Software development productivity and project success rates: Are we attacking the right problem?

Joe Marasco
February 15, 2006

from The Rational Edge: Practitioners of iterative development have made considerable improvements regarding requirements management and traceability techniques over the past decade. But how do we know we’re managing the right requirements in software development projects? Joe Marasco takes up this question and proposes a solution to an interesting problem.

I have recently been confronted by an interesting contrast regarding progress in the arena of software development:

- We are producing software systems that are much more complex and sophisticated than those of yesteryear. Our progress, both in terms of functionality and productivity, is impressive.

On the other hand:

- Our progress in terms of project success is not impressive. Imagine that the software industry has a meter that monitors project successes and failures -- a dial gauge with a needle pointing along a numerical scale. Over the past several decades, the needle records a slowly advancing success rate; it’s moving in a positive direction, but the pace is glacial.

Of course, we can always debate what we mean by “success” and “failure.” There are two general definitions of success. The first is that of the business executive: Does the delivered system meet or exceed the expectations we had for it? The second is that of the project manager: Did the system come in “up to spec,” on time, and on budget? The disparity between these two definitions represents a communications gap in itself. But regardless of which definition we use, one thing is clear: We are not getting better fast enough, despite spending billions of dollars a year on software development technology and tools.
In this paper, I propose a broader definition of the problem and suggest a testable solution.

Let's look at the data

In the 1920s, the governor of New York, Al Smith, repeatedly endeavored to bring clarity to debate by simply remarking, "Let's look at the record." Here are two sets of numbers that should concern us.

The first has to do with the amount of money spent on software development. The Gartner Group reported that there were $3.7 billion spent worldwide in 2004 on application development tools. This was up more than 5 percent from that spent in 2003; it is therefore not unreasonable to assume that in 2006 somewhere in the neighborhood of $4 billion will be spent. Note that this does not include money spent on methodology, consulting, or other development expenses.

The second number is the software development project success rate. Dr. Philippe Kruchten, professor of software engineering at the University of British Columbia, has provided me with data from the Standish Group's CHAOS reports, shown in Table 1.

<table>
<thead>
<tr>
<th>Table 1: Software project success rates reported by Standish Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>First CHAOS report</td>
</tr>
<tr>
<td>&quot;Extreme CHAOS&quot;</td>
</tr>
<tr>
<td>Most recent CHAOS</td>
</tr>
</tbody>
</table>

My conclusion is that we are making progress on the success-rate front, but slowly. The improvement is about 1.7 percentage points a year and appears to be linear based on this small sample of data. If the current improvement rate continues, we should achieve a 50 percent success rate in the year 2014.

Software is the engine of our economy, at the root of almost everything we do. Our success rate has been creeping up in recent years, but one thing is sure: there has been NO breakthrough. Unless you consider almost "doubling" the success rate in nine years a breakthrough; I would claim that this effect is the result of a very low baseline for comparison. We may still be in the business of harvesting low-hanging fruit.

And there is another issue. If we are attacking the wrong set of problems, we may find after a while that our success rate stops improving. That is, current approaches are still squeezing out failures, but we may soon run into the law of diminishing returns. We haven't yet seen the success-rate curve flatten out, but there is no reason to expect that the linear progress we have seen over the last decade will continue indefinitely. In fact, we can be sure that at some point we will hit the wall. We just don't know when. Prudence dictates that we investigate alternatives before we are faced with even more unpleasant realities than we have today.
Where has all the money gone?

Most of the money over the past twenty-five years has been spent in two areas: methodologies and tools. Let's look at each of these.

Methodology has improved. We have seen a steady progression from functional decomposition to structured analysis, which in turn was replaced by object-oriented analysis, and then by modeling languages and notations. By the turn of the century, we saw acceptance of standards, among them the UML (Unified Modeling Language), adopted by the Object Management Group, and, later, the Rational Unified Process. All these things have helped. In particular, the notion of a solid architecture as the backbone of software systems has finally taken hold. More projects now use iterative development in one form or another. Recently, agile methods have added more evidence that we can tune our processes to be more productive and avoid the catastrophic outcomes that overtake many projects.

