

Making RAD Work for Your Project

--Extended version of
March 1999 IEEE Computer column --

B. Boehm
March 1999

USC-CSE-99-512
IEEE Computer
March 1999, pp. 113-117

Making RAD Work for Your Project

-- Extended version of March 1999 IEEE Computer column --
Barry Boehm, USC
March 1999

Abstract

A significant recent trend we have observed among our USC Center for Software Engineering's industry and government Affiliates is that reducing the schedule of a software development project was becoming considerably more important than reducing its cost. This led to an Affiliates' Workshop on Rapid Application Development (RAD) to explore its trends and issues. Some of the main things we learned at the workshop were:

- There are good business reasons why software development schedule is often more important than cost.
- There are various forms of RAD. None are best for all situations. Some are to be avoided in all situations.
- For mainstream software development projects, we could construct a RAD Opportunity Tree which helps sort out the best RAD mixed strategy for a given situation.

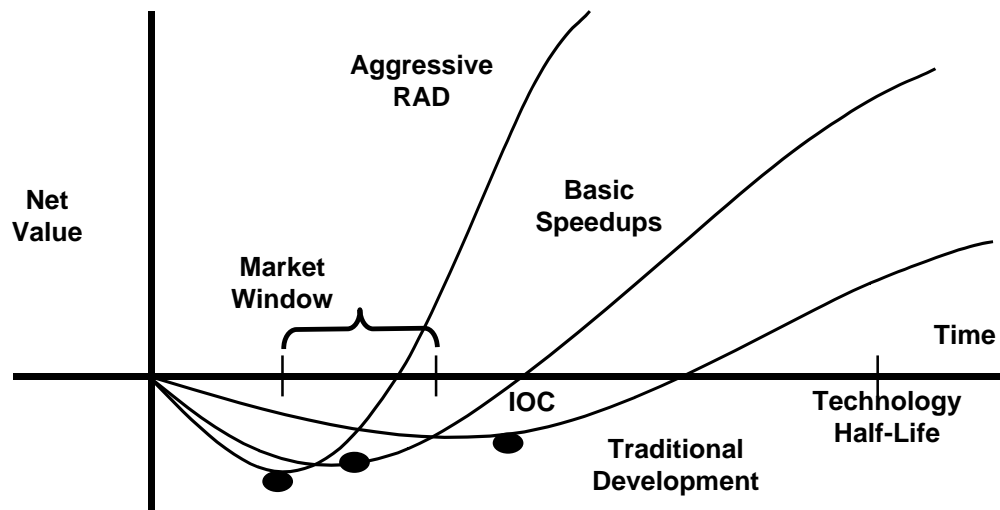
RAD Business Case

The main business reason for RAD derives from the fact that any investment in software development is done to reap a downstream benefit flow in higher profits, reduced inefficiencies, better public service, etc. Clearly, the sooner the initial operational capability (IOC) is ready, the sooner you can enjoy the benefit flow. Three additional features heighten the business justification (see Figure 1):

- IOC's earlier in a market window will capture more market share, revenues, and profits.
- Earlier revenues and profits are paid in less-devalued dollars.
- Ever decreasing technology half-lives reduce the time window within which the benefits, revenues, and profits can be realized.

Thus, investing in RAD strategies has strong business value, even in the conservative care in which they increase development cost (generally, they don't).

Figure 1. Net Value (Benefits-Costs) of RAD Investments



RAD Forms: GRAD, CRAD, FRAD, and DRAD

There are a number of RAD forms in which to invest. Some are best for some situations, and not for others: Generator RAD (GRAD), Composition RAD (CRAD), and Full-System RAD (FRAD). The most frequently used form on ambitious systems is not recommended in any situation: Dumb RAD (DRAD).

Generator RAD involves the use of very high level languages such as spreadsheets, business fourth generation languages such as IDEAL or Focus, or other domain-specific languages for such domains as finance, industrial process control, or equipment testing. Portions of these domains are sufficiently bounded and mature that you can simply use the language to state what information processing capability you want, and the generator assembles, integrates, and executes the pre-written components necessary to provide you with the capability. With GRAD, individual users with relatively little programming expertise can generate an application in a few hours to days that would have taken several programmers several months to produce 20 years ago.

