AberdeenGroup

InSight

Programmer Productivity Reconsidered: Reusability Considered Harmful — Refactoring Not

Improving programmer productivity has for many years been the Holy Grail of development technologies, and over the last five years, reusability has been the method of choice. However, evidence increasingly suggests that encouraging reuse in today's toolsets is having little positive effect on productivity — and sometimes even a negative effect. The problem, Aberdeen suggests, is that development tools suppliers and users alike are taking too narrow a view of programmer productivity, mistaking a possible means (reuse) for a goal (faster, repeated delivery of software value-add to the customer). IT buyers should focus on tools that have proven able to deliver order-of-magnitude improvements in programmer productivity in particular situations, improve the end result, or speed upgrade of existing programs — such as refact oring.

Reusability Considered Harmful

In the early 1970s, a classic computer science article appeared, titled "GOTOs Considerec Harmful." A searing critique of then-common programming practice, this piece and the resulting debate eventually led not only leading-edge ISVs but also the bulk of developers and IT departments to change their development "best practices" in favor of GOTO-less "structured programming."

Over the last five years, the idea of "reusability" has enjoyed the kind of cachet that GOTOs once had. The logic is simple and inexorable: Developers need to speed development; an obvious way of speeding development is by writing fewer new lines of code. And the more code is reused, the less new code is written, so to improve programmer productivit y, development toolset suppliers should encourage reusability. All this is true, and most of today's integrated development environments make visible efforts to encourage code reuse—and yet, over the last five years, programmer productivity has not visibly increased.

Aberdeen believes that it is time to call the overfocus on reusability what it is: a harmful waste of IT buyers' and developers' time. Moreover, we believe that it indicates a flaw in our ideas about programmer productivity. Below, we discuss why in our view reusability has failed. Then we suggest a better definition of programmer productivity and more promising alternative new technologies that we believe will deliver better value to the customer in the long run.

The Woes of Reusability

Over the last five years, reusability has been a strong focus of many development toolset suppliers' marketing pitches. At first, suppliers touted the ability to have 60% of the code in any application reused from previous "legacy components." When it became clear that in the real world, these solutions could achieve only 20% reuse at maximum, newer generations have touted the ability (properly used, of course) to drive reuse up to 40%. The key benefit of reusability is supposed to be improved programmer productivity.

Aberdeen boldly asserts that today's efforts to improve reuse have a minimal effect on programmer productivity; in fact, they can actually decrease productivity in particular cases. Reuse affects only coding an d testing — although nothing prevents reuse of design templates, few toolsets implement design reuse or coordinate it with code reuse. The time saved in coding and testing from reuse is lessened by the added effort to find the components to be reused — an effort that becomes more significant as the project becomes larger and the number of components to search for "the right one" reaches the thousands and tens of thousands, with few effective tools for component searching.

To see how this works in the real world, consider a typical object-oriented development project following the "waterfall" method — 50% of the time spent in requirements gathering and design, 30% in coding, 10% in testing, 10% in application deployment. Assuming 20% reusability, reuse affects mostly coding. So at best coding is decreased to 24% of the total project time. If the reused code is free of bugs, and no new bugs are introduced in integrating old code and new, testing is reduced to 8% of the time, for a total savings of 8% of the time of the project — hardly a major gain. And, of course, increasing reusability to 40% would save 16% of the time — still not a major savings.

Now consider the possible negative effects of emphasizing reusability. Developers must learn something about the code available for reuse, adding at worst 5% to the time of the project. Next, developers must search a database of existing code — *for each object class* to be created — to find out whether there is one to be reused, and if that one is the "most reusable" class — adding perhaps 4% to the codin g phase. Finally, testers must verify that the connections between the "reused" classes and the new classes are correct — another 1%. At worst, in other words, we may actually lose 2% to 5% of development time.

Reconsidering Programmer Productivity Itself

Now suppose that reusability did indeed achieve a significant gain in programmer productivity, as defined by lines of code (its typical metric). Does this mean that reusability is a Good Thing?

One problem with that assumption we can see immediately: Reuse of object classes is only one way in which more tested code can be incorporated in a program. Code generation from a design is not "reuse," but because developers do not need to write any lines of code, they are clearly more productive. Writing in a higher level language can improve the speed to production of an application by an order of magnitude in some cases. Upgrading older applications rather than writing new ones or integrating applications together as "composite applications" is clearly more productive than writing them from scratch. Even measuring "programmer experience" as a productivity factor can deliver significant advances in programmer productivity because programmers have their own programs that they amass over time and reuse (although such programs are not counted as reuse).

A second problem is less obvious: focusing on today's most accepted definition of programmer productivity — lines of code per day — ignores the long-term side effects of a particular technique. Reuse can improve, although typically not significantly, software quality, scalability, and flexibility or decrease it. These characteristics of a program, in turn, can affect programmer productivity for follow-on upgrades to the program. Poor-quality code means more fixes after the program is deployed; inflexible or less scalable code means greater difficulties in changing the program when users' needs change.

