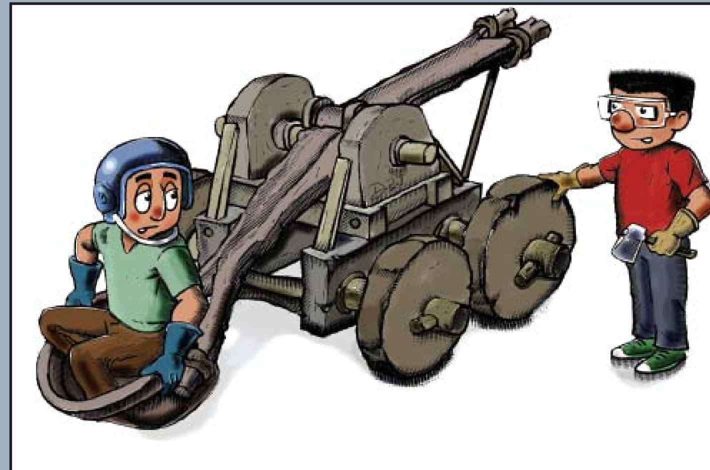
THE ADVENTURES OF RICKY & STICK



Fables in Software Acquisition

by

David Carney and David Biber

This work was prepared for the United States Air Force.
The Software Engineering Institute is a federally funded research
and development center sponsored by the U.S. Department of Defense.

Copyright 2006 Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

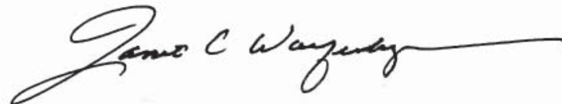
For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

To the members of the acquisition community:

On a short plane ride recently, I had the enjoyable experience of reading this little "comic book." It's a brief excursion into the complexities of software acquisition processes, using the metaphor of two well-meaning kids who, despite the best of intentions, always end up in trouble.

This book isn't an official guide to best practice, and it certainly isn't a textbook. But in a kind of off-beat way, it's an entertaining yet insightful look at some of the things that can really happen in software acquisition; each fable is based on true examples where our acquisition system has broken down.

I'd be surprised if, for most everyone in the acquisition business, there isn't something in this book that will ring a bell. For a pleasant respite from the standard official documents we all read daily, I recommend it highly.



JANET C. WOLFENBARGER,
Brig. Gen., USAF
Director, Acquisition Center of Excellence
SAF/ACE,
(703) 253-1333

Foreward
by David Carney

Several years ago, I had the good fortune to take a brief vacation from my normal chores of writing technical papers about software. I left the realm of data, executive summaries, issues, and findings, and spent several enjoyable days writing a short, rather tongue-in-cheek essay about the dangers and challenges of using commercial, off-the-shelf (COTS) software in government systems. The essay was written as a pastiche of Mao-Tse Tung's famous "Little Red Book" and my hope for the essay was simply that it would amuse a few people. I was thoroughly unprepared for how deeply it resonated in the DoD community. The Red Book has been reprinted numerous times, and I am still gratified to receive email from people for whom its "quotations," in the form of spurious Chinese aphorisms, are considerably more meaningful than any of my dry technical reports about the challenges of using commercial software.

It is always dangerous to try to repeat good fortune. However, I was recently asked to offer a few suggestions that address some high-level topics related to software acquisition. The request was for "something short and to the point," that would prepare beginning program managers for the delights that await when they find themselves stuck between demanding users, angry PEOs, and frustrated software engineers.

Perhaps against my better judgment, I chose to use an approach similar to the Red Book in writing this little book. While I have tried to keep the present volume from looking too much like a new version of the Red Book, there are some obvious similarities. It has (I hope) a certain humorous quality. Like the Red Book, it is premised on the idea that a brief, metaphoric approach can often convey more than verbose papers that are technically worthy, but aesthetically dull. And it is also patterned after well-known models, the most familiar of which was a comic strip fixture during the 1980s and 90s.

I decided that these little stories should be "fables," each of which includes a "moral" relevant to software, to acquisition, or to government programs. Possibly the most important point (and yet another similarity to the Red Book) is that these fables are based on real-world experiences: all of the situations in this book are inspired by programs that are known to me. Those programs encountered—and often foundered on—issues familiar to any observer of DoD acquisition: requirements, testing, integration, maintenance, commercial products, laws, mandates, funding, schedules, and, of course, bureaucracy. From observations of these programs, I selected some of the most representative as candidates for my anecdotal descriptive method. Aside from their topics, a common thread among these fables is that, for the Program Manager working in the complex and chaotic reality of government acquisition, the need is to keep sight of a few simple, fundamental realities. These realities are all too easy to dismiss as mere common sense, which they are. But in the frantic weeks before Milestone B, when

the world seems to be coming apart at the seams, it is amazing how easy it is to let such common sense fly out the window. At that point, a besieged Program Manager, no matter the level of experience, can sometimes make decisions that appear reasonable in the pressure cooker of the SPO, but in retrospect seem harebrained. It is precisely at that point that the Program Manager needs a lifeline to basic principles and calm rationality.

There are many topics that his book could address: common sense is in need on many fronts. From the large number of possibilities, I chose the following:

Testing and Modeling

Estimation and Metrics

Requirements

Integration and Interoperability

Deployment

Business processes

These are nothing more than starting points, of course, since they all blur, and it is impossible to keep a discussion of any of these topics from wandering into some of the others. I beg the reader's indulgence in this matter, since I wanted, in the spirit of fables going as far back as Aesop, to use each fable merely as an entry point for discussion and reflection. Thus, many of these fables will have multiple interpretations. This is not, I think, a fatal flaw: if the adventures of my hapless heroes provide a number of useful metaphors for the woes faced by Program Managers, so much the better. In the same vein, there is a certain redundancy in many of these tales that is not accidental. Familiar problems, even if seen many times before, can appear novel and strange when they pop up in unfamiliar contexts, and so telling the same story in different ways may have some value.

