in two of three or three of five successive reporting periods exceeds 20 percent of total effort or is less than 10 percent of total effort.[4,5]

How Control Charts Work

A control chart is a plot of a variable of interest versus time in which the variable has specified control limits. In the 1920s, Walter Shewart developed control charts as a statistical method for analyzing manufacturing defects in the mechanical switching relays used in telephone switching centers. Shewart made the distinction between common- and special-cause effects.

When plotted against time, common-cause effects exhibit random fluctuations around a mean value. In software development, common-cause effects of rework are the noise in the development process that results from factors such as variations in developers' skill levels and motivations, fluctuations in requirements, lack of familiarity with the application domain, and product complexity. One of the goals of quality improvement is to reduce the mean and deviation of common-cause variations.

Special-cause effects are the result of exceptional situations that produce spikes and troughs in a variable of interest that cause the value to lie outside the band defined by an upper control limit (UCL) and a lower control limit (LCL). A process that is in control exhibits random fluctuations about a mean value, and all variations are within the UCL and LCL. An out-of-control process exhibits patterns that violate the UCLs and LCLs for variables of interest.

For manufacturing processes, Shewart used three standard deviations from the mean as the UCL and LCL ($\mu \pm 3\sigma$) and two criteria to indicate patterns of special-cause effects worth investigating:

- two of three successive measurements on the same side of the mean and more than 3σ beyond it, or
- four of five successive measurements on the same side of the mean and more than 2σ beyond it

Determining the UCL and LCL of a control chart for a particular variable of interest is based on pragmatic considerations. We have found that, for most organizations, a UCL of 20 percent and an LCL of 10 percent are both desirable and achievable for software rework.

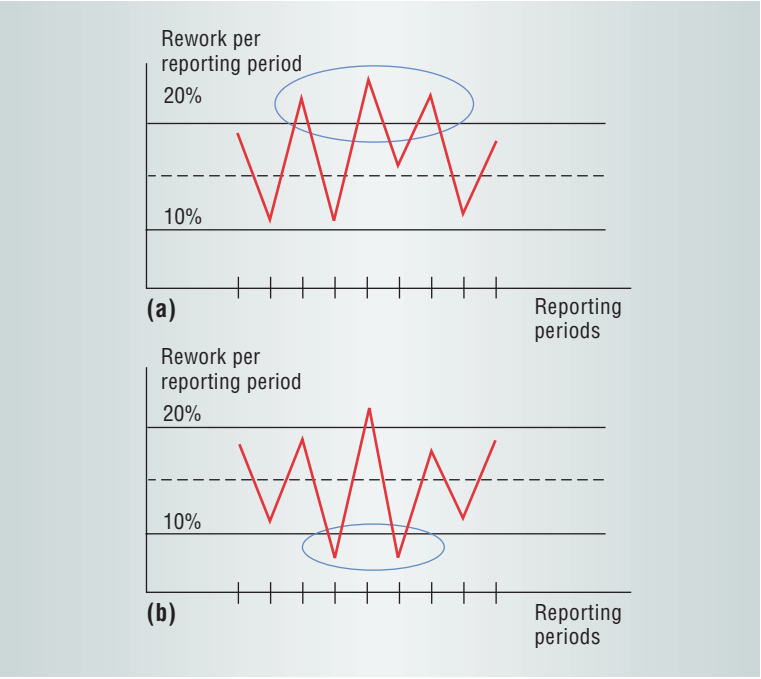

Figure 4. Rework control charts Illustrating (a) excessive rework and (b) insufficient rework.

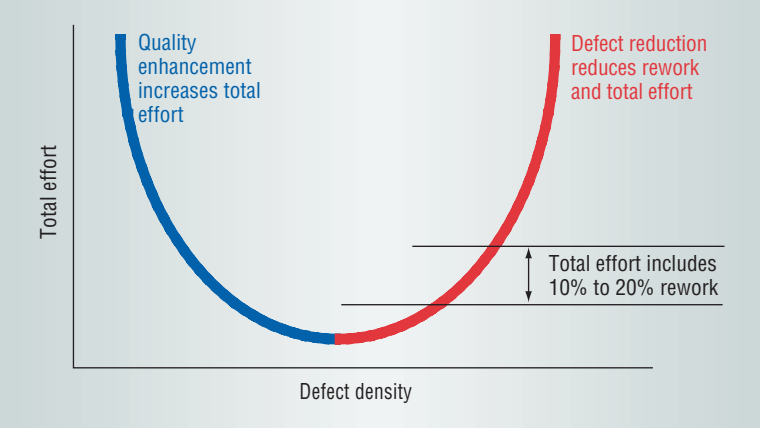

Figure 5. Effort saved by reducing defects, thus reducing avoidable rework.

Figure 4 shows situations in which rework is excessive and insufficient. In Figure 4a, rework is excessive in three of five successive reporting periods, indicating that corrective action is warranted, while Figure 4b shows insufficient rework in two of three reporting periods. The anomaly in Figure 4b might be the result of insufficient review and testing, which warrants corrective action. It might also be the result of exceptional quality in those versions, which warrants determining root causes so that the organization can emulate causative factors in other projects.

