

One Size Does Not Fit All

Software Development Challenges and Strategies for Better-Managed Projects



**One Size Does Not Fit All: Software Development Challenges and Strategies
for Better-Managed Projects**

EXECUTIVE SUMMARY

For today's private businesses, mismanaging a software development project can mean being consumed by competitors; for a public entity, it can mean ineffectively delivering services to its citizens.



TABLE OF CONTENTS

Abstract	4
The Software Crisis	4
The Nature of Software Engineering	4
Best Practices for Improving the Software Development Process	6
Don't Go It Alone: Choosing an IT Services Provider	9
Conclusion	10
ABOUT KEANE	11

TABLE OF FIGURES

Figure 1 Developing software is all about people, process, and tools	6
---	---



Abstract

The software crisis — a term coined almost four decades ago — still plagues IT organizations today. Manifested as over-spent budgets, missed deadlines, and unmet requirements, the software crisis costs the IT industry billions of dollars each year. For today’s private businesses, mismanaging a software development project can mean being consumed by competitors; for a public entity, it can mean ineffectively delivering services to its citizens.

Although throngs of texts, papers, and other narratives have been written during the past 40 years to address the subject (and challenges) of software construction and its related activities, there is still no comprehensive solution to the problem. Why not? Software engineering is a young, complex, and ever-changing discipline.

For an individual IT organization with limited resources, solving the long-standing problem of software engineering is improbable. However, whether you are an architect, project manager, or CIO, this paper presents a number of strategies and best practices that will help you set realistic goals and mitigate the risks inherent to software development projects.

The Software Crisis

By the mid-1960s (less than three decades after the genesis of programming), the process of software development reached a discouraging state. Budgets were over-spent, deadlines were missed, requirements were not met in their entirety or at all, projects were cancelled, and documentation was inadequate or nonexistent (leading to almost unmaintainable and unenhanceable code). Dr. Winston Royce, software engineering and development guru, stated that software construction is an “unexpectedly hard” problem, possibly the hardest problem in engineering, and the longest running dilemma in the engineering world.¹

In 1968, the NATO Science Committee

assembled dozens of information technology industry experts, theorists, and technology leaders to address these problems, which later became known as the “software crisis.” While they did manage to define software engineering as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, the summit was not able to articulate the details of an approach that would actually resolve the software crisis.

The software industry’s response to the crisis has been compared to the late-19th-century pharmaceutical industry’s unlikely claims of cure-all concoctions, which were peddled by fly-by-night experts. Today software vendors make similar assertions: “Just use our tools and processes and all your problems will be solved!” While currently some of the fundamental characteristics and nuances of software development may have changed, the software crisis still exists and is costing the IT industry billions of dollars. The average software development project misses completion dates by 50%, and 75% either do not function as intended or are not used at all.²

The Nature of Software Engineering

Why does software engineering pose such a problem? The nature of software engineering is complex. Even the term itself is subject to debate. Here are a few of the reasons that make software development management so challenging:

Ambiguous Semantics: The term “software engineering” is ambiguous and its definition is heavily debated. There is significant question within the IT industry as to whether the term deserves the moniker “engineering” in the first place. Founded in 1934, The National Society of Professional Engineers (NSPE) is an advocacy group promoting the licensing and education, and protecting the rights, of professional engineers throughout the United States. In the 1990s, the NSPE filed a lawsuit

to prohibit entities from using the term “software engineer” as a job title and won the action late in the decade. The argument was based on the notion that software engineering did not meet the minimum requirements and exhibit the same characteristics as traditional engineering disciplines (for example, professional licensure). The software community has since ignored the verdict, and even the United States government uses the term as a job classification.

Engineering can be defined as the application of science (physical manifestations that appear in nature) to solve a problem.³ Yet, some maintain that there is very little science involved with software engineering. If anything, it is based on the implementation of mathematical theories or algorithms that may have no manifestation in nature at all (if you destroy the computer, the software no longer exists). According to this definition, anything that is not realizable in a practical sense is not engineering.