I began these ramblings talking about good fortune, and I have gone on too long. But it must be said that an additional pleasure for me was the enormous good fortune to collaborate with David Biber, whose brilliance and invention gave life, personality, and character to Ricky and Stick. He took my rather bland prose descriptions and made them so real that, by now, these likable rascals have truly become alive in my mind, and their exploits seem more like memories than fiction. His contribution to this work is inestimable.

Finally, I have tried to keep this work short. This was partly a pragmatic concern. A reader of the Red Book once complimented me that I had written it "so that it could be read on the flight from Washington up to Boston." Since that reader has recently been transferred back to the Pentagon, I hope that this little book will at least keep his attention on the return flight from Logan down to Reagan.

Software Engineering Institute
Carnegie Mellon University
October, 2005

THE ADVENTURES OF RICKY & STICK

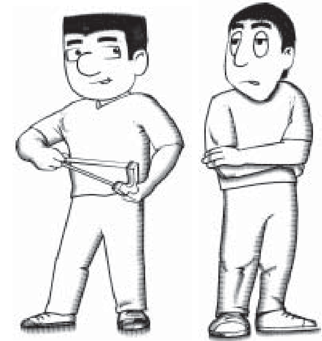
Fables in Software Acquisition

by

David Carney and David Biber

with Book Design and Production by Bob Fantazier

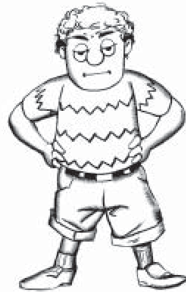
Ricky and Stick were best friends. They lived on the same street, and played together a lot. Ricky was a month older than Stick, and he always told Stick that this made him a lot smarter.



Their parents could not quite understand why Ricky and Stick got into trouble so often.

It seemed that they always started out with great ideas, but somehow, one thing led to another, and they ended up behind the eight-ball most of the time.

Other children lived in the same neighborhood as Ricky and Stick. There was one boy that they called Mean Wally. Wally was really a pretty nice kid. But he was always criticizing them, which they thought was a mean thing to do.



Gloria lived in the next block and was in the same class as Ricky and Stick. Ricky was annoyed that she was such a better student that he was, since their teacher, Mrs. Perillo, was always praising Gloria.

Ricky and Stick had another friend, a boy named Bob. Bob was usually nice to Ricky and Stick, and he often tried to help them on their projects. But they seldom took his advice. Bob was older than Ricky and Stick, and his hair was white all over. They called him Coconut Bob because of his white hair.

Table of Contents	Page
Testing and Modeling	7
Estimates and Metrics	15
Requirements	23
Integration and Interoperability	31
Deployment	39
Business Processes	47

DODI 5000.2, May 12, 2003

E5. ENCLOSURE 5

INTEGRATED TEST AND EVALUATION (T&E)

E5.1.1. The PM, in concert with the user and test and evaluation communities, shall coordinate developmental test and evaluation (DT&E), operational test and evaluation (OT&E), LFT&E, family-of-systems interoperability testing, information assurance testing, and modeling and simulation (M&S) activities, into an efficient continuum, closely integrated with requirements definition and systems design and development. The T&E strategy shall provide information about risk and risk mitigation, provide empirical data to validate models and simulations, evaluate technical performance and system maturity, and determine whether systems are operationally effective, suitable, and survivable against the threat detailed in the System Threat Assessment. The T&E strategy shall also address development and assessment of the weapons support equipment during the SDD phase, and into production, to ensure satisfactory test system measurement performance, calibration traceability and support, required diagnostics, and safety. Adequate time and resources shall be planned to support pre-test predictions and post-test reconciliation of models and test results, for all major test events. The PM, in concert with the user and test communities, shall provide safety releases to the developmental and operational testers prior to any test using personnel.

E5.1.2. The PM shall design DT&E objectives appropriate to each phase and milestone of an acquisition program. Testing shall be event driven and monitored by the use of success criteria within each phase, OT&E entrance criteria, and other metrics designed to measure progress and support the decision process. The OTA shall design OT&E objectives appropriate to each phase and milestone of a program, and submit them to the PM for inclusion in the Test and Evaluation Master Plan (TEMP). Completed IOT&E and completed LFT&E shall support a beyond LRIP decision for ACAT I and II programs for conventional weapons systems designed for use in combat. For this purpose, OT&E shall require more than an OA based exclusively on computer modeling, simulation, or an analysis of system requirements, engineering proposals, design specifications, or any other information contained in program documents (10 U.S.C. 2399 and 10 U.S.C. 2366, references (h) and (ae)).

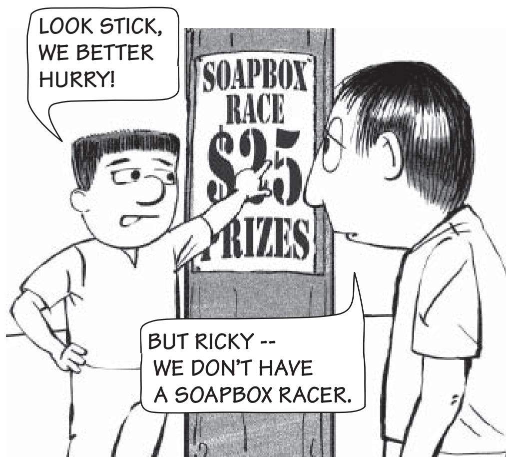
E5.1.3. T&E Strategy

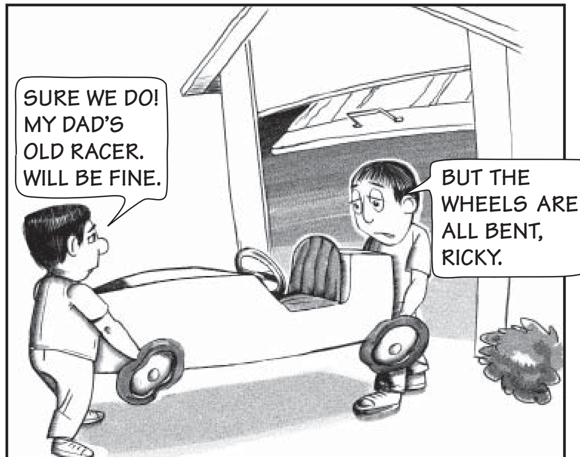
E5.1.3.1. Projects that undergo a Milestone A decision shall have a T&E strategy that shall primarily address M&S, including identifying and managing the associated risk, and that shall evaluate system concepts against mission requirements. Pre-Milestone A projects shall rely on the ICD as the basis for the evaluation strategy.