The main problem with GRAD is scalability, as the generators often buy ease of expression and assembly at the cost of efficiency. This is not a problem for small applications, but has led to major disasters for large systems. For example, the New Jersey Department of Motor Vehicles' registration system was written in IDEAL to meet a rapid schedule, but was so slow that at one point over a million New Jersey vehicles were driving around with unprocessed license renewals [1].

Composition RAD (CRAD) involves the use of small "tiger teams" to rapidly assemble small to medium-large systems based on large components (networking packages, GUI builders, database management systems, distributed middleware) and applications-domain class libraries. Typically, a 3-5 person tiger team can compose an

application in 3-5 months that would take a team of 15-20 programmers a year or two to do using traditional methods.

CRAD is more scalable than GRAD, but can still have serious scalability problems on large systems. A recent example was the London Ambulance System [2], which used Visual Basic for rapid composition. The system went live on a Monday morning, and was so backed up by Tuesday afternoon that it was shut down. After about a week of attempted performance improvements, it was abandoned forever.

Another frequent problem to beware of is that a number of independently-developed GRAD and CRAD systems will have difficulty interoperating with each other and with the rest of your applications, with incompatible class libraries, GUI's, and infrastructure. The best remedy for this problem is to develop an enterprise architecture of standard inter-application interfaces; common data and class definitions; GUI guidelines; and preferred componentry where necessary.

Some treatments of RAD confine its use to GRAD and CRAD approaches. But for larger systems not well suited to GRAD and CRAD, there are a number of effective techniques for reducing cycle time. These Full-Scale RAD (FRAD) techniques are discussed in more detail below.

Avoiding Dumb RAD (DRAD)

Before discussing FRAD techniques, let us dispose of the major RAD form to be avoided: Dumb RAD (DRAD). This form is most frequently encountered when a decision-maker sets an arbitrarily short deadline for the completion of a software project, without having done any analysis of its feasibility. This can happen for political reasons ("It's got to be running before the next election"); early indecisiveness ("We spent 15 months getting approvals from the lawyers, planning committees, and standards people, and there are only 9 months left to do the job"); alarmist fears of closing market windows; or pure optimism about your organization's technical capabilities.

The usual result, as documented in numerous additional failed-RAD projects such as Bank of America's Master Net, the London Stock Exchange, the American Airlines consortium's CONFIRM system, and the Denver Airport baggage system [1,2], is that Dumb RAD projects quickly turn into Disaster RAD or Death-march RAD projects.

The best way to deal with the threat of a DRAD project is to turn it into a SAIV (schedule as independent variable) project. Try to convince the DRAD advocates that they don't want their careers stained by having established a Disaster or Death march RAD project like the ones just cited (software cost and schedule models can help here by providing objective evidence of the risks). But tell them that if a schedule deadline is their critical success criterion, that you can organize to deliver them the most valuable capability, achievable within that deadline. The SAIV approach (and its fixed-cost counterpart CAIV cost as independent variable) works as follows:

1. The clients identify their longest acceptable schedule.
2. The developers estimate the schedule required to develop the required capabilities. It is best to use a mix of expert judgement and software cost and schedule estimation models.
3. If the required development schedule is comfortably within the acceptable schedule, proceed into development. Otherwise:

4. The clients prioritize their desired capabilities.
5. The developers architect the system so that low-priority features can be dropped if the schedule is running out.
6. The project develops a core capability, and proceeds to add highest-priority capabilities until the schedule runs out.

Each of the steps is critical. For example, if time is running out, and you don't have pre-prioritized features and an architecture which enables features to be easily added or dropped, attempts to drop features will generally make the completion take longer.

An Opportunity Tree for Full-Scale RAD

For larger systems, *full-scale RAD* techniques offer several effective methods for reducing cycle time. RAD effects are hard to analyze because the only schedule savings that count come from tasks on the project's critical path. If we view a software project as an activity network of tasks along a development timeline (see Figure 2), we can identify the possible sources of schedule savings by asking the question, "What can we do about the critical-path tasks and the timeline to reduce the project schedule?"

This leads to the RAD Opportunity Tree shown in Figure 3. It is a hierarchical taxonomy of sources of cycle time reduction, which can be used as a framework for assessing various mixed strategies for tailoring a full-scale rapid application development approach to a given organization's environment, culture, technology, and constraints.