Some schools of thought treat software engineering in a more abstract sense. That is, although traditional engineering (sequential-type models) and concurrent engineering (spiral- or iterative-type models) have proven, successful results in other categories of engineering (such as construction), applying these models to the process of developing software has resulted in a much higher rate of failure. Therefore, some believe that software engineering is based on other engineering disciplines only in spirit and that the disciplines themselves are fundamentally different. It is interesting that the ambiguous and controversial view of software engineering’s definition parallels the often ill-defined and historically unsuccessful processes of writing the software applications themselves. One cannot help but wonder that if a definition can be agreed upon and widely accepted by the IT community, it might be easier to define a more predictable and repeatable process for developing software.



Immature Discipline: Software engineering is still a very young discipline. The field itself (however it is defined) is only 50 or so years old. By comparison, the field of chemistry is over 200 years old. (Many scholars recognize Antoine-Laurent de Lavoisier as the father of modern chemistry whose work took place in the late 18th century.) If we consider the amount of progress made in the field of chemistry in the past 200 years, we can imagine the amount of work software engineers have left to do.

Too Many Moving Parts: While the activities involved with writing software may be based on hundreds or thousands of well-understood moving parts, the ability to assemble those parts into a repeatable and predictable manner is difficult. Collaborative learning evangelist and author, Dr. Brad J. Cox compares the problem with the craft of musket making before Eli Whitney. In developing the concept of mass production for the purposes of manufacturing muskets, Eli Whitney built machines that produced each of the musket's constituent parts. This allowed less-skilled laborers to assemble the weapons at a higher rate and at

"If we are ever going to lick this software crisis, we're going to have to stop this hand-to-mouth, every-programmer-buildseverything-from-the-ground-up, pre-industrial approach."

—Brad J. Cox, collaborative learning evangelist

a cheaper cost. This was a much more desirable alternative to the skilled craftsmen that were required to hand-build rifles prior to this.⁴

While the concept of interchangeable parts can be applied to other areas of manufacturing, it is difficult to apply them to software development. In some cases, the atomic parts we once

thought were immutable are now moving targets as well. For example, an algorithm that performed well on one hardware platform may not operate as well on another platform as hardware technologies evolve. This is not to minimize the difficulty of building bridges or high-rise buildings, but software engineering presents its own unique set of challenges. While circuits and chips are based on the fundamental and well-understood principles of electricity, built on top of this is an operating system. This immediately injects the problems of software engineering. Even if we assume that the operating system is well understood, predictable, and stable (which, in many cases, it is not), custom software applications built on top of operating systems amplify a pre-existing problem. Software developers are not only trying to address the business requirements, they also have to wrestle with the problems and idiosyncrasies of the technical environment. As Cox states, "Before the industrial revolution, there was a nonspecialized approach to manufacturing goods that involved very little interchangeability and a maximum of craftsmanship. If we are ever going to lick this software crisis, we're going to have to stop this hand-to-mouth, every-programmer-buildseverything-from-the-ground-up, pre-industrial approach."

Measurement of Effectiveness and Quality: Because of the imperceptible nature of software, not all software development project successes and failures are measured adequately. What is the baseline for "failed" projects: cost, time, customer satisfaction, throughput, or response time? There is consensus that projects fail, but what are the semantics and measurement criteria of that failure? Applying the Six Sigma quality management method will address many of these issues, but in most cases this heavyweight approach is inappropriate for small- and medium-sized software development projects.

Like software engineering, software

quality is not always consistently defined. Many base quality solely on the number of defects identified, but there is much more to the definition. Consider portability, efficiency, usability, testability, understandability, and extensibility. Software quality is difficult to measure. Other engineering disciplines measure quality in standard ways, such as the temperature a piece of steel can withstand or the speed at which a vehicle can travel. While some software characteristics such as response time can be measured in a relatively straightforward fashion, other characteristics cannot. For example, how does one determine how modifiable a component really is? How does one define a "small" or an "acceptable" defect? Research in this area generally concludes that probability-based methods for measuring reliability are ineffective, but statistical methods tend to be more robust (such as measuring the average total number of failures, the number of failures per time increment, or the average time interval between two failures). Unfortunately, this still does not help us with many of the more intangible properties of software (for example, an efficient user interface).