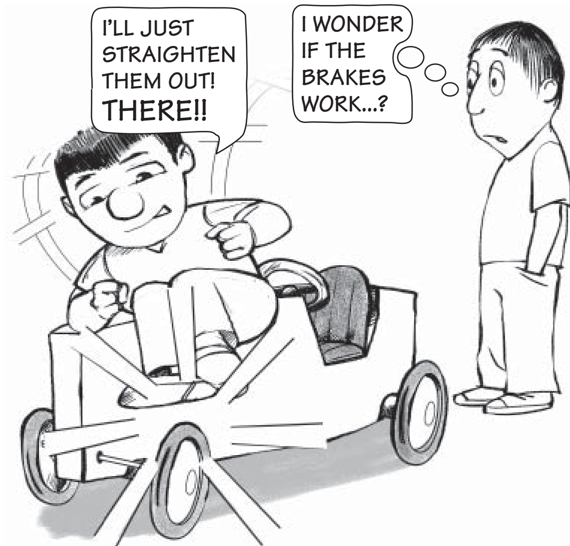
34 ENCLOSURE 5

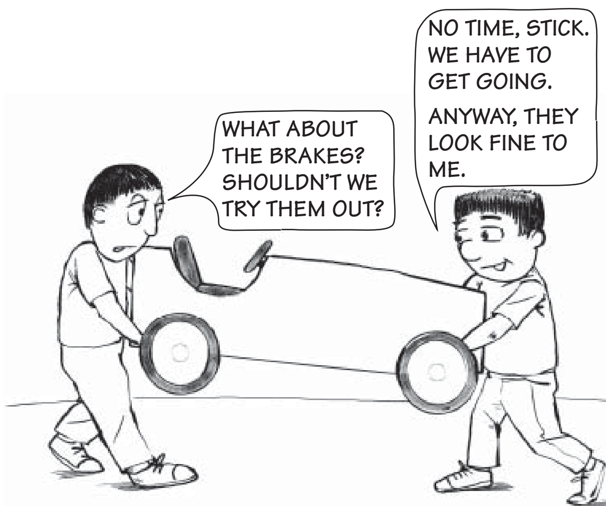
NO NEED TO TRY IT OUT— IT'LL WORK JUST FINE

Testing and Modeling

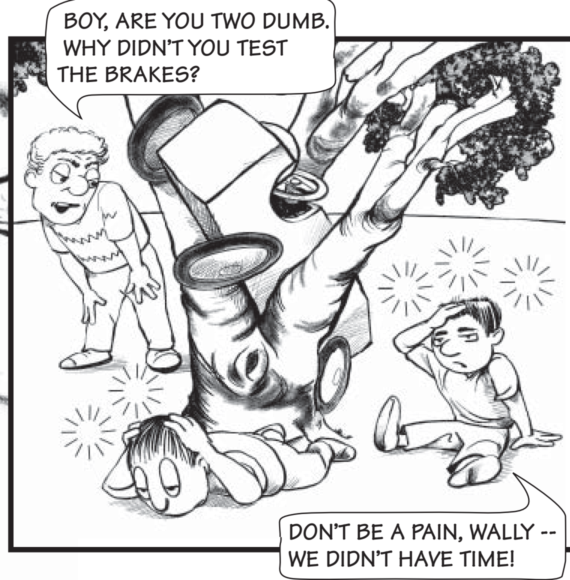

LOOK STICK,
WE BETTER
HURRY!
SOAPBOX
RACE
$25
PRIZES
BUT RICKY --
WE DON'T HAVE
A SOAPBOX RACER.


SURE WE DO!
MY DAD'S
OLD RACER.
WILL BE FINE.
BUT THE
WHEELS ARE
ALL BENT,
RICKY.


I'LL JUST
STRAIGHTEN
THEM OUT!
THERE!!
I WONDER
IF THE
BRAKES
WORK...?


WHAT ABOUT
THE BRAKES?
SHOULDN'T WE
TRY THEM OUT?
NO TIME, STICK.
WE HAVE TO
GET GOING.
ANYWAY, THEY
LOOK FINE TO
ME.

WALLY,
LOOK OUT!
WE CAN'T
STOP!!!

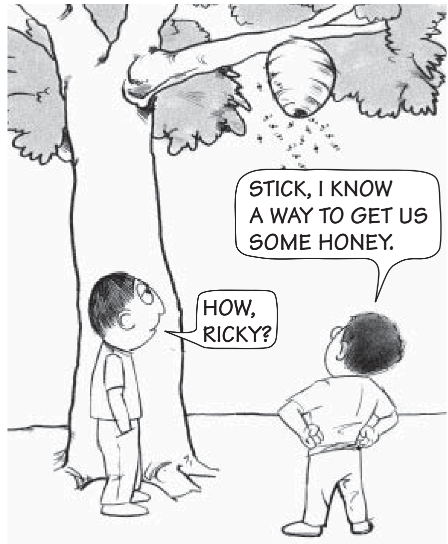

BOY, ARE YOU TWO DUMB.
WHY DIDN'T YOU TEST
THE BRAKES?
DON'T BE A PAIN, WALLY --
WE DIDN'T HAVE TIME!

