White Paper

Ranking IT Productivity Improvement Strategies

Flashline *Achieving a Better Return on Software™ Series*

FLASHLINE® return on software™

Key Message

Software development and maintenance productivity investments must be considered as a portfolio of strategies strongly aligned with overall corporate business goals.

Challenge

Possible productivity improvement strategies:

- Address different levels of improvement
- Deliver results in short, medium or long term
- Have varying degrees of risk

Levels of Productivity Improvement

- Individual
- Project
- Departmental/ Business Unit
- Enterprise

Productivity Improvement Strategies Considered

- Inspections
- Personal Software Process (PSP℠)
- Team Software Process (TSP℠)
- Rational Unified Process (RUP)
- Capability Maturity Model (CMM)
- Agile Development
- Extreme Programming (XP)
- Reuse
- Component-Based Software Development

Management Risk Issues

- Scope of effort
- Organizational readiness
- Technical maturity

Martin Griss
Flashline Software Development Productivity Council

The short-term bias in today's ROI mania may lead to poor choices. What to do, what to do?

IT directors and application/product development managers face numerous challenges in selecting and funding strategies for productivity improvement in software development and maintenance. They must choose from a large set of apparently competing new processes, tools, and technologies. Some of these solutions are touted as 'silver bullets,' applicable to all projects, while others are applicable only to projects of a certain type or size. Some seem to work with small teams of empowered developers; others seem better suited to large, distributed organizations. Some solutions promise low-cost, low-risk improvements that can be widely and quickly introduced across the company, while others are more costly, engender greater risk, and must be tested and refined in a pilot before wider deployment. Furthermore, various combinations of these solutions can be either mutually complimentary or potentially incompatible.

The purpose of this paper is to explain an effective way of ranking IT productivity strategies. Some of these strategies will affect individual or small team methods and tools, while others require the collaboration of multiple project teams to achieve the promised benefits. The key message is that one needs to strongly align the software improvement goals with the overall business goals, and that the strategy should be to create and carefully state and manage a portfolio of short, medium, and long-term improvement efforts, rather than focusing only on short-term tactical improvements.

Business and risk-driven decision criteria

In order to effectively evaluate the various productivity-improvement strategies it is necessary to weigh and prioritize a mix of criteria in order to provide a balanced scorecard.

First and foremost, we must consider the specific business goals and their urgency. Without a well-articulated business goal it will be difficult to motivate and sustain the investment, especially for a longer-term, strategic intervention. There are various common goals, typically:

- Improve business and IT agility
- Improve quality
- Reduce development and maintenance time
- Reduce development and maintenance costs

The prioritization of goals has a significant impact on the shape of the initiatives. For example, when considering reuse as a productivity strategy, if "time to market" has priority over "cost" one will be willing to invest in many more reusable assets than if cost is the driving concern – even if there is a possibility that some of these assets may not be immediately used.

After prioritizing goals we must then consider how quickly we need to see a visible benefit, and how widely to adopt the necessary changes across the organization in order to achieve the desired goal. This leads to consideration of risk management and incremental piloting and adoption strategies, which can reduce risk, but also can reduce the pace and size of the payoff.

Figure 1 - Software Development Productivity Improvement Strategies

Productivity Improvement Strategy	Orientation						Characteristic Features and Focus
	IND*	Project			BU*	ENT*	
		S	M	L			
Personal Software Process (PSP SM) [7]	•						**Build individual skill and process; focus on individual process, design, estimating and review discipline; self-calibration. CMM compatible.**
Team Software Process (TSP SM) [8]		•					**Extends PSP with specific roles, risk management to build quality products on cost and schedule. CMM compatible.**
Inspections [5,10,17]	•	•					**Systematic approach to cost-effective review of architecture, design and code.**
Rational Unified Process (RUP) [13]		•	•	•			**Model driven architecture, design and development; customizable framework for scalable processes; defines many roles, artifacts and incremental/ iterative workflows. Some reuse-oriented extensions, such as RSEB[2] and EUP[16].**
Agile Development [11,12,17] **Extreme Programming [12,17](XP)**		•	•				**Include most of high-collaborative, incremental test and feature-driven development, effective pair-programming. Focus on people and communication, incremental delivery.**
Open Source [14]					•	•	**Community building methods and technology, informal reuse, strong architects/gatekeepers roles; "free" beta-testers/co-developers; quality through "thousands of eyes."**
Capability Maturity Model (CMM) [6]	•	•	•	•	•	•	**Builds organizational capability; focus on repeatability and predictability of process, defines key process areas that are addressed, provides 5-level assessment model and incremental adoption guidelines. (PSP and TSP are individual and small team instantiations of CMM guidelines).**
Reuse [2,9,15,16,18] **Product Line Development [2,18]** **Component-Based Software Engineering (CBSE[18])**		•	•	•	•	•	**Proactive supporting, management, architecting and designing asset reuse across multiple provider and user project teams.**

*IND = Individual, BU = Business Unit, ENT = Enterprise

Key risk management issues include scope, organizational readiness, and the technical maturity of the proposed change. Organizational readiness is a big issue; prior and ongoing experience with (competing) initiatives and interaction with other initiatives (software or otherwise) will determine how much resistance must be overcome and how much (extra) learning and adoption time is needed.