Uniqueness: Despite advances in reusability, no two systems are ever exactly alike. Although different software programs may solve similar problems, their designs may vary greatly. In his book, *Facts and Fallacies of Software Engineering*, Robert Glass discusses how a diverse set of problems requires a diverse set of solutions, too diverse at this time to make large-scale component reuse viable.⁵ It is the exception to the rule to find a large component that is reusable across applications, let alone domains (between finance and manufacturing, for example). Sure, it might be easy to build a reusable component that parses strings and share it across many applications. It is quite a different matter to build a generic component that computes local tax rates that can be reused across the



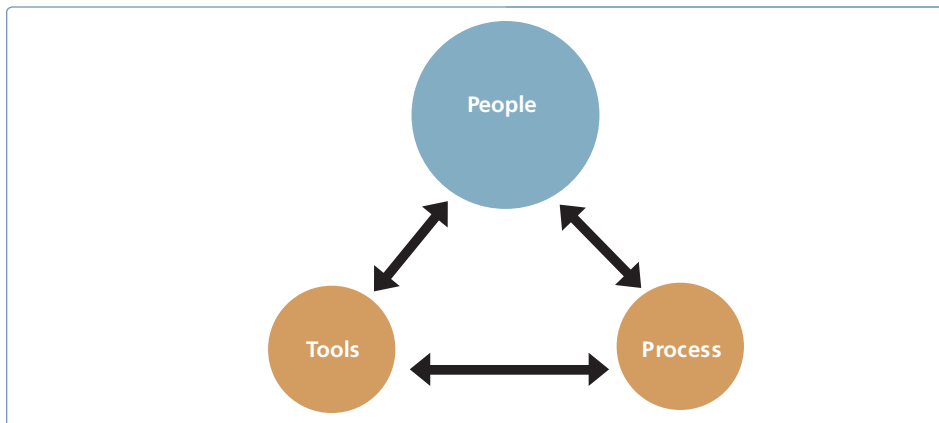


Figure 1 Developing software is all about people, process, and tools; people being the most important component.

country or the world.

Complexity: As the size of a problem increases, the complexity of its solution increases exponentially (in some estimations as much as four to one).⁶ This issue reverberates through design, construction, test, and maintenance activities, causing the curve to rise even faster.

Error Replication: In many cases, software errors occur only under a very specific set of circumstances and may be difficult to replicate. What might be thought of as a set of unrelated behaviors may ultimately be coupled (or vice versa), making debugging almost impossible. Multiple defects may cancel each other out, increasing the amount of time it takes to uncover them.

Best Practices for Improving the Software Development Process

Unlike the silver bullet's ability to slay the werewolf, there is no one solution to solving all the problems that have plagued software development for decades. This is according to Frederick Brooks, known for his landmark contributions to computer architecture, operating systems, and software engineering. Brooks asserts that past developments in software engineering address only the byproducts (necessary evils) of the software engineering process, such as disparate technologies and programming environments or the slow speed inherent

in certain design paradigms.⁷ What really needs to happen is for software developers and managers to address the essence of software engineering. In other words, they need to acknowledge that developing software is complex and software is inherently complicated. Approaches that address the nature of software engineering include the following:

Utilize Incremental (or Iterative)

Techniques: Some schools of thought think of software as being grown (evolved), rather than built.⁸ It is rare that requirements are well understood enough to build a system accurately in one pass because the variables are almost always too complex.⁹ Clear and stable requirements are an unattainable goal. Furthermore, the stakeholder may not even have control over his or her own environment. Therefore, review requirements early and often. One of the best ways to learn something is to try it and integrate lessons learned from prior iterations into subsequent iterations. There are a number of schools of thought on where to start early iterations — with the largest components, the riskiest components, or the least understood components. In any case, assume the first iteration will be a “burned pancake” to be thrown out. Use it only to adjust the temperature of the pan and move on. Iterative development is a core concept in agile development

techniques, but is not necessarily relegated to those frameworks. (For more information on agile development techniques, see www.agilealliance.com.)