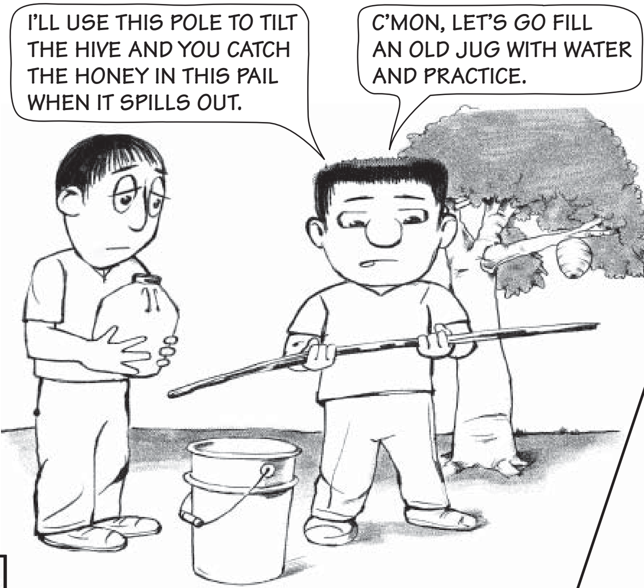
The stories about the pain and failure caused by inadequate testing are probably the best known tales in the software community; some of them have taken on a near-legendary status. Nor are they all legends, since testing really *is* a messy isssue. It's costly, it's time-consuming, and (so the theorists insist) nearly impossible to do perfectly. Even worse, testing tends, whether rightly or wrongly, to come late in the day, and for managers already behind schedule, it's often tempting to cut the testing resources to the bone.

But in yielding to that temptation, you're potentially adding to the painful tales and legends. You may really think that there's a compelling reason for skipping a crucial testing cycle. (Maybe, if you don't hurry up, you'll miss the race…). But chances are that by taking that route, by doing the real-world equivalent of operating a downhill racer without testing the brakes, the eventual crash is almost guaranteed. In retrospect, so were most of the DoD testing failures that have occurred over the years.

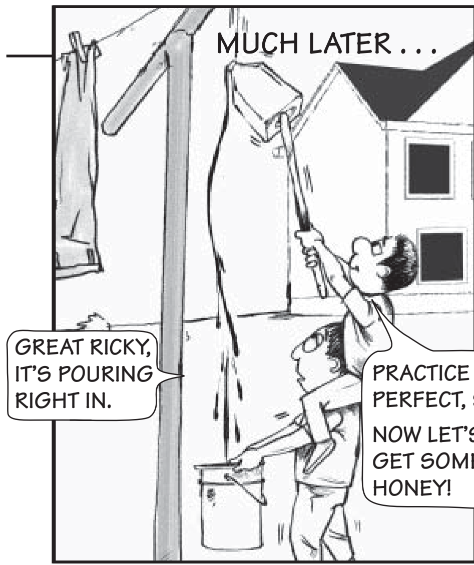
To be sure, there's no easy answer to the question: How much testing is enough? But there's a very easy answer when we get into a situation like Ricky and Stick: You've got to do at least *some*. And it must be real testing of the parts that really need to be tested.

Bottom line: No matter what schedule pressure you may be under, the outcome of a battle may someday depend on the system you're building. So if testing is getting squeezed, you may want to ask the contractor: "What are we risking by skipping this set of tests?" When lives are at stake, the "but we're way behind schedule" argument just isn't good enough.

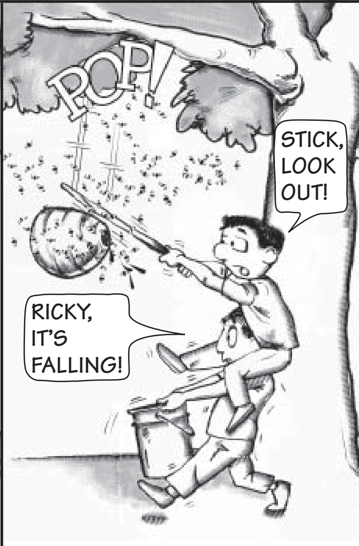

STICK, I KNOW A WAY TO GET US SOME HONEY.
HOW, RICKY?


I'LL USE THIS POLE TO TILT THE HIVE AND YOU CATCH THE HONEY IN THIS PAIL WHEN IT SPILLS OUT.
C'MON, LET'S GO FILL AN OLD JUG WITH WATER AND PRACTICE.

THIS IS HOW WE'LL DO IT.
CAREFUL RICKY, DON'T LET IT SPILL OUT TOO FAST.


MUCH LATER . . .
GREAT RICKY, IT'S POURING RIGHT IN.
PRACTICE MAKES PERFECT, STICK.
NOW LET'S GO GET SOME HONEY!

GET READY STICK, HERE IT COMES!

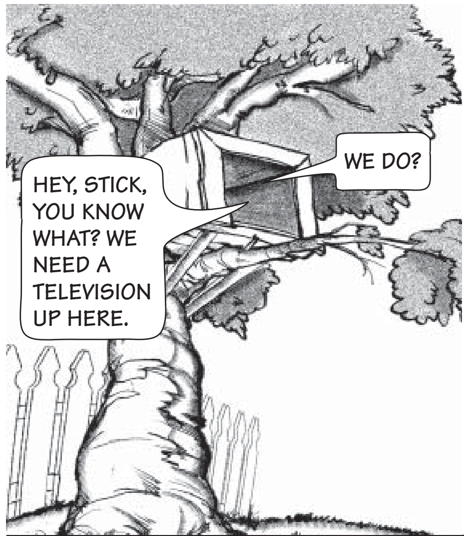

POP!
STICK, LOOK OUT!
RICKY, IT'S FALLING!

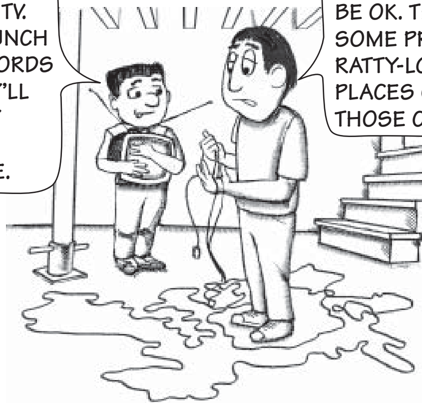
OOWWWWW!