Figure 2. Typical Software Development Activity Network

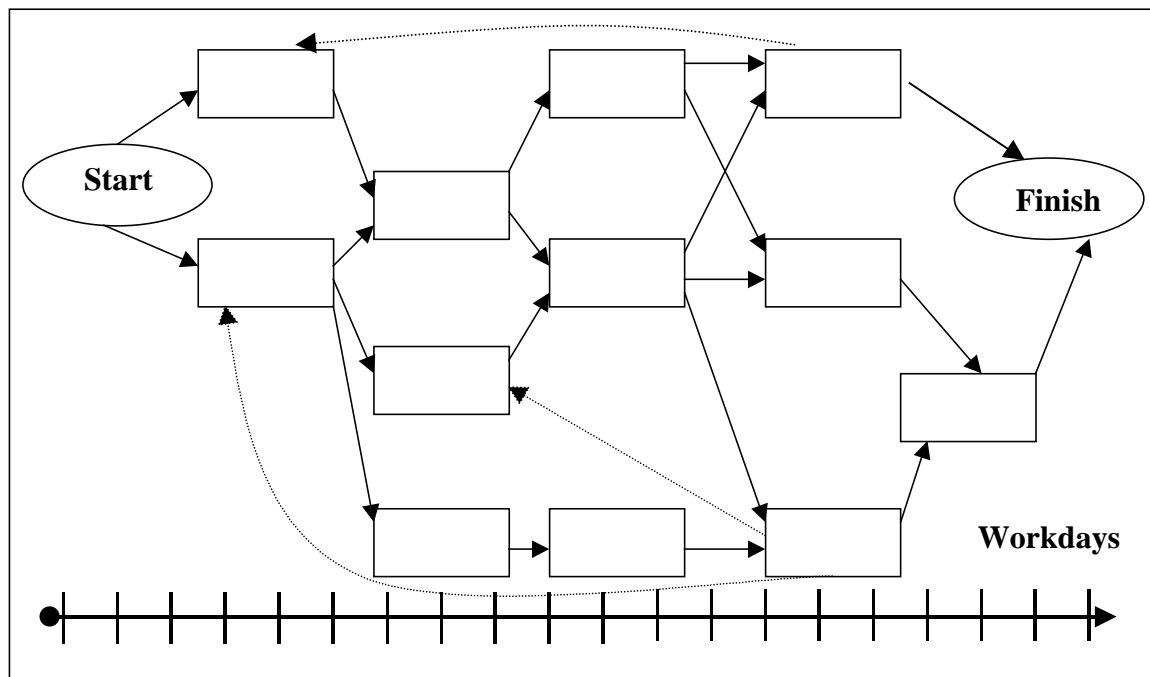
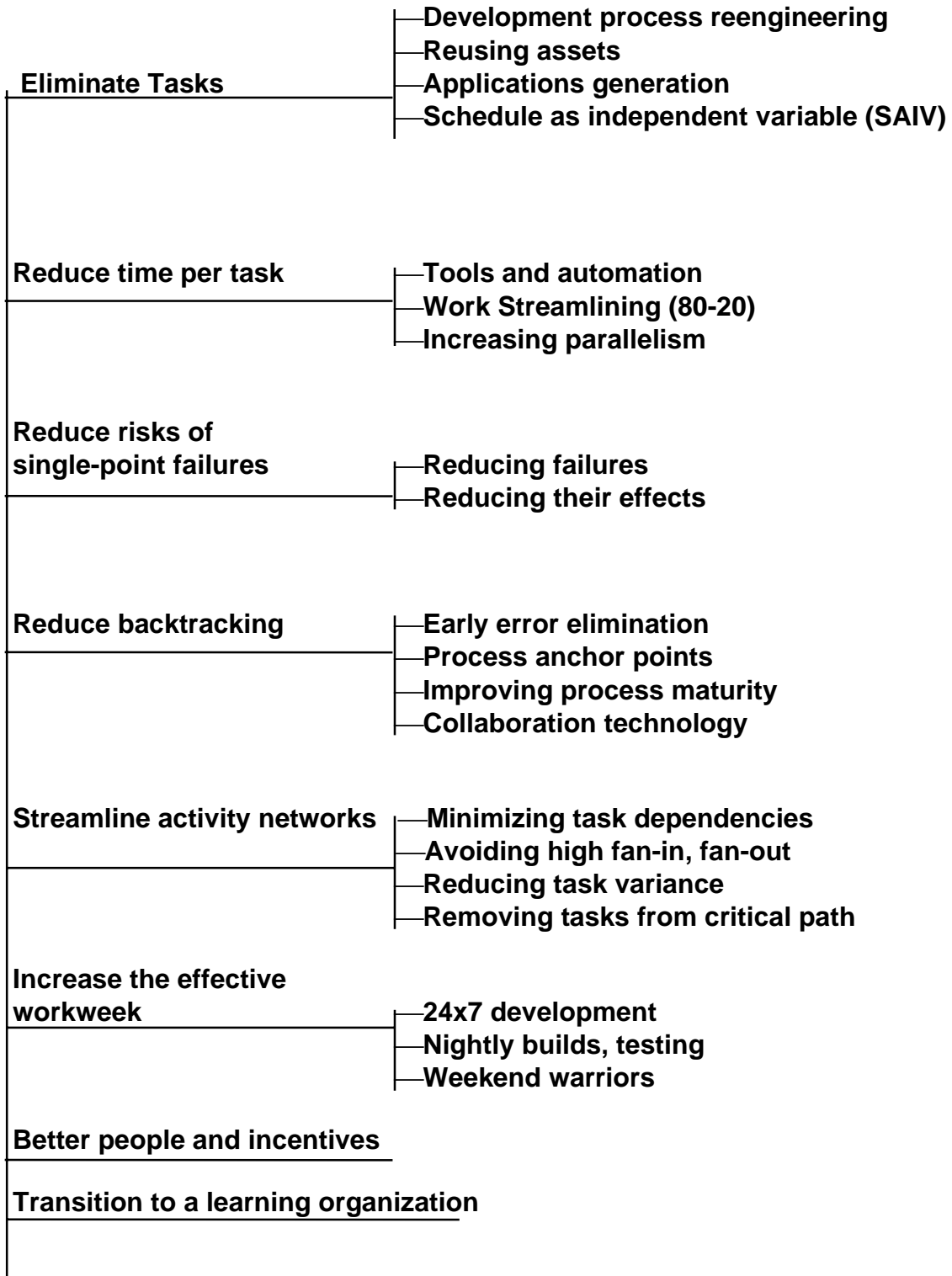