Procure Great Designers and Developers:

This is a concept that deserves special attention. Subscribe to the notion that developing software is all about people, process, and tools. While finding the right talent is only a third of this equation; it is the most important component. Glass states that “the most important factor in software work is not the tools and techniques used by the programmers, but rather the quality of the programmers themselves.”

Author and computer science professor of practice Gary Pollice, who has been investigating how “soft” issues affect software development, such as the way people collaborate, echoes this sentiment. Yet, he goes further by saying that people make the difference between the success and failure of a project.¹⁰ Scores of additional references can be cited (see the list of endnotes at the conclusion of this paper) that exemplify the importance of people over process and tools; yet, the IT industry continues to look the other way and assume that people can be swapped in and out of a project without recourse. Rather than finding qualified personnel, the industry often falsely relies on tools and techniques to fill in the gaps made by unskilled developers all in an effort to save time and money.

The importance of people in software development is not a new concept, as computer scientist Raymond Rubey addressed in 1978, “When all is said and done, the ultimate factor in software productivity is the capability of the individual software practitioner.”¹¹ In the context of more contemporary frameworks, author and agile development practitioner James Highsmith explains that when the façade of formal and rigid process is removed, the reason projects are successful is because of the people.¹²



The Software Engineering Institute (SEI) Capability Maturity Model (CMM) focuses on the identifiable and repeatable processes to enable quality. Significant improvements in software quality have come about as a result. In 1995, the SEI developed the People Capability Maturity Model (P-CMM) to address the personnel issues in organizations. While CMM is a widely embraced framework, P-CMM unfortunately remains a largely unexplored realm. Yet, performance levels are 5 to 28 times higher in good programmers than in poor ones.¹³ Not all developers are created equal, and don't assume that process and tools can build the better programmer. It isn't always necessarily the case that the best resource is the one with the most acronyms on their résumé. Intelligence, problemsolving skills, and craftsmanship are equally important as experience.

Reuse Components:The software industry is attempting to define the creation of software as a construction of pre-existing components, rather than the design of workflows, algorithms, and modules from scratch. Likewise, many industry leaders and academics believe that software should be assembled and not written.¹⁴ There are very few atomic pieces of functionality that have not already been authored, and developing a new system is just a matter of assembling components that have been produced by other vendors or that are products of previous implementations. While beyond the scope of this article, component-based development (CBD) and, more recently, serviceoriented architecture attempt to address these issues (among others).

While these techniques are certainly a step in the right direction to achieving these goals, component reuse has its own set of risks that need to be realized and managed. Challenges with componentbased development include:¹⁵

- Who will own the component?
- Who will maintain the component?
- Who will guarantee the component's

reliability and security?

- What happens if the component becomes obsolete?
- How do you design a component without knowing how it will be used ahead of time?
- What happens if not all components are available at the time of assembly?
- What happens if the component does not exactly meet user requirements?
- Who incurs the cost of developing the component?

Additionally, in a consulting environment where in many cases IT services providers are being paid to address the needs of an individual business unit, it is important that stakeholders realize the importance of moving away from the "stove pipe" mentality and adopt architectures that are for the greater good of the enterprise as a whole. Individual business units may have a myopic view of the organization and can tend to consider only their own needs. In scenarios where the project sponsors include the IT organization, addressing this issue at an organizational level may not be as much of a problem. It is not until these governance questions are answered, that CBD will provide a viable "cookbook" method for constructing software to the extent that many hope it will be.