Models are great, but they're not the real thing. And they can be very deceptive when misused. It's all too easy to use a model that is grossly oversimplified; even worse is to use a model that takes no account of the real risk conditions that will be present (like the risk of getting stung by a flock of angry bees!). So we're constantly in danger of letting the most optimal scenario be the basis of our models, convincing ourselves that we're modeling the true context that the system will encounter.

This pitfall is *sooo* prevalent in software development; it's almost too easy to construct happy models that will give you happy results. But happiness isn't what you want, truthfulness is. If the model doesn't truly mimic the conditions the system will face in the field, then none of the simulations you run will tell you much about how the system will actually perform.

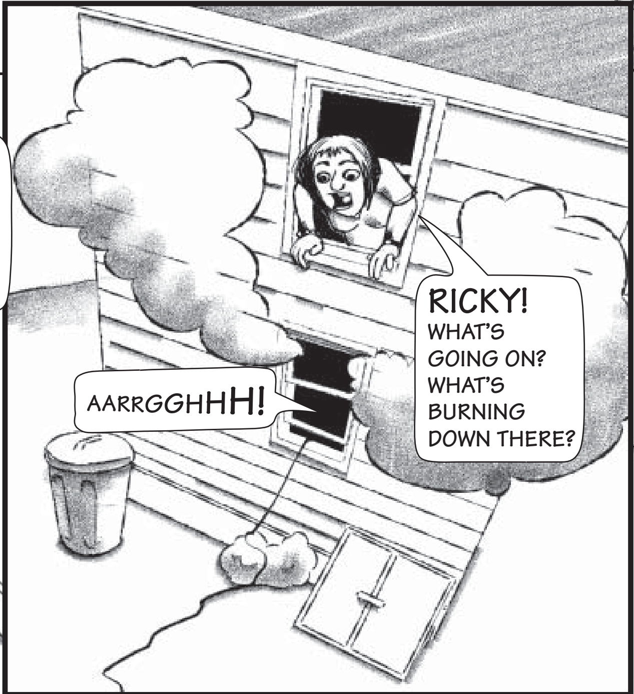
Bottom line: It's great if your testing plan includes using models and simulation. But don't model what you hope to find; model what will really be out there. Two good questions for the contractor might be: "What does that model leave out? And what's the delta between the model and reality?"


HEY, STICK, YOU KNOW WHAT? WE NEED A TELEVISION UP HERE.
WE DO?


SURE. NO ONE'S USING THIS OLD TV. IF WE HOOK A BUNCH OF EXTENSION CORDS TOGETHER, THEY'LL REACH FROM MY LIVING ROOM TO THE TREE HOUSE.
I HOPE IT'LL BE OK. THERE'S SOME PRETTY RATTY-LOOKING PLACES ON THOSE CORDS.

HEY GUYS, I SEE YOU'RE PLAYING WITH ELECTRICITY.
WANT ME TO TAKE A LOOK TO SEE IF EVERYTHING'S OK?
NO NEED, BOB, WE'RE JUST FOOLING AROUND.

OK, STICK, I'M GOING TO PLUG 'ER IN. GET READY TO WATCH SPONGEBOB!


AARRGGHHH!
RICKY! WHAT'S GOING ON? WHAT'S BURNING DOWN THERE?

HEY GUYS, I WONDERED WHAT THE COMMOTION WAS ABOUT.
NEXT TIME, DON'T BE SO QUICK TO TURN DOWN A HELPING HAND!

Given the realities of human nature and project schedules, it's a very common situation: we turn down an offer of outside assistance because (we tell ourselves) we want to keep on schedule, and some meddling outsider will only slow things down. But that's only part of the reason. What's really lurking in the back of our minds is that, if we let someone else look too closely (like letting Bob check out Ricky's wiring scheme), he might find something seriously wrong, which would *screw everything up.*

And that's precisely why the independent observer is there. Because everyone has blind spots; it's just a fact of life. The impartial and independent observer can often help you see through those blind spots; that's why the "I" in IV&V is so important. So while the temptation is to keep the IV&V guy from prying too much (lest he find something that you'd *really* prefer not to know about), a more productive approach, hard as it is, is to welcome him in and give him free rein to find what faults he can.

The moral is that the IV&V guy isn't the enemy. On the contrary, he's often the only one who can keep your house from burning down, just because a bit of wire has frayed and he's the only one who has noticed.

DODI 5000.2, May 12, 2003

E6. ENCLOSURE 6

RESOURCE ESTIMATION

E6.1.1. Cost Analysis Improvement Group (CAIG) Independent Life-Cycle Cost Estimates (LCCEs). The OSD CAIG shall prepare independent LCCEs per 10 U.S.C. 2434 (reference (ah)). The CAIG shall provide the MDA with an independent LCCE at major decision points as specified in statute, and when directed by the MDA. The MDA shall consider the independent LCCE before approving entry into SDD or into Production and Deployment. The CAIG shall also prepare an ICE for ACAT IC programs at the request of the USD(AT&L) or the ASD(C3I). A CAIG ICE is not required for ACAT IA programs. (DoD Directive 5000.4, (reference (bb)))

E6.1.2. Cost Analysis Requirements Description (CARD). For ACAT I and IA programs, the PM shall prepare, and an authority no lower than the DoD Component Program Executive Officer (PEO), shall approve the CARD. DoD 5000.4-M, reference (bc), specifies CARD content. For joint programs, the CARD shall cover the common program as agreed to by all participating DoD Components, as well as any DoD Component-unique requirements. The teams preparing the program office LCCE, the component cost analysis, if applicable, and the independent LCCE shall receive a draft CARD 180 days, and the final CARD 45 days, prior to a planned OIPT or DoD Component review, unless the OIPT leader agrees to other due dates.

E6.1.3. CCDR System. The CCDR system is the primary DoD means of collecting data on the costs and resource usage that DoD contractors incur in performing DoD programs. The Chair, CAIG, shall prescribe a format for the CCDR and the SRDR, and establish reporting system policies in DoD 5000.4.M-1, reference (aw). The Chair shall monitor the implementation of policy to ensure consistent and appropriate application throughout the Department of Defense. The Chair may waive the information requirements of Table E3.T3. of enclosure 3.