Figure 3. The RAD Opportunity Tree



This model is a bit oversimplified, given that the real world has partial dependencies and more complex constraints, but these do not cause major complications with respect to the use of the model to identify sources of cycle time reduction. Each major source is elaborated below.

1. Eliminate tasks

Development process re-engineering can discover and eliminate tasks that don't add value, such as unnecessary purchase approvals or change control boards operating at too low a level. Asset reuse and automated application generation can eliminate many tasks, but they require up-front investment in domain architecture and product line infrastructure. The *schedule-as-independent-variable* (SAIV) approach simply means scoping the project to the schedule. It can also be highly effective, but again requires an up-front investment in architecture and feature prioritization so that developers can drop features without causing problems in other areas.

2. Reduce time per task

Tools and automation can speed development when existing tasks lack adequate technology support. Collaboration technologies, such as groupware or wide-area workflow management tools, can help distributed teams work together. Pareto 80-20 analysis can be effective for work streamlining. For example, if 20% of the tasks cause 80% of the time delays, then task streamlining should be focused onto that 20%. Developing many modules in parallel can accelerate development. Such parallel development can be enabled by precise, well-validated module interface specifications. These specifications enable many programmers to work in parallel with minimal interface reconciliation and cross-module ripple effects.

3. Avoid single-point task failures

System development projects are sometimes prone to single point failures, which can delay task completion. For example, hardware platforms or components can fail at the most inopportune moment. Similarly, key project personnel can, at a critical juncture, leave the project. You must detect or anticipate where these point failures can occur and take preventive measures. Such measures could include scheduling backup hardware or providing a backup for key team members. Software inspections, besides detecting defects, provide product knowledge backup.

4. Reduce backtracking

Rework is perhaps the most common time sink in system development projects. If rework stems from defects and risks, you can reduce it by developing and employing taxonomies of defects and risks with their corresponding repair actions. Even more effective, however, are techniques for defect and risk avoidance and prevention because they eliminate sources of rework. Prototypes, standards, consistency checkers, and risk assessment techniques are most effective for defect and risk avoidance.