On a more pragmatic note, developing components for reuse is harder than developing single-use components. Although this may seem like another paradox, reusable components increase in complexity to accommodate their generic nature. By some estimates, reusable components should be tested in a minimum of three environments before they are considered viable. This incurs additional time and money, and probably isn't within the scope of the current project.

Use Tools Effectively:Tools are enablers of reuse and reengineering, but remember this one caveat: a poorly understood, improperly implemented, or forced usage of a tool can sometimes

cause as much, if not more damage than not using tools at all. Limitations and dangers of tools include:¹⁶

- Cost of acquiring, learning, and applying the tool
- Limitations and/or deficiencies of the tool
- False expectations of what the tool can actually do
- Powerful tools applied inappropriately
- Mismanaging a tool
- Over dependency on the tool

Bringing tools into the enterprise requires a culture change and buy-in from all effected parts of the organization. Without the proper communication, selection processes, and training, tool implementations will result in failure. Remember that the tools are only as good as the people and process wrapped around them. When these concepts are understood, effective use of tools can have a significant impact on the success of a project. To use tools effectively, be sure to:¹⁷

- Identify tool dependencies
- Use simple tools and simple configurations of complex tools — don't over-engineer use of the tool
- Avoid inadequate tools — don't use a tool for the sake of using a tool
- Develop and share tool-specific knowledge
- Communicate with management about the requirements for tools
- Keep tools upgraded
- Avoid the tool pit — Use combinations of tools as necessary, instead of looking for the "silverbullet" tool. Make sure the tools are there only to support the people and processes, and use only the features that improve efficiency.¹⁸
- Remember that paper and pencil might be the best tools after all

Establish a Framework Implementation Plan: Derived from a concept originated



“When all is said and done, the ultimate factor in software productivity is the capability of the individual software practitioner.”

— Raymond Rubey, computer scientist

by IBM and Rational Software, the Framework Implementation Plan identifies what software development lifecycle will be used on the project (linear, such as waterfall; spiral; iterative, such as Rational Unified Process; agile, such as XP; and others). It provides an explanation of the process or work approach for the project and explains why the process was chosen. Furthermore, it describes the Software Development Life Cycle (SDLC)-specific phases and activities to be used, the artifacts to be created, and a role map of responsibilities. Most importantly, it explains how the approach will be customized for a particular project. Just as there is no silver bullet, not all frameworks are created equal. No two projects are the same; so do not attempt to apply frameworks as if they are. While a treatise on framework selection is beyond the scope of this document, suffice it to say that a framework must be selected, modified if necessary, and documented. As software professional Hank Rainwater states: “Create and tailor methods to fit your enterprise and ensure that they all have one goal in common: Deliver quality software promptly.”¹⁹

In a consulting environment, the foundation for the plan may be located in the IT provider’s proposal and further defined in the statement of work (SOW). Continue updating this artifact through the inception phase activities and add additional detail, have management agree on it, and communicate it to the entire team as well as the relevant stakeholders.

Identify a Process: While we suggest in the paper that in many cases process is overemphasized, it remains a necessary factor. And while not all projects need to accept predefined processes as written (see previous section on “Establishing a Framework Implementation Plan”), the point is that some process should be identified. Do not confuse the concept of using a rigorous process with formality. Rigor is a strict and systematic approach that can be used to control costs and schedules, measure progress, and increase confidence in quality.²⁰ Formality requires that work products be verified by mathematical principles. Formality is not necessarily a requirement. Rigor is. Although there is certainly nothing wrong with applying past experiences and rules of thumb when necessary, beware of chaos rearing its ugly head, masking itself in the form of customizing the process. Customization is not an excuse for being sloppy.

Achieve Consensus: When developing a framework implementation plan, stakeholders need to agree on what artifacts will be generated. One tip for this would be to generate only those artifacts that provide value. Ask yourself, “How will this artifact be used in the future?” If the answer is ambiguous or nonexistent, then consider not producing the artifact. As Pollice mentions, “Only do those activities and produce those artifacts that directly lead to delivering value to your customers and stakeholders.” Once the artifacts are identified, one can determine how to back into them with activities and tasks. Additionally, if the team does not expect the artifact to be updated after its initial release, serious consideration needs to be made as to whether or not the artifact should be generated at all. Finally, remember that it is one thing to identify a process, but you need to be sure the process is followed.