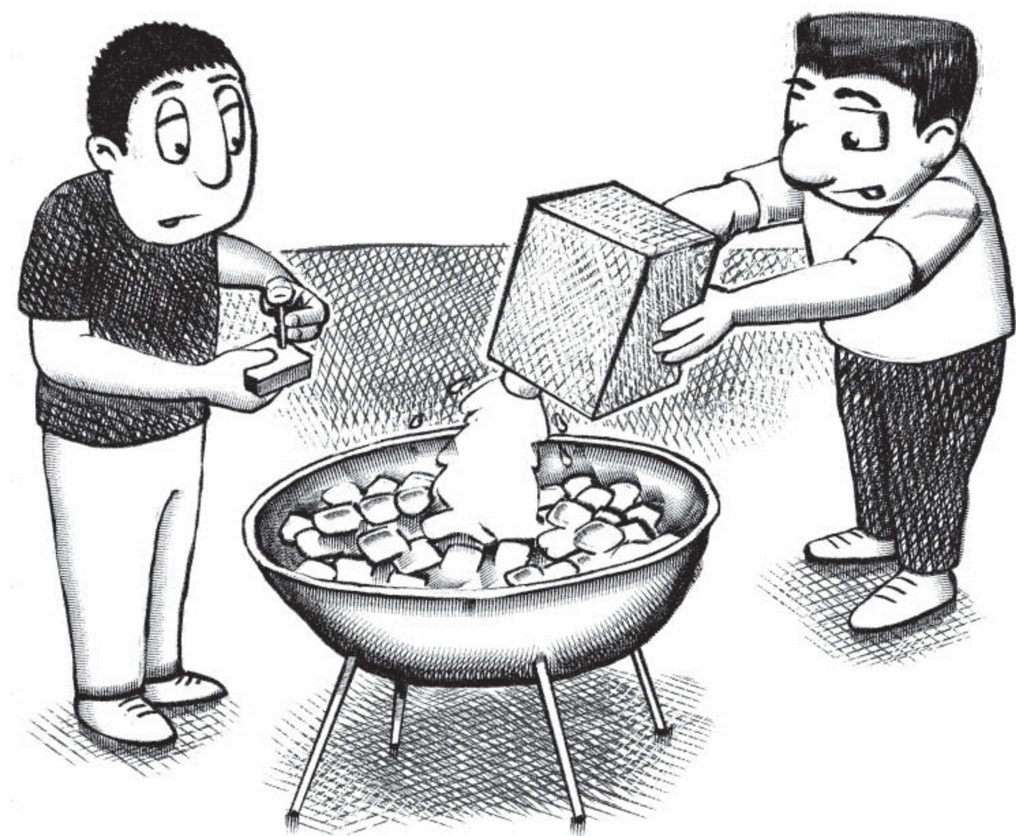
E6.1.4. CAIG Procedures. The DoD Component responsible for acquisition of a system shall cooperate with the CAIG and provide the cost, programmatic, and technical information required for estimating costs and appraising cost risks. The DoD Component shall also facilitate CAIG staff visits to the program office, product centers, test centers, and system contractor(s). The process through which the ICE is prepared shall be consistent with the following policies (reference (aw)):

E6.1.4.1. The CAIG shall participate in IPT meetings (Cost Working-level IPTs/Integrating IPTs/OIPTs);

41 ENCLOSURE 6

THERE, THAT SHOULD BE ENOUGH.

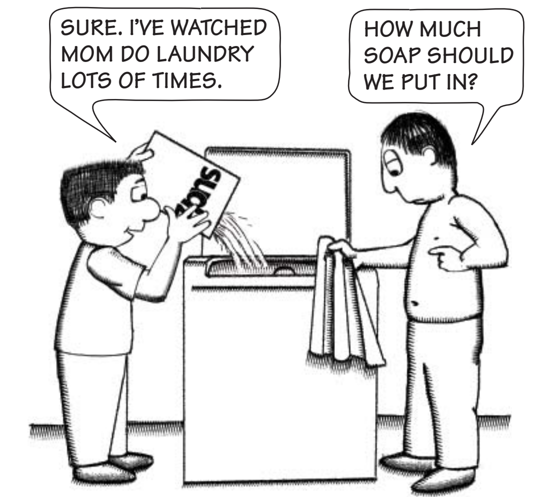
Estimates and Metrics



THIS IS GREAT, STICK. THIS BRIGHT GREEN PAINT WILL MAKE OUR PLANE PERFECT!
DO YOU REALLY THINK SO RICKY?

SURE IT WILL – OOPS!
SPLAT!
HEY!!! YOU GOT PAINT ON MY SHIRT!!! MY MOM WILL KILL ME!

DON'T WORRY, STICK, WE'LL JUST PUT IT IN THE WASHING MACHINE.
ARE YOU SURE YOU KNOW HOW TO RUN IT?


SURE. I'VE WATCHED MOM DO LAUNDRY LOTS OF TIMES.
HOW MUCH SOAP SHOULD WE PUT IN?
suds

WE BETTER USE A LOT! YOUR SHIRT IS A MESS.
THAT SHOULD DO IT.
suds

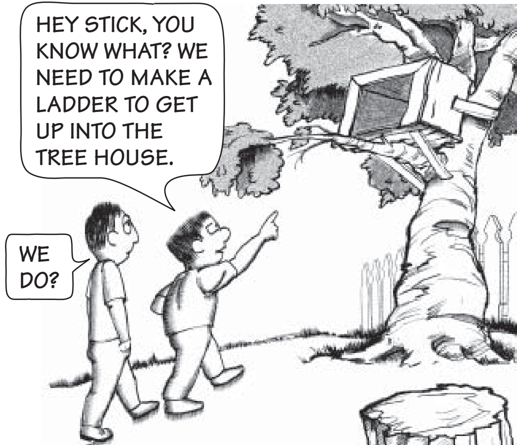
HELLLPPPP!!!!!