As you can see, one must carefully consider the tradeoffs between risk management, incremental adoption, desired size and pace of visible improvement, and the technical maturity of the approach. Among other things, this will determine how big a pilot (if any) will be needed to validate and fine-tune the proposed practice for the chosen portion of the organization.

Treat the decision problem as investment portfolio management

One cannot simply pick a single productivity-improvement solution based on short term ROI and overall expense. Doing so ignores the much greater potential of longer-term programs like reuse or overall process improvement. It makes more sense to view the issue as an investment in a portfolio of improvement projects, with a periodically rebalanced mix of time-scale, scope, and risk. A simple yet effective risk-weighted strategy for this would distribute the proposed projects into a rough two-dimensional matrix of *short / medium / long* time-scales against *high / medium / low risk*, and adjust the effective anticipated pay-off for each project by a correction factor for risk and time-based discount, such as Net Present Value. We would then apply this to a variety of projects, periodically examining and rebalancing the portfolio, confident in seeing useful, strategic improvements[f1].

Not all improvement projects address the same issues

Productivity Improvement Investment Portfolio

- Aligns with corporate goals
- Balances time-scale, scope, size of investment, anticipated return, and risk
- Periodically re-balanced

A quick review of a few basic statistics about software projects is useful in considering sources of potential improvement.

IT managers are often fearful of investing in improvement projects and technologies because they do not always deliver the promised improvements. However, those same IT managers also know that too many business-critical software development projects fail to deliver the promised functionality on deadline, if they ever deliver at all. Standish Group[f2] research reveals that less than 30% of projects deliver results on their promised schedule and cost, many of these with reduced functionality. Still others significantly overrun schedule or budget. In the face of these statistics, choosing to do nothing to improve the situation is in fact more risky than choosing to launch one or more improvement efforts.

Avoidable Rework

- In unrelated projects, caused by
  - Misunderstanding
  - Changes in requirements
- In related projects, caused by
  - Redundant development
  - Duplicate discovery and repair of defects
  - Wasted maintenance

An analysis of many projects reveals that "avoidable rework"[f3] is a significant problem affecting quality, cost, and schedule. Furthermore, the longer rework is delayed in the development cycle the more it costs and the longer it takes to complete. For single, unrelated projects, avoidable rework is most often caused by misunderstanding of and/or changes in requirements. For groups of related projects (product lines and application families) the impact of rework is compounded by redundant development, duplicate discovery and repair of defects, and wasted maintenance.

The "Top-10 factors influencing software quality and productivity," developed by Boehm and Basili[1], reminds us that finding and fixing software problems after delivery is often 100 times more expensive than finding and fixing them during the requirements and design phase. Their research indicates that 40% -50% of the effort on traditional software projects is spent on avoidable rework. Their analysis shows that about 80% of this avoidable rework comes from 20% of the defects, most of which are located in just 20% of the modules. This suggests that improvement efforts should be focused on early

f1. For an example of this approach see Grady's "A Portfolio of Investment Choices"[5], pg 68

f2. The Standish Group, 2001 – less than 30% of business enabling IT initiatives are on time and on budget, often delivering less than 70% of promised functionality; about 50% are significantly off target and the rest fail totally. Many of the failures are some 63% over schedule and 45% over budget. Large projects in large companies do even worse.

f3. Not all rework is avoidable; requirements, understanding and the technical and business environment do in fact change while the project is underway; while (more) agile methods such as the spiral lifecycle, incremental/iterative development and extreme programming go a long way to shortening cycles and improvement alignment with changing customer needs, some rework is unavoidable.

f4. In general, time and cost are related in a non-linear way, as shown by the well-known Putnum and CoCoMo estimation formulas.

f5. Adopting incremental/iterative development, XP and Agile methods reduce the feedback loop, partially correcting requirements mismatches and other defects earlier. Pair Programming and Model With Others replaces of the distinct peer reviews of code and models with ongoing reviews done during the team development[17].

detection and prevention of defects, and on finding and reducing the number of "error-prone" modules, using a combination of reviews, testing, careful design, and the implementation and reuse of high-quality modules. The evidence further suggests that peer reviews (especially perspective-focused reviews) catch 60% -75% of defects, while disciplined personal practices can reduce defect introduction rates by up to 75% [f4].