Concomitant with these improved methodologies are tools that really help. Enterprise software developers have seen configuration management and version control systems mature, and have also seen software development or programming environments become almost commodities. Automated testing methods have greatly improved. There is a higher degree of integration among these tools than ever before, and, compared to their ancestors, many of these tools truly save time and reduce errors.

And it cannot be disputed that increasingly sophisticated languages have emerged, from C++ to Java to C#. The libraries that come with each of these languages provide more and more opportunities for large scale reuse, something that has a very positive impact on overall software development productivity.

Productivity vs. success rate

Besides the success-rate metric, we also need to consider productivity. Is our software becoming more or less expensive to develop?

This is a difficult question, because the applications that we develop continue to become more and more rich and sophisticated in their scope and functionality. The software projects we now undertake are much more complex than software projects were ten or twenty years ago. There is no doubt that we expect more from software today than ever before.

In a sense, we have a "chicken and egg" problem here. Are we doing more daunting things today because we can? That is, are we attempting more ambitious projects because we have better methodologies and tools that enable their conception and execution? Or have the methodologies and tools just kept pace with the increasing challenges and degree of difficulty demanded by the external world? Frankly, I don't think there is a clear and unambiguous answer to this question.

What we can say is that when projects are successful, we have productivity levels that are much higher than heretofore. The mountains we are climbing are higher, and we are climbing them with proportionately fewer resources than in the past. That is the bright side of software development: better systems and higher productivity.
The dark side of the equation is project success rates. Why does the needle on success rates move so slowly? It is almost as though all the advances in methodologies and tools have helped us build more ambitious systems, yet there is some other factor that is keeping us from being successful more of the time. Perhaps there is something in the approach of the past several decades that is missing the point.

**The CEO's problem**

Let me put this another way. Assume for the moment that 75 percent of the $4 billion software tools money spent in 2006 is disbursed by the Fortune 1000 companies. Simple division then tells us that:

On average, each Fortune 1000 company spends $3 million on software development tools each year and sees less than a two percentage point improvement in project success rates.

I indented and bolded that sentence for emphasis: the $3 million must be spent just to keep up. While that money does work to increase productivity and allow the creation of more competitive applications, it is doing very little to raise the chances of success on the next set of projects.

Today we can say that roughly one project in three will be successful; 35 percent is a good estimate. If all goes well, it will take us another *eight years* to get to 50 percent.

Most CEOs are either concerned about this or they should be.

**Where's the problem?**

A moment's reflection leads us to believe that the problem is not where we first think it is. Fortunately, many minds greater than mine have been pondering the problem, and the answers are starting to emerge.

When I talk with my most knowledgeable colleagues in the field, I continue to get the same answer over and over again. We are getting better at producing software, but problems persist at the front end. That is, we are still having difficulties getting the requirements right. We do a poor job of specifying what it is that we want built. Requirements are often ambiguous, unclear, incomplete, or contradictory. And developers often guess what is desired, only to have to come back later in the cycle and rework their software once discrepancies have been discovered. We discover the problems too late, after much time and money has been spent building the wrong thing. Rework decreases overall productivity, and sometimes the rework is so overwhelming that the project fails.

Previous software development spending was not for naught, but there is now a wiser way to spend at least a portion of it. In effect, we are spending our money solving the wrong problem. No matter how good our "production tools" are in developing software, they cannot overcome a mangled or misdirected specification. Project success-rate improvement will almost certainly hit the wall unless we do a better job of specifying our requirements.

Figure 1 illustrates the danger.
Software development productivity and project success rates: Are we attacking the right problem?

Figure 1: While software development success will improve with better tools and techniques, errors in requirements contribute to a consistent, absolute project failure rate that cannot be reduced. This graph places that rate at 50 percent.