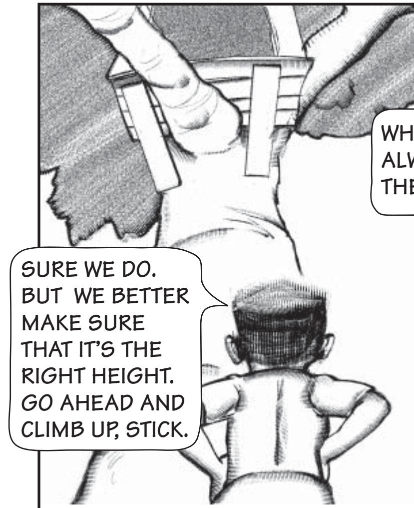
When we have no specific knowledge about the needed quantity of some resource, whether dollars, labor days, or laundry detergent, the only thing we can do is make an estimate. What with all of the pressure we are typically under, it's not uncommon to take an approach based on faith in our own ability to guess well and fueled by optimism; this is the "seems ok to me" syndrome. Sometimes we get lucky and everything comes out fine: a program manager, looking at some unfamiliar metric, with no context and no explanation, might make an excellent decision. And on a different day, Ricky might guess the right amount of detergent to use.

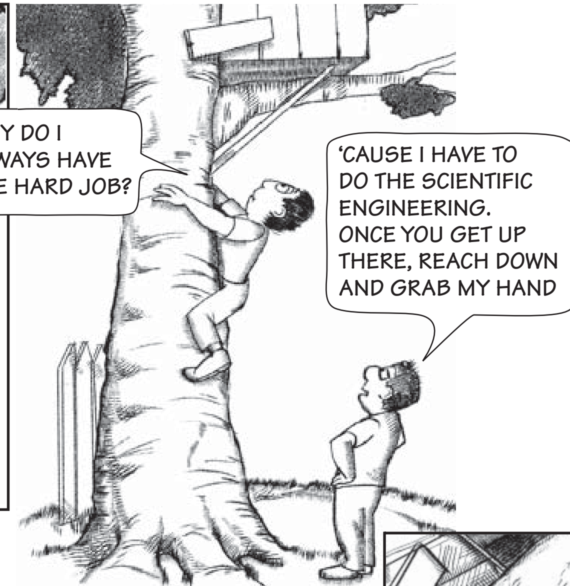
But, sadly, guesses like these often turn out to be wildly inaccurate. All too often, the floor gets sopping wet and Mom has to call the repairman.

It's really okay to opt for prudence, especially if there's no other guide. Ricky (and, it seems, a large number of teenagers) *could* take the time to read the label on the detergent box. Program managers, faced with a difficult decision and nothing on which to base it, *could* seek out assistance. Perhaps there's some website, some guidebook, some other source of information available somewhere, with advice, based on experience, to which you can turn.

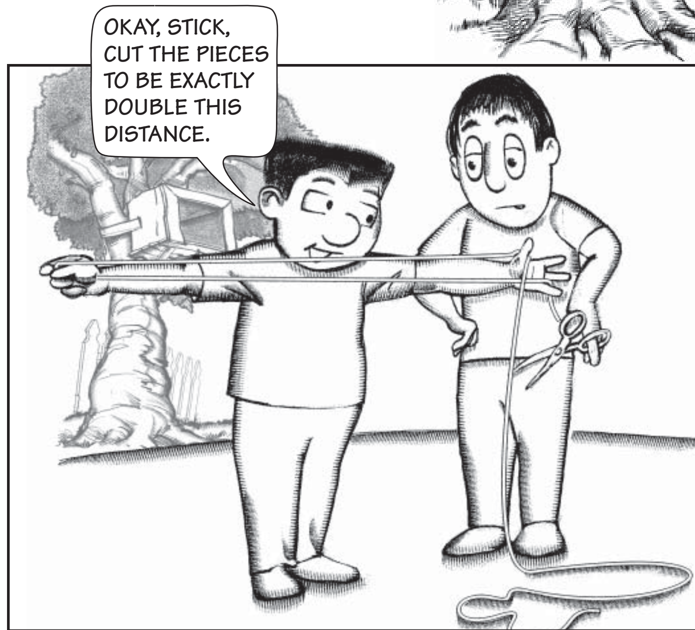
In brief, wisdom is better than guesses, and there's a lot of wisdom out there that's often ignored. The wisdom that exists may only be partially applicable, and there may still be a lot of guesstimation to do. Or maybe there's no such wisdom to be found at all. But in that case, you're no worse off than when you started. And no one can later call you on the carpet and say: "Why didn't you ask Bob?"


HEY STICK, YOU KNOW WHAT? WE NEED TO MAKE A LADDER TO GET UP INTO THE TREE HOUSE.
WE DO?


SURE WE DO. BUT WE BETTER MAKE SURE THAT IT'S THE RIGHT HEIGHT. GO AHEAD AND CLIMB UP, STICK.


WHY DO I ALWAYS HAVE THE HARD JOB?
'CAUSE I HAVE TO DO THE SCIENTIFIC ENGINEERING. ONCE YOU GET UP THERE, REACH DOWN AND GRAB MY HAND

I'M REACHING AS FAR AS I CAN, RICKY!
I CAN ALMOST TOUCH YOUR HAND, STICK, SO OUR TWO ARMS' LENGTH IS EXACTLY HOW LONG THE LADDER SHOULD BE.


OKAY, STICK, CUT THE PIECES TO BE EXACTLY DOUBLE THIS DISTANCE.

#?!#%! . . . DARN, STICK, YOU CUT THE ROPE TOO SHORT!

How many of us *haven't* done something like this? And always, deep down, we know we're being as dumb as Ricky was. Usually, it doesn't do all that much harm. But, every now and then, people who are otherwise rational really do hold their arms out to measure a picture, walk across the room trying to hold their arms steady, and then bang a nail into a wall to hang the picture on. The results are usually embarrassing, and sometimes *really* annoying.

Yet it's not unheard of, in a big, expensive, serious DoD program, for someone to do pretty much the same thing, and use a thoroughly ad hoc method for determining a metric that needs to be more precise. We've all probably witnessed a scene where someone with precious little coding experience says "No problem—we can get that new module written and debugged in a couple of days, for sure!" And then, it's not only foolish, it can do lots of harm.