These considerations suggest an expanded definition of programmer productivity: *the ability to achieve more rapid delivery of customer value-add, repeatedly, over an application's lifecycle.* In other words, we add two new elements to the standard definition of programmer productivity:

1. The goal is not lines of code per day but amount of "value-add" delivered to the customer. Although this goal is necessarily more vague, it focuses on getting to production with the right solution for the customer, rather than getting a lot of coding done to deliver late, poor-quality, inflexible, less scalable code.
2. Developers are not rewarded for achieving results today at the expense of tomorrow. IT must assume that software will need to be upgraded, sometimes even during software creation when requirements change — in fact, IT will spend far longer on upgrading the typical set of software than on creating it.

Better Criteria for Development

The flaws of reusability and the new definition of programmer productivity also suggest new criteria for development methodologies:

1. The methodology should begin its effect in the design stage and "ripple" across all stages of the development lifecycle. Because reusability affects only coding and testing, its effect is cut in half. But note that if a toolset encouraged reuse of "templates" or "patterns" in the design phase, the main value to the developer would be from automatic code generation from the design, not reuse. Table 1 shows the likely programmer-productivity effects of some of today's programmer-productivity technologies.
2. The methodology should focus on not only programmer productivity but also other factors key to immediate customer satisfaction — such as the quality of the software produced and its user-friendliness. Reusability can introduce bugs as well as fix them.
3. The methodology should consider long-term application value and not just value-add "out of the box." Reusability makes no effort to assess the scalability of existing code, nor its "flexibility" in allowing easy, rapid upgrade for new technologies and new users' needs after production.

Below, we describe new technologies that help methodology users meet these criteria: design-driven development to allow "rippling" of productivity gains across the lifecycle, driving testing through the lifecycle to aid both productivity and software quality, and refactoring and "metacode" to aid long-term programmer productivity.

Criterion 1: Design-Driven Development

A good design, driven forward to coding, testing, and deployment, is critical to successful implementation of high-quality, large-scale applications. Developers of these applications must juggle time-to-market, tight budgets, and complex requirements with the result that projects are frequently late, "buggy," and poorly documented. Formal design tools enforce cross-developer adherence to standards and rules for higher quality code, simplify complex requirements, and make project management more predictable. Good design tools do all this and help speed coding and testing as well.

The advent of Web services makes a good design-driven development environment even more valuable. The proliferation of Web service standards makes the design tool's ability to enforce these standards more valuable. The need for a common set of capabilities in any Web service makes the design tool's ability to embed these capabilities at the earliest stages of programming more useful. Design-driven development can move semiautomatically from design to deployment, reducing the complexity of a highly complex Web services development process.

Users should note that many design-driven development tools are not for all purposes. VPEs (visual programming environments), the ability to generate designs from metadata and then generate code from these designs, and formal design tools focused on code are useful for user-interface-driven, data-driven, and function-driven applications, respectively. A formal design tool may very well be overkill for smaller projects with short time to production, whereas a VPE may not scale to large-scale applications. IT buyers should look at a variety of design-driven development tools, such as Rational Rose, Compuware's OptimalJ, and Microsoft's Visual Studio .Net Enterprise Architect.

Criterion 2: Driving Testing Through the Lifecycle

The ability to improve software quality by testing across the full lifecycle of the application is a crucial value-add to enterprises' software quality efforts. Until now, testing has been done primarily during development. However, today's applications typically grow rapidly and are deployed over the Internet, so IT organizations need to plan for potentially continuous deployment. Moreover, the typical systems management solution that detects unanticipated problems is not connected to the application development process and therefore cannot deduce the causes of development-related problems. As a result, increasingly complex applications generate a stream of execution problems that systems management solutions are increasingly ill equipped to solve.

Today's most sophisticated testing solutions span the application lifecycle, including code analysis in the development phase, functional and performance testing in the QA phase, user experience monitoring in the preproduction/deployment phase, and application-service-level monitoring in the production/maintenance phase.

Users can employ these technologies to accomplish the following tasks:

- Accelerate testing — Test management methodologies let testers schedule tests unattended, thereby making testing possible 24×7 and allowing testers to focus on analysis, not test execution; execute all required testing tasks — functional, site, data comparison, performance, and load — from a single user interface; and generate detailed logs of tests and test results.
- Communicate effectively between testing and development — Integration of test management and defect tracking means that a tester can send a failed test and all relevant data rapidly to development. Moreover, it minimizes misunderstandings and the possibility of losing important pieces of test scenario data.
- Match testing to business priorities — By integrating requirements management and test management, users can ensure that tests focus on areas of the application that are especially important to the organization, speeding implementation of the parts of the application that matter.

Automated testing — executed effectively — can reduce application testing cycle costs dramatically, as much as 50% to 75% compared with manual testing process costs. Because bugs detected later in the development lifecycle require much more time and effort to fix, moving testing forward to design can cut an additional 20% off the coding and testing phases — and improve user satisfaction. A wide variety of cross-lifecycle testing tools exists today, including CA's, Compuware's, and Mercury Interactive's offerings.

