It’s 10 years now since James Martin brought Rapid Application Development (RAD) to prominence in his book, *Rapid Application Development* (Macmillan, 1991). Since then, the arguments for speeding up software development have intensified, yet the RAD paradigm has failed to set the software world alight. What’s gone wrong for RAD?

RAD promised a powerful formula for cutting through traditional software project delays and a giant leap forward in developer productivity. Generally speaking, these hopes have failed to materialize. Few organizations claim to have benefited from RAD techniques.

Traditional software development is based on the specify-design-build-use paradigm. The basic premise of this paradigm is that once a system is specified and designed it can be built and implemented without major remodeling. It has been apparent for many years that this paradigm has major drawbacks when applied to business-oriented software development. Increasingly, systems were delivered that met user requirements at the time they were specified but failed to meet user needs at the time of implementation months or years down the line. RAD aimed at reducing the time between determining requirements and implementing them, reducing the likelihood they would change in the interim period. There is little opposition to the laudable ends of RAD, but the means have caused debate and dissension. RAD is still a misunderstood concept.

Some developers understand RAD to mean incrementally speeding up each stage of the traditional software life cycle. To others, RAD is primarily about automation and the extensive use of software development productivity tools. For some, RAD will always stand for “Rough and Dirty” development—an excuse for sidetracking the disciplines of software engineering standards. Many believe RAD is only suitable for small, relatively low-key projects. Others argue that RAD principles and techniques (Joint Application Development, time-boxing, prototyping, and clean rooms, to name a few) can be applied to any software project.

RAD is accused of being anti-quality. It is commonly believed speed and quality are incompatible in software development. You can have one or the other but not both at the same time. On the other hand, supporters of RAD argue that modern quality principles are inherent in RAD. Fitness for purpose, avoiding waste, getting things right the first time, individual responsibility for quality, and meeting customer requirements is as engraved in RAD as in quality. What RAD tries to avoid is the bureaucratic nature of current quality control and assurance practice.

Other major concerns with RAD include the costs of maintaining clean rooms and of funding the extensive user involvement required for prototyping. Many organizations have found the culture changes required for RAD impossible to achieve either within the business as a whole or within project teams. Soft-
ware engineers, too, are reluctant to embrace a development ethos that necessitates leaving their comfort zones and subjecting themselves to the intense periods of activity and accountability required for RAD. A recent article reporting on a RAD project in England was entitled, “It’s a Lot of Bits of Paper and Ticks and Post-It Notes and Things: A Case Study of a RAD Project” (P. Beynon-Davies, et al. Information Systems Journal, July 2000). This is hardly what Martin had in mind, and demonstrates that RAD is floundering to establish credibility in the software development mainstream. How can direction and impetus be restored to RAD?

The first essential is to clearly distinguish between two forms of RAD: Rapid Program Development (RPD) and Rapid System Development (RSD).

RPD starts with a program specification. The aim of the project is to implement the specification in the shortest possible time. Experienced programmers are required who can quickly understand a specification, produce high-quality code under strict time constraints, and deliver results. There need not be much, if any, direct communication with or involvement of users so that well-developed interpersonal skills are not essential for RPD staff.

RSD is a different prospect altogether. The project starts with the kernel of an idea for a new or updated computer-based system. Extensive user involvement is required to determine current ways of working, reactions to proposals, and other system-specific implications for change. Prototypes of proposals need to be developed for analysis and scoping of requirements. RSD thus requires developers with both technical and interpersonal skills. There is no up-front specification; this must be developed as the project progresses. The whole process is much more chaotic than RPD, and the aim of applying RAD techniques is to bring order to the uncertainty. The success of the project is determined by the ability of a team to turn an initial system concept into a working system that adds value to a business operation in a short period of time. Value is the operative word in RSD, and this leads to the recognition that RAD is essentially value engineering.

There is nothing so wasteful as doing with great efficiency that which doesn’t need to be done. Formalized methods often lead to inertia in the development process by over-intellectualizing it and instigating bureaucratic controls. In the pursuit of what they perceive as a perfect solution, software engineers can easily become detached from the real world of users and from the costs of their work. Inordinate amounts of time and effort can be expended on analyzing and implementing functions not crucial to the development but that capture the interest of developers. Value engineering aims to eliminate all work in a project that does not add value to a product.

Value engineering is an iterative process, the aim being to arrive at the simplest and easiest-to-implement set of requirements and system designs. Most software engineers do not think in this way. Lack of domain knowledge often means that user requirements, once stated, are taken as fixed. Proposed designs often have more to do with developer preferences and capabilities than with simplicity and ease of implementation.

Most causes of unnecessary work in projects relate back to two things: Initial requirements and design decisions. Inadequate and unnecessary requirement specifications lead to all kinds of misunderstandings, wasted effort, and rework. Overly complex designs cause needless construction, testing, and implementation costs. Value engineering means keeping things simple. A by-product of this philosophy is that simple systems are easy to maintain so the benefits of value engineering recur throughout a system’s lifetime.

Essentially the message of value engineering in RAD is to start from implementation and work backward. If a set of requirements leads to a complex-looking implementation, then the requirements need to be revisited to see how they can be re-scoped to
simplify implementation. If it is not possible to scale down the requirements then the next thing is to revisit the proposed system design to see how this can be simplified.

Getting enough developers to offer value engineering to clients is not easy. Simplicity is not a trait that comes easily to software engineers raised on complexity. There needs to be something more than exhortation to get most developers to “think simple.” In order to make value engineering work it is essential to instill strict financial discipline in projects. The best way of ensuring nonvalue-adding work is not done is not to fund it. In traditional system development, project costs are estimated using a bottom-up approach. The costs of the individual activities required to produce products are summed to estimate the total project cost. For value engineering it is essential to use top-down cost evaluation. Project sponsors are asked what they are prepared to pay for the system and when it is required. The end products are then scoped to fit in with what can be achieved within the time and budget constraints. In this way developers are forced to concentrate on value-for-money outcomes. Non-value-adding work is transparent and can be ditched.

In rapidly changing environments, developers must deliver systems quickly to meet business, and not their own, timetables. Spending months and years developing systems to high standards is fruitless if over time requirements change beyond recognition. Software development must serve its customers. Simple value-for-money systems that work are better than expensive and complex ones delivered late, over-budget, and that are difficult to maintain. If software engineers can learn to embrace value engineering principles, then RAD can mature and lose its promissive image.

Failing that we may look back on RAD in the same way as prohibition: a good idea but it didn’t work.

Alan Howard (A.Howard@lmu.ac.uk) is a senior lecturer in information systems development at Leeds Metropolitan University, England.