PROCESS IMPROVEMENT

Avoidable rework is the bane of software development and modification. Vic Basili and Scott Green reported that 40 to 50 percent of the work on clean-room projects at the NASA Goddard Space Flight Center was avoidable rework.[6] Other reports indicate that rework can amount to as much as 80 percent of total work.[7-9] An analysis of Cocomo II data indicates that most of the savings in effort from improved software process maturity, software architectures, and software risk management came from reducing avoidable rework.[2]

Clearly, identifying and remedying the causes of avoidable rework offers outstanding opportunities for higher quality, improved productivity, increased worker morale, and increased customer satisfaction. Figure 5 illustrates the effort saved by reducing defects (mistakes made), which reduces avoidable rework and thus reduces total effort. A well-known example is Raytheon's reduction of rework from 41 to 11 percent over four years, for a net savings of $16 million.[10]

Figure 5 also illustrates the additional effort that organizations must expend on quality enhancement to meet the stringent requirements of safety-critical and mission-critical systems, which demand the lowest possible defect levels. Process improvements that reduce defects can help offset that additional effort.

The fundamental tenet of process improvement is that superior products result from superior processes. From Walter Shewart's seminal work on quality improvement to process improvements based on the current SEI Capability Maturity Model Integration and ISO Software Process Improvement and Capability Determination process maturity models, investments in process improvement have repeatedly yielded significant returns.[11-13] Process improvement involves changing work activities, organizational structures, roles, relationships, methods, tools, and techniques to improve quality, productivity, morale, profits, and customer satisfaction.

Identifying the root causes of avoidable rework can provide key indicators of the processes most in need of improvement. Improvements in requirements-based testing and increased attention to interface design (two of the most common causes of avoidable rework) might, for example, cut avoidable rework in half. If avoidable rework were 40 percent of total work (not untypical), this reduction would boost organizational performance by 20 percent.

Avoidable rework is unnecessarily high in most software organizations. Identifying and correcting its root causes is a high-leverage mechanism for improving quality, productivity, morale,

profits, and customer satisfaction. Organizations seeking to improve these factors should try to answer questions in six areas:

- How can we collect rework data without intruding into and disrupting the developers' creative efforts more than is necessary?
- What percentage of avoidable rework is retrospective, and for which product attributes does it occur?
- How can we better anticipate the capabilities we will need in the next version so that we can reduce retrospective rework?
- What kinds of mistakes do we make and when do we make them? Could we detect them sooner? How could we prevent them?
- How much corrective rework does it take to fix our mistakes?
- How can we improve our processes to reduce or eliminate these kinds of mistakes?

The answers to some of the questions about avoidable rework may be idiosyncratic to particular projects and organizations, but most are common to all types of organizations and software-intensive systems and products. Whether avoidable rework comes from local circumstances or the failure to apply well-known practices, understanding and correcting its root causes is the key to cost-effective process improvement. ■

References

1. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.
2. B. Boehm and V. Basili, "Software Defect Reduction Top 10 List," *Computer*, Jan. 2001, pp. 135-137.
3. M. Fowler and K. Scott, *UML Distilled*, 2nd ed., Addison Wesley, 1999, p. 28.
4. W. Shewart, *Statistical Methods from the Viewpoint of Quality Control*, Dover, 1956.
5. S. Sytsma and K. Manley, "Common Control Chart Cookbook;" www.sytsma.com/tqmtools/charts.html.
6. V. Basili and S. Green, "Software Process Evolution at the SEL," *IEEE Software*, July 1994, pp. 58-66.
7. C. Jones, *Applied Software Measurement*, McGraw-Hill, 1996.
8. NSF Center for Empirically Based Software Engineering (CeBase), "Summary of the First CeBase e-Workshop," Mar. 2001; www.cebase.org/www/researchActivities/defectReduction/eworkshop1/.
9. S.E. Cross, "Message from the Director," *2002 Annual Report*, Carnegie Mellon Univ., Software Eng. Inst., p. 3; www.sei.cmu.edu/pub/documents/misc/annual-report/2002.pdf.
10. R. Dion, "Process Improvement and the Corporate Balance Sheet," *IEEE Software*, July/Aug. 1993, pp. 28-35.
11. P. Crosby, *Quality Is Free*, Penguin Books, 1980.
12. P. Crosby, *Quality Is Still Free*, McGraw-Hill, 1996.
13. Carnegie Mellon Software Eng. Inst., "Demonstrating the Impact and Benefits of CMMI: An Update and Preliminary Report," Oct. 2003;www.sei.cmu.edu/publications/documents/03.reports/03sr009.html.

Richard E. (Dick) Fairley *is a professor of computer science and director of the software engineering program in the OGI School of Science and Engineering of the Oregon Health and Science University. His research interests include software systems engineering, analysis and design, project management, and practical software process improvement. Fairley received a PhD in computer science from the University of California, Los Angeles. He is a member of the IEEE Computer Society. Contact him at d.fairley@computer.org.*

Mary Jane Willshire *is dean of computer science for the Colorado Technical University system. Her research interests include software engineering, human-computer interaction, and database management systems. Willshire received a PhD in computer science from Georgia Tech. She is a member of the IEEE, IEEE Computer Society, SWE, and the ACM. Contact her at mjwillshire@coloradotech.edu.*