The floor at 50 percent may be lower; we don't know. But if requirements errors are indeed the root cause of a large percentage of our failures, not addressing this issue is going to cause us to cease to improve. Conversely, if we did make progress in this arena, we might see a dramatic improvement over the linear progress to date.

But what about requirements management tools?

But, you say, we have made progress in the area of requirements. Aren't there tools out there that address this problem?

We have made some progress. We no longer view paper napkins and Post-It Notes as legitimate vehicles for transmitting requirements. And the notion of use cases has dramatically improved our ability to describe what systems do from the users' point of view.

But note something interesting. All the effort has gone into two areas: managing requirements and something called "requirements traceability."

Requirements management is the art of capturing requirements, cataloging them, and monitoring their evolution throughout the development cycle. Requirements are added, dropped, changed, and so on, and we now have requirements management systems that allow us to keep track of all this. That is a good thing.

Traceability is a bit more ambitious. It attempts to link later-stage artifacts, such as pieces of a system and their test cases, back to the original requirements. That way, we can assess if we are actually meeting the requirements that were called out. This is a harder problem, but, once again, there has been substantial progress.

To all this I say, wonderful, but not good enough.
The crux of the matter

What good does it do to "manage" and "trace" 500 requirements if 250 of them are wrong in the first place?

Get the requirements right, and you only need to devote your valuable time and resources dealing with those correct requirements.

We have today very few tools that enable the specifiers of their systems to validate their requirements. That is, far too many of the requirements we are managing and tracing today are simply wrong: ambiguous, unclear, incomplete, or contradictory. We can now make sure that none of them are forgotten, but we are doing virtually nothing to improve the quality of the requirements.

I claim that until we address this problem, significant progress will not be made on the fundamental problem of software development project success.

Unfortunately, until very recently there have been no software tools that allow you to validate your requirements.

What about iterative development?

The folks who endorse iterative development -- and I am one of them -- often claim that requirements only evolve as a result of the iterative development process itself. This position is often misstated as, "You don't really know the requirements when you start." What are we to make of this?

Requirements do evolve as part of the process. But that is no reason not to try to validate the requirements you have in your hand at any time. If you believe that a system should do X, Y, and Z, you have a responsibility to state that as clearly as you can. X, Y, and Z should be validated to the extent possible.

Later on in the iterative development process, we may decide that requirement Y is irrelevant and drop it. At that time, we may add requirement W. You need to take just as much care in validating W as you previously did in validating X, Y, and Z. Requirements added later in the cycle are often the ones that cause the most trouble, as we frequently believe we don't have the time to spell them out as carefully as the original ones.

The conclusion is that there is no contradiction between iterative development and validating requirements.

The three-percent challenge

Let's take a different approach. Let's "follow the money." If we want to improve some area of practice, we should spend some money on it.

Here is a simple and revolutionary idea. Today, you are most likely spending no money at all to validate your requirements, yet that neglect is costing you money. I would like to suggest that in
the coming year, you devote a mere 3 percent of your software development tools budget -- that's roughly $100,000 for an average Fortune 1000 company -- to tools that improve the quality of the requirements you produce by validating them. It is a very inexpensive experiment to perform, because for $100,000, you can initiate several pilot projects using the most powerful tools we have today in this domain. I predict that you will see a positive ROI in the first year.

**Summary**

It's time to spend more money on getting requirements right. We have harvested the low-hanging fruit of managing and tracing requirements, largely because those were the easier problems to solve. But figuring out how to validate requirements is a harder problem. It should not surprise us that solving this problem will yield us a correspondingly higher reward. My claim is simple: If you want to see the needle move more quickly on software development project success, focus your efforts on validating your requirements.

**Notes**

1. For more information, see the Standish Group Website at [http://www.standishgroup.com/](http://www.standishgroup.com/)

© Copyright IBM Corporation 2006
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)