Table 1: The Programmer-Productivity Effects of Today's Techniques

Technique	**Lifecycle Stages Affected**	**Likely Maximum Time Saved (%)**
Object-oriented programming	Design (+), coding, testing	10%-20%
Automated deployment	Deployment	10%
Higher level coding		
— transactional (4GL)	Design (data-driven), coding, testing	50%-70%
— VPE	Design (GUI-driven), coding, testing	50%-80%
— standards-based	coding, testing	10%-20%
Reusability	Coding, testing	5%-15%
Open-source programming	Coding, testing	20%-30%
Components	Design (+/-), coding, testing (small)	5%-15%
Infrastructure solutions	Design, coding, testing	50%-80%
Extreme programming	Design, coding, testing	10%-30%

Source: Aberdeen Group

Criterion 3: Refactoring

Refactoring is a technique for re-architecting code (and databases) incrementally to make it easier to upgrade (and understand) in the future. Refactoring is not only a developer technique but also a set of tools that can semi-automatically survey an application and refactor it. Refactoring is therefore eminently suited for inclusion not only in toolsets supporting an agile programming approach but also in any development solution. Users of refactoring should note that, as previously mentioned, it is much harder to refactor — or change — the data accessed by a program than it is to change the code in a program.

Refactoring has a slightly negative effect on project time in initial application development. From then on, however, refactoring can have a major effect on programmer productivity. Moreover, refactoring can help users avoid "software sclerosis," in which legacy applications become more and more difficult to upgrade to new technologies and hence more and more costly to maintain. Few true refactoring tools exist today, although new suppliers such as WebPutty are beginning to offer refactoring capabilities.

Criterion 3: 4GLs, Metacode, and OmniCompetence

Beyond a certain size, applications inevitably have to deal with large amounts of data, which, in turn, call for databases. Interfacing code to data is among the toughest programming jobs at any time, and the "object-relational mismatch" has only made things worse.

Aside: Components Considered Helpful

The IT buyer should note that although reusability itself may have minimal effect, the use of component libraries acquired from component suppliers such as Component Logic or Component Library (or any major development toolset supplier) is likely to have a major positive effect on programmer productivity and software quality.

These components provide a "software infrastructure" that allows higher level programming — and higher level programming such as 4GLs has long proven its ability to improve productivity (by contrast, "reusability" focuses on components developed inside the enterprise, which typically cannot be used as infrastructure). This is especially true of business-logic components — although some components such as EJBs whose aim is to hide data access may have performance drawbacks. The IT buyer should therefore look especially at component libraries that allow the developer to code at a higher level.

In large application development projects of the late 1980s and early 1990s, 4GLs sped up time-to-production by focusing on simplifying "transactional" code. Typically, a 4GL did this by providing English-language-like methods that developers used to code queries, updates, and other typical data-access operations — including handling transaction concurrency below the programming level.

The 4GL is an example of *metacode*, as coined by Simon Williams in his book, *The Associative Model of Data* (Boston: Lazy Software Ltd., 2002, p. 139) — code that operates on data at the metadata level, without needing to know the details of how the corresponding data is stored on disk. Where the purpose of a program is to access and display data, metacode can be combined with *data-driven design* — an approach that generates code directly from the metadata stored in a database's data dictionary.

For applications that need to access multiple data sources — such as enterprise portals or business process integration solutions — it is possible to simplify data access still further via "omnicompetence" (another Simon Williams concept). Omnicompetent code can operate on metadata from multiple databases, allowing the same code to be reused no matter what database is accessed. Enterprise information integration (EII) solutions are examples of omnicompetent solutions; Lazy Software's querying tool is as well.

IT buyers should favor tools that do not require developers to concern themselves with the storage details of database data. At the very least, toolsets that support standard SQL, data-access wizards, 4GLs, and data-driven design functionality are far better. These toolsets can yield scalability, programmer productivity, and flexibility all at once. Although Lazy Software and EII suppliers are among the few that offer omnicompetent programming tools, tools that allow generation of designs from data-dictionary metadata (e.g., Oracle) are widely available.

Aberdeen Conclusions

It is time for development toolset suppliers and buyers to emerge from the "larval stage" of reliance on outdated buzzwords such as "reusability." Aberdeen believes that the first step is to redefine programmer productivity to aim at real user needs — faster, repeated delivery of real customer value-add. The second step is to find technologies that deliver major improvements in achieving this new goal — technologies such as design-driven development, refactoring, and cross-lifecycle testing. Reusability is a God That Failed; effectively used, these new technologies have a much better chance of being Tools That Succeed.

- Wayne Kernochan

This *InSight* is one of a five-part series on the challenges and opportunities facing today's software development strategists. The following are titles and links to the other *InSights* in the series:

Developing the Enterprise's Software Portfolio: A Long-Term Strategy to Avoid "Software Sclerosis"

Agile Programming, Extreme Programming, and Refactoring: Not Just Another Development Fad

Existing Application Upgrade Is Key to Future E-Business Strategies

Coming soon: Software Quality: Test, Automate, Satisfice

Analyst Name: Wayne Kernochan
Practice Area: Enterprise XML, Database Development & Development Tools

Aberdeen Group
260 Franklin Street, Suite 1700
Boston, MA 02110-3112
www.aberdeen.com

Phone: (617) 723-7890
Analyst Bio: Wayne Kernochan