Grady[5] offers a more detailed set of cost factors for development (see Figure 2), as derived from studies at HP and elsewhere. Maintenance is treated as yet another iteration of development, preceded by knowledge recovery. In Grady's model, performing inspections on requirements, design, and code will reduce some of the rework in all areas, at the expense of some extra work. Grady estimates a maximum cost saving of no more than about 10% short term and 10% longer term (with similar time savings[f5]). If only code inspections are performed (as is the case in many organizations) the cost savings will be significantly less, perhaps only 3%. Fortunately, inspections can be taught quickly and introduced with relatively low risk.

Synergy vs. competition of potential improvements

Often, decision makers feel that they have only a single choice among apparently competing solutions. For example, people often ask "should we choose CMM, reuse, or Extreme Programming?" or "should we choose RUP, PSP/TSP, or Extreme Programming?" or "should we choose reuse or inspections?" In fact, several of these techniques can be synergistic, or can be addressed as important precursor steps to others; some can be explored as part of several parallel pilots.

Categories of Improvement

- Organizational (process)
- Project (methodology)
- Architectural (technology)

CMM primarily addresses organizational/process improvement; inspections generally address project/method improvement; and reuse addresses architecture/product/technology. There is potential overlap and synergy in several places. In particular, as certain organizational aspects of reuse are addressed[3, 4], a more holistic integration of people, process, and technology improvements is accomplished. Method/process improvements such as RUP or Agile Development can harmoniously interact with reuse, especially if customized, staged, and positioned appropriately[2, 16].