Process anchor points provide a management framework to help determine process goals, objectives (milestones), and progress measures [3]. Process anchor points and baselining help to establish attainable progress markers and overall project development “velocity.” In turn, project velocity should increase as development processes mature, stabilize, and get reused. Such maturity happens most rapidly through top management commitment and resource investment to make it happen. Finally, rework can be reduced by tightening convergence loops or by articulating where progress disconnects occur. For example, when design reviewers review designs outside the presence of the designers, then some record of their discussions and understanding must be prepared, conveyed, and re-explained to the designers. Instead, it is far more efficient to co-locate design reviewers and designers together, or employ collaboration technology to help capture and fill in the gap between the reviewers and designers.

5. Streamline activity networks

Project activity networks can reveal many possible paths to project completion. PERT (Program Evaluation and Review Technique) and CPM (Critical Path Method) tools can help identify critical paths in workflow, resource dependency, or schedule. When projects cross organizational boundaries, their activities should do so off the critical path. When activity networks get too “bushy” - display a high number of input or output paths - bottlenecks can occur. Decompose and spread out these high fan-in, high fan-out nodes and increase parallelism.

Last, as the critical path determines the shortest route to project completion, then look for ways to get time-consuming tasks off the critical path. This may be possible through task decomposition and parallelization, or through network reconfiguration. A good example is to replace the lengthy critical design review with individual design inspections and a short overall review. Some strategies, such as pre-positioning facilities, components, tools, experts, or data, may add somewhat to the cost but be worth it in schedule savings. A good example is “overinvestment” in reusable components. Typical software reuse ROI models conclude that you should expect to use a component at least 3 times to achieve a net payoff, but a lower expected number of uses is appropriate if schedule is more important than cost. Domain engineering can provide a powerful framework to preposition assets for RAD; see [4] for a number of examples.

6. Increase the effective workweek

Although seldom a viable option, making project staff work harder occurs all too frequently. Simply providing better clerical support to your technical people can reduce schedules and improve morale. If you can, outsource non-critical development tasks to others, such as offshore providers or corporate divisions in other countries. Doing so can permit round-the-clock development. To succeed, however, this option usually requires creating a shared product vision, establishing ground rules for collaboration, and ensuring consistent technical decision making. Similarly, you can employ second- or third-shift workers or automation mechanisms like Microsoft’s nightly builds, developer-tester buddy system, or continuous automated testing.

7. Better people and incentives

Bright, quick, versatile, adaptable, creative, experienced, and focused people will be far better at RAD than “average” people. Consequently, the demand for such talent is high. Look for the best, but when you must settle for “best available,” you’ll want your people to perform to their full potential. Motivation is key, but reinforcing it with personal commitment and job or career incentives (e.g., dollar, vacation, or career development rewards for early team project completion) is most effective.

8. Transition to a learning organization

The highest level of software process maturity is the transition to a learning organization. Learning organizations can do more than optimize and manage their processes. They have instead cultivated a culture of process measurement and analysis for continuous improvement and process redesign as routine activities, rather than as uncommon events. The first seven elements of the RAD Opportunity Tree provide a set of RAD improvement hypotheses that a learning organization can employ. Good sources of RAD ideas are the books by McConnell [5] and Arthur [6].

Summary

Use Generator RAD or Composition RAD when you can. Watch out for their scalability and compatibility risks.

Use the RAD Opportunity Tree to craft your best RAD strategy when GRAD and CRAD are too risky.

Invest in continuous process improvement and product line management to enable more RAD opportunities in your future.

The biggest RAD improvements are likely to come from better people and prepositioning assets.

Avoid attempts to push you into Dumb RAD. Use the schedule as independent variable (SAIV) approach as your best counterstrategy.

References

1. R. Glass, Software Runaways, Prentice Hall, 1998.
2. S. Flowers, Software Failure: Management Failure, John Wiley & Sons, 1996.
3. B. Boehm, "Anchoring the Software Process," IEEE Software, July 1996, pp. 73-82.
4. B. Boehm, A. Egyed, J. Kwan, and R. Madachy, "Using the WinWin Spiral Model: A Case Study," IEEE Computer, July 1998, pp. 33-44.
5. S. McConnell, Rapid Development, Microsoft Press, 1996.
6. L. Arthur, Rapid Evolutionary Development, John Wiley and Sons, 1992.