A way to avoid the trap is to realize that metrics are not second-class citizens. Doing the sexy engineering tasks is important, but getting valid metrics on those tasks shouldn't be an afterthought. Another pitfall is haste: we're often in a hurry and don't want the delay that careful measurement demands. For big projects (which tend to be late almost by definition), enforcing a rigorous metrics program can slow things up to an alarming degree.

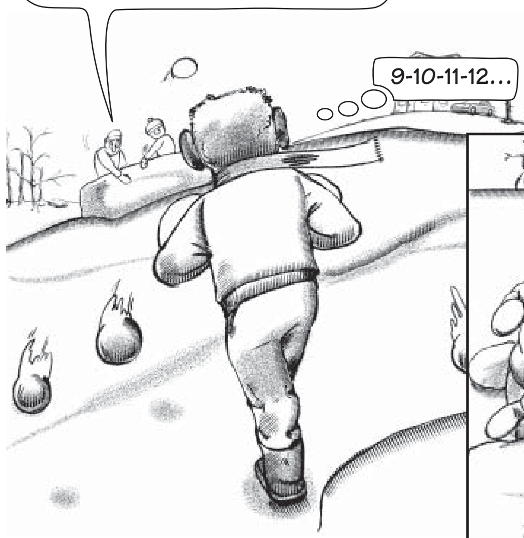
But that's the way it is, and it can't be changed. If we skimp on getting sound numbers on which to make sound decisions, if we accept rough figures as though they were accurate, and let guesses count as gospel, then we'll fall even further behind, because our arms-length guess was screwy, the ladder won't reach, and we'll have to start all over again.

Bottom line: With your program's future on the line, it is prudent to ask your contractor some hard questions about the relevance and accuracy of whatever figures are quoted to you. Said differently, do you *really* think he can keep his arms that steady as he walks across the room?

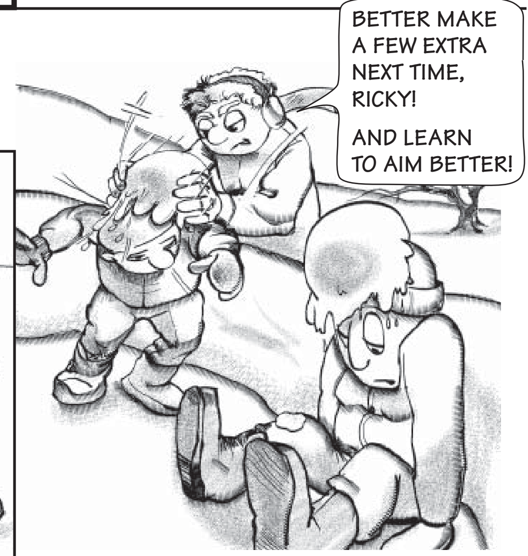
C'MON, STICK, WE'VE GOT TO CLOBBER WALLY WITH SNOWBALLS WHEN HE SHOWS UP. WE NEED TO HAVE ENOUGH ON HAND TO MAKE HIM BEG FOR MERCY
HOW MANY IS THAT?

WELL, WE DON'T ALWAYS HIT EVERY ONE OF OUR SHOTS, SO LET'S MAKE TEN SNOWBALLS EACH. THAT'LL MAKE HIM CRY.
THAT'S A LOT OF SNOWBALLS, RICKY. WE BETTER WORK FAST BEFORE HE SHOWS UP.

THAT'S IT, STICK. A NUCLEAR STOCKPILE OF MEGATON SNOW MISSILES.
I THINK THAT'S WALLY COMING!


YOU'VE HAD IT WALLY! WE'VE GOT TWENTY KILLER SNOWBALLS AND WE'RE GOING TO BLAST YOU!
9-10-11-12...

. . . 18, 19, 20. TWENTY SHOTS, TWENTY MISSES. NOW, YOU'RE BOTH OUT OF AMMO!


BETTER MAKE A FEW EXTRA NEXT TIME, RICKY!
AND LEARN TO AIM BETTER!

A software project plan is little more than a codified set of assumptions, expectations, and hopes. It typically contains some number of estimates based, more often than not, on optimism. Yet the sheer statistics of software failures, especially IT failures, would suggest that a healthy dose of caution, and probably of pessimism, would be more appropriate.[1]

Ricky thought he was being appropriately cautious when he estimated that, because he and Stick *didn't always hit every one of their shots,* they'd need ten snowballs each. But his reasoning was upside-down. He never once considered how many of their shots actually *did* hit the target; as it turned out, this was certainly fewer than one in ten. In other words, the number of snowballs wasn't significant, but only the number of hits. (The reader will already have noted that, given Ricky and Stick's throwing skill, they probably shouldn't have been planning to barrage Wally with snowballs in the first place. But that's a different fable.)

The moral is that we need to stop counting the wrong things, and start counting the right things. Easier said than done, perhaps. But it shouldn't be all that difficult to take a long hard look at whatever initial estimates you currently have, and wonder "What are these numbers based on? What *aren't* these number based on?" There's a good chance that, somewhere in the answers to those questions, you'll be saving yourself from getting walloped by a whole lot of snowballs.

[1] There are many sources for such statistics. One source often referenced is Lyytinen, K. and Hirschheim, R., (1987), "Information Systems Failures: A Survey and Classification of the Empirical Literature," Oxford Surveys in Information Technology, Vol 4. However, there are many others, whose numbers vary somewhat, but whose essential conclusions do not.

DoD 4140.1-R, May 23, 2003

C2.3.2.3. The using DoD Component shall include the current MME code on supply support requests to the other DoD Components.

C2.3.2.4. If weapon system application files, which are described in section C7.4., below, indicate that a secondary item has multiple applications, it normally shall be assigned the highest applicable essentiality code. A secondary item may have a different essentiality code for each of its end item applications.

C2.3.2.5. The DoD Components shall review and validate the assignment of essentiality codes periodically to ensure that they reflect