Figure 2 - Management Cost Model

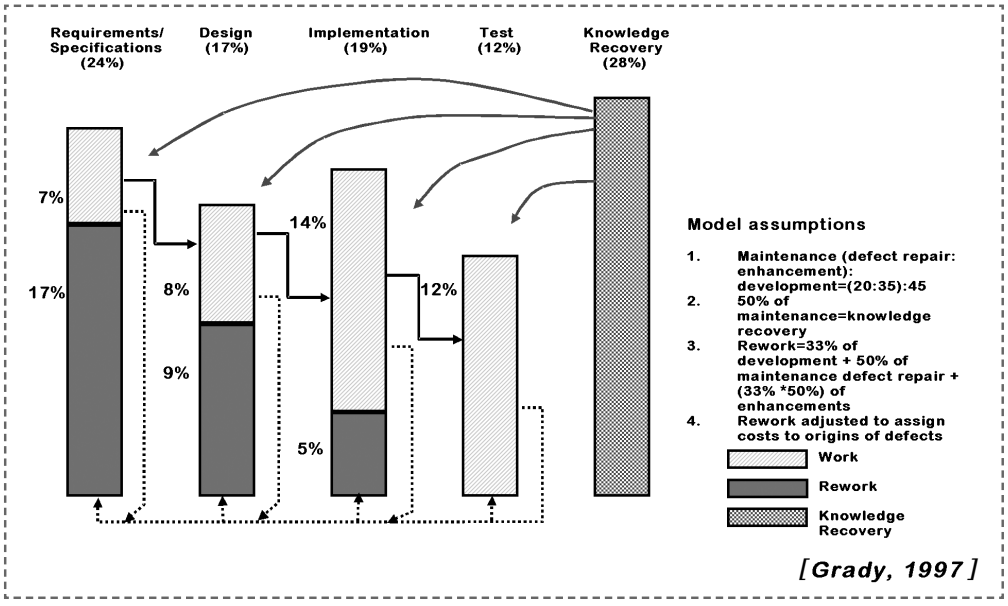

Figure 3 - Summary and comparison of approaches

Approach	Benefits and Potential Improvements	Risks	Timescale	Costs
Inspections	Catch defects early, avoid rework. Important to do more than just code review for full, early benefits. Can reduce overall cost by 3-20%, reduce later testing and rework time.	Low, but some resistance to extra busy work.	Quick adoption, early benefits.	Small amount of training, can introduce incrementally or in parallel. May add extra time earlier in process. Some reviews redundant with Agile approaches.
RUP	Improve capture and analysis of requirements, define and articulate architecture, develop detailed specification of interfaces, systems and components; synergizes well with Designed Reuse. Provides a well-understood framework for customized/standardized methods and processes of various levels of ceremony.	People may resist extra modeling and documentation work, creation of models, keeping models	Can be learned in several week long classes, but takes many months to become skilled	Tools and training classes, and especially a period of mentoring to help master the skills
Reuse - Facilitated (Open Source)	Reduce time to market and improve costs, but relatively low to medium levels of reuse (~5-15%) and consequent modest benefits. Can save development costs by about 5-10%. Open Source approaches inside or inspired by an enterprise are a form of Facilitated Reuse[15]	Low risk to introduce toolset and policies. May need some incentives and education to help expand use.	Can be started in a few weeks, with a few months to get sufficient assets into the repository.	Low cost to install repository, initial assets, start evangelizing or incenting (re-)use; some additional cost to harvest and package assets.
Reuse - Managed	Improve quality, reduce time to market, and improve interoperability. Cost savings of about: 10-15%, some reduction in development time, improve quality by 1.5-3x.	Will encounter more resistance and need more process and education as well as some organizational changes.	To ensure that chosen assets used and significant cost and quality benefits result, must institute process policy and review steps to ensure adequate control.	Takes time, training and process changes to get people to comply with managed asset policy; extra work to re-document/re-engineer assets. Need support team to support repository and manage assets and process.
Reuse - Designed	Greatly improve quality, reduce time to market, improve interoperability, get high levels of reuse (~ 60%-90%) Can reduce cost by 15-35%, time by 2-3x and improve quality by 5-10x, reduce overall long-term maintenance costs	Need to a relatively stable domain and (predictably) "bushy" product line to ensure maximum benefits from larger initial investment in architecture, design, frameworks and generative. Domain or technology could change before payback achieved. Proceed incrementally.	May take several typical project cycles (2-3 years) to build enough high-quality assets to produce high-levels of reuse and dramatic improvements.	Increased cost to do some domain engineering, train architects, component developers, incrementally reengineer assets or develop more reusable ones from domain models
Agile Development/XP	Improve customer orientation, quality	Typically, pilot with one project, takes some months before all resistance overcome to new way of work	Some resistance and disbelief that should work. *May take several iterations to convince people.	Training, experience consultant to help start-up.
CMM improvement (PSP and TSP are individual and small team instantiations)	Improve predictability and productivity, reduce defects. Improved discipline and stabilized configuration management synergize with reuse program to reduce cost of reuse.	May trigger resistance to "increased process"; takes a while to yield visible, measurable benefits.	Typically takes 12-18 months per CMM level; PSP and TSP can be taught and adopted in several months.	Significant training and reinforcement needed to ensure that people stay the course and follow the process. PSP and TSP are useful skill improvement experiences.

As an example, consider the adoption of peer reviews, inspections and/or perspective-driven reviews. These are relatively low-ceremony, low-risk, easy-to-learn processes that can dramatically improve the quality of various software artifacts — such as architectures and designs — early in the development cycle, well before code is produced. Small teams can adopt these techniques independently; over time, the techniques can be spread to the entire organization. They also help reduce rework and testing time. If your organization is not engaging in some form of these reviews, you should seriously consider their introduction for the immediate benefits they are almost guaranteed to deliver. Later, you will probably have to modify the specific process steps and perspective checklists as your organization moves to more encompassing reuse processes and/or Agile development. But having already laid the foundation you will have gained early wins. Agile development methods include or supplant some inspections and peer reviews [17].

Observations and summary

In summary, it is important to develop and refine an overall improvement strategy that carefully selects a set of short, medium, and long-term improvements, and actively schedules and coordinates the next steps in and manages the flow of the elements in the portfolio.

Figure 3 summarizes some of the potential benefits for each of the approaches. The three modes of reuse mentioned — Facilitated, Managed and Designed — vary in the levels of proactiveness involved in both the management of the reuse process, and in the design and management of reusable assets. [15]

I have seen variants of this business-goal-driven improvement portfolio at several companies, including Hewlett-Packard®, Agilent™ and others. For example, in one HP business-unit with several divisions, the business goals were to improve overall software quality, reduce cost, and enhance interoperability. This led to a portfolio that involved a significant investment in a component-oriented architecture, improved testing and code reviews, and reuse-oriented design and management of components.

In a multi-divisional instrument business unit, the crucial goal was to significantly decrease time to market at the same time that the variety of products was increasing. In nine months they developed an instrument architecture and designed supporting components. Over the space of several years they expanded the number of components, increased the use of object-oriented modeling and implementation of new components and significantly reduced time to market, shrinking new product development time from over 18 months to as little as 5 months, with levels of reuse ranging from 25% to over 50%.