Recognize That Process Does Not Come Free: If you are using a heavyweight methodology such as the Rational

Unified Process, remember that there is a cost associated with implementation. The cost comes not only from the tools, but for inexperienced implementors, a significant cost comes from training

“Create and tailor methods to fit your enterprise and ensure that they all have one goal in common: Deliver quality software promptly.”

—Hank Rainwater, software professional

“Only do those activities and produce those artifacts that directly lead to delivering value to your customers and stakeholders.”

—Gary Pollice, computer science professor of practice

and ramp-up activities. In a consulting environment, these costs are not only incurred by the IT services providers, but also by the customer.

The reason for this is that the lifecycle process does not occur in a vacuum. Software development processes affect all manner of roles, including stakeholders, subject matter experts, users, and testers. Adequate knowledge and buy-in need to exist with all parties to guarantee a successful implementation. While at first acknowledging this may be a daunting task (in terms of cost and time), the investment in ramp-up activities will result in long-term cost savings not only for the project, but also for the organization.

Remember that new tools and processes also add to costs by lowering a programmer’s productivity in the short term. Benefits can be realized only when the learning curve is conquered, and if results are assessed realistically and quantified.²¹ Be sure to avoid zealots who attempt to strong arm the team into implementing unsubstantiated



techniques.

Beware of Brooks' Law: According to Brooks' Law, while programming work performed increases linearly with the number of programmers, the complexity of a project increases exponentially with the number of programmers. Therefore, hoards of programmers working on a project will become entangled in a web of confusion.²² Brooks himself states: "Adding manpower to a late software project makes it later."

Never forget these tenets. Although most managers measure staffing in person-months (or similar), mistakes are made in assuming that all personmonths are the same, and that people demonstrate equal productivity. In extreme cases where resources are added to a late project, the productivity factor may very well be negative. Certainly there are situations where exceptions to this rule can be made, such as when new resources have extensive domain expertise. But even in such circumstances, careful planning and consideration should be made to minimize exacerbating a late project.

The Only Thing Not Susceptible to Change Is the Fact That Things Change:

For all but the most trivial of implementations, gone are the days when we can assume that only when requirements gathering activities are complete, accurate, and immutable can we start design activities. While the waterfall lifecycle may be appropriate in cases where the business processes are known intimately or when the project is small (and there is little room for misinterpretation), in most other cases it will probably serve as an inadequate framework approach. The biggest problem with the waterfall approach is that it does not present the users with anything on which they can provide feedback until very late in the process, thus uncovering risk late. Changes made late in the schedule are exponentially more expensive to address than those uncovered earlier. It is probably a safer bet to choose a more

flexible process that accommodates for a changing environment and presents stakeholders with artifacts and work products for review earlier, rather than later. Conversely, do not assume that if you expect requirements to change to a minimal or moderate degree, that you will necessarily need to use an agile development framework. Finally, while understanding that requirements will change, design must accommodate making future changes easy to apply. This is yet another fundamental difference between software engineering and other engineering disciplines.

Make a Connection: Most programmers are detailoriented, and fortunately for them writing software is a detail-oriented business. When requirements are incomplete or misunderstood, more often than not this is due to a lack of detail and is a recipe for scope creep. This is called the "Curse of the Casual Requirement."²³ An unfortunate side effect of this occurs when a project is running late; in many cases the developers endure the brunt of the catchup activities ("We need to code more, code faster. There's missing functionality; programmers and testers have to work nights and weekends to get it done!"). This is because the developers and testers are the last ones to touch the product and are sometimes unfairly required to make up for mistakes that were made much earlier in the process (during requirements gathering activities). Failure of most projects is not attributed to technology problems; it is due to poor requirements-gathering activities and it is not until user acceptance testing that these problems are really discovered.

Have a plan in place to transition artifacts from business analysts to designers and developers. Make sure that each role's responsibilities are well defined and that expected actions of all parties are known when this transition is made. Are the business analysts or the technical analysts responsible for defining the business rules? All too often,

business analysts assume their work is done, and then artifacts are handed over to technical analysts and developers only to find that not enough information is available to create a design. While it is perfectly acceptable for designers to pick up these activities at earlier points in the lifecycle, all parties must be aware that this is the expectation. There have been cases where teams as small as three (two of which were working in the same cube) have encountered this miscommunication. Placing a developer on the requirements team may alleviate some of these issues.

Foster Communication: Never underestimate the organizational impact and business changes that implementing a piece of software can have. Do not forget fundamental project management tenets. It is critical that communications take place in the organization before, during, and after the project is implemented. All too often it is the case where users find out only during an implementation that changes are on the horizon. In many cases this results in reluctant and resentful users.

Don't Go It Alone: Choosing an IT Services Provider

While there is no single approach to guaranteeing a smoothly functioning application development project, many organizations choose to employ a partner with considerable experience with a variety of business and technical environments and development frameworks. To help ensure success, performing a detailed assessment of the potential partner's qualifications is essential.

When selecting an application development partner, consider the following:

Experience: Has the IT services provider successfully implemented development projects similar to the scope and complexity you need? Does the company have an in-depth understanding of what is required for a successful initiative?



Does the partner company have the necessary skills to complete the project? Once a partner is selected, it is vital that both the partner's role and the scope of the project are clearly defined.

Also important is meeting potential team members. Be sure to ask them about their experience overcoming obstacles with past software development projects.

Proven Methodology: Does the partner employ proven methodologies when setting up the project to ensure that the appropriate level of commitment and resources are designated for each component of the project? Also important, can the IT services provider demonstrate the flexibility to alter methodologies as necessary?

Keane has expertise in a wide array of proven methodologies, including e-Development, a licensed extension of the Rational Unified Process.

Best-Practices Approach: Does the firm have hands-on experience with software development and does it have validated, best-practices approaches?

Good Business and Cultural Fit: Does the company have the necessary background in your industry? Does the potential partner understand your business processes? Does it understand, and fit in with, your business culture?

Predictability: Does the firm use common technology migration processes, management disciplines, methodologies, and metrics across all teams to ensure consistent, predictable, and high-quality performance?

Conclusion

Never take claims of a one-size-fits-all approach to the software development process at face value. Certainly problems will occur and eliminating them in their entirety is unlikely to happen anytime soon. Instead, set realistic goals. Take the appropriate steps to mitigate risks by not overdoing the process, being flexible, and surrounding yourself with talented

professionals.

Christopher Desany is a senior principal consultant and technical architect with Keane, Inc. He can be reached at Christopher_G_Desany@Keane.com.

End Notes

1. Royce, Winston, "Current Problems," Aerospace Software Engineering: A Collection of Concepts, ed. Christine Anderson and Merlin Dorfman, American Institute of Aeronautics, Inc., 1991.
2. Gibbs, W. Wayt, "Software's Chronic Crisis," Scientific American, September 1994.
3. Liu, Lay, "Is Software Engineering Actually Engineering?" The Iron Warrior, October 2003.
4. Cox, Brad J., "Planning the Software Industrial Revolution," IEEE Software Magazine — Software Technologies of the 1990s, The Institute of Electrical and Electronics Engineers, 1990.
5. Glass, Robert, Facts and Fallacies of Software Engineering, Addison-Wesley, 2003.
6. Woodfield, Scott N., "An Experiment on Unit Increase in Problem Complexity," IEEE Transactions on Software Engineering, March 1979.
7. Brooks Jr., Frederick P., "No Silver Bullet — Essence and Accidents of Software Engineering," Computer, 20, no. 4, The Institute of Electrical and Electronics Engineers, Inc., April 1987, 10–19.
8. Mills, H.D., "Top-Down Programming in Large Systems," Debugging Techniques in Large Systems, ed. R. Ruskin, Prentice-Hall, 1971.
9. Hohmann, Luke, Journey of the Software Professional — A Sociology of Software Development, Prentice-Hall, 1997.
10. Pollice, Gary, Liz Augustine, Chris Lowe, and Jas Madhur, Software

Development for Small Teams A RUP-Centric Approach, Addison-Wesley, 2004.