Finally, another systems business unit chose to improve its overall software predictability and quality, and reduce the amount of duplicate or near duplicate code, by establishing a multi-year process improvement initiative leading to CMM 3 (and higher), as well as a systematic, architected component reuse project that started by mining and reengineering key components from several existing and ongoing projects, and then moved to more reuse-focused architecture and components designed for reuse.

About The Author

Martin Griss is highly regarded as one of the leading authorities on software reuse, having spent nearly two decades as Principal Laboratory Scientist at Hewlett-Packard and as Director of HP's Software Technology Laboratory. At HP, Griss was widely known as the company's "Reuse Rabbi" where he helped introduce software reuse into the company's development processes and defined a reuse consultancy. His research has covered software reuse processes and tools, software agents, software factories, component-based software engineering, and software-bus frameworks. Griss has written extensively on software reuse for a number of industry publications, and is co-author of the widely read "Software Reuse: Architecture, Process and Organization for Business Success." He is currently an Adjunct Professor of Computer Science at the University of California, Santa Cruz, and at the University of Utah.

Mr. Griss earned a B.Sc. from the Technion in Israel in 1967 and a Ph.D. in Physics from the University of Illinois in 1971.

About the Flashline Software Development Productivity Council

Martin Griss is a member of Flashline's Software Development Productivity Council (SDPC), an unprecedented assemblage of knowledge and experience covering the broad spectrum of software development areas including software engineering process and environments that affect today's corporations. The members of the SDPC are individually recognized authors and consultants in a wide range of subjects covering the entire software development lifecycle and its relation to the corporate bottom line. Their expertise guides the evolution of Flashline's products and services. Through direct consultation with Flashline customers the SDPC aids in the realization of the goal of greater return of software.

References

1. Boehm, B. and Basili, V. Jan 2001, Software Defect Reduction Top-10 List. IEEE Software.
2. Jacobson, I., Griss, M. and Jonsson, P. 1997. *Software Reuse: Architecture, Process, and Organization for Business Success*. Addison-Wesley.
3. Griss, M. Mar 1998. CMM as a Framework for Systematic Reuse - part 1. Object Magazine.
4. Griss, M. June 1998. CMM, PSP and TSP as a Framework for Adopting Systematic Reuse - part 2. ObjectMagazine.
5. Grady, R. B. 1997. *Successful Software Process Improvement*. Prentice-Hall.
6. Paulk, M.C., et al. July 1993. Capability Maturity Model, Version 1.1. IEEE Software. pp.18-27.

   Paulk, M. C., et al. 1995. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley. See also more recent SW-CMM and CMMI: http://www.sei.cmu.edu/cmm
7. Humphrey, W. S. 1996. *Introduction to the Personal Software Process* (SM). Addison-Wesley. See also http://www.sei.cmu.edu/tsp/
8. Humphrey, W. S. 1999. *Introduction to the Team Software Process*(SM). Addison-Wesley, 1999. See also: http://www.sei.cmu.edu/tsp/
9. Lim, W. C. 1998. *Managing Software Reuse: A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components*. Prentice-Hall.
10. Gilb, T., Graham, D. and Finzi, S. 1993. *Software Inspection*. Addison-Wesley. See also: http://www.result-planning.com/ and http://www.mfagan.com
11. Highsmith, J. 2002. *Agile Software Development Ecosystems*. Addison-Wesley. See also: http://www.agilealliance.org
12. Beck, K. 1999. *Extreme Programming Explained: Embrace Change*. Addison-Wesley. See also: http://www.xprogramming.com/
13. Kruchten, P. 2000. *The Rational Unified Process: An Introduction (2nd Edition)*. Addison-Wesley. See also: http://www.rational.com/products/rup
14. Feller, J., Fitzgerald, B. and Raymond, E. S. 2001. *Understanding Open Source Software Development*. Addison-Wesley. See also: http://www.opensource.org/ and http://www.osdn.com/
15. Griss, M. (Available Q2 2003). Reuse Comes in Several Flavors, Flashline, Inc.
16. Ambler, S. W. 2002. *Enterprise Unified Process*. Ronin International.
17. Ambler, S. W. Aug 2002. *Validating Agile Methods*, The Cutter IT Journal.
18. Heineman, G. T. and Councill, W. T. 2001. *Component-Based Software Engineering – Putting the Pieces Together*, Addison-Wesley.

FLASHLINE®
return on software™

■ 1300 East 9th Street, Suite 1600, Cleveland, OH 44114 ■ T: (216) 861-4000 ■ F: (216) 861-1861 ■ www.flashline.com