11. Rubey, Raymond L., "High Order Languages for Avionics Software — A Survey, Summary, and Critique," Proceedings of NAECON, 1978.
12. Highsmith, James A., Agile Software Development Ecosystems, Addison-Wesley, 2002.
13. Glass, Robert.
14. Pressman, Roger S., "Software Engineering," Software Engineering, The Institute of Electrical and Electronics Engineers, 2000, 57–74.
15. Vitharana, Padmal, "Risks and Challenges of Component-Based Software Development," Communications of the ACM, 46, no. 8, August 2003, 67–72.
16. Hohmann, Luke.
17. Hohmann, Luke.
18. Pollice, Gary.
19. Rainwater, J. Hank, Herding Cats: A Primer for Programmers Who Lead Programmers, Apress, 2002.
20. Ghezzi, Carlo, Mehdi Jazayeri, and Dino Mandrioli, Fundamentals of Software Engineering, Prentice-Hall, 1991.
21. Glass, Robert.
22. "Brooks' Law," http://en.wikipedia.org/wiki/Brooks%27_law
23. Duncan, Christopher, The Career Programmer: Guerilla Tactics for an Imperfect World, Apress, 2002.

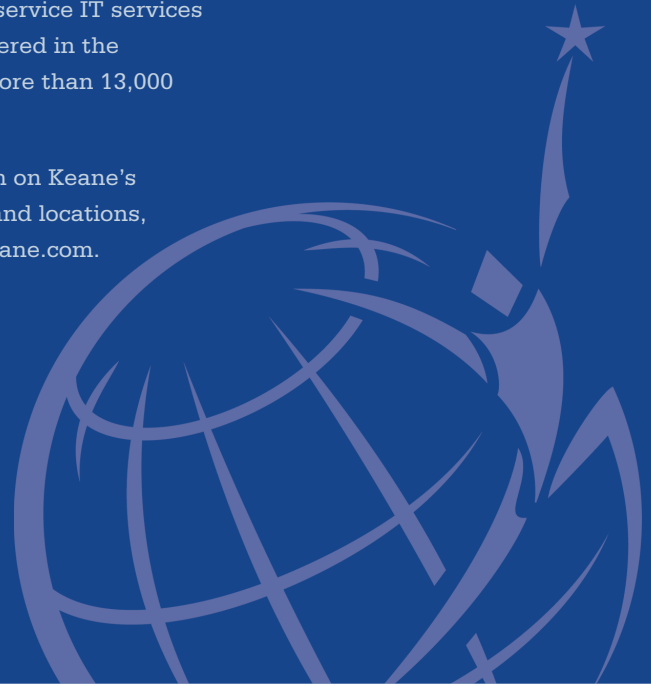


About Keane

Keane partners with businesses and government agencies to *optimize* IT investments by delivering exceptional operation, maintenance, and evolution of mission-critical systems and business processes. Keane helps clients realize the greatest value from their IT investments by leveraging an insider's hands-on understanding of the nuances and subtleties of their applications, processes and infrastructure making the recommendations we give more actionable, the work we do more pragmatic, and the results realized more measurable.

In business since 1965, Keane is an agile, mid-sized, full service IT services firm with headquarters in the United States and more than 13,000 employees globally.

For more information on Keane's services, solutions, and locations, please visit www.keane.com.



Keane

Corporate Headquarters
88 Kearny Street, Suite 1650
San Francisco, CA 94108

For more information about Keane's services, contact us at:

877.88.KEANE
info@keane.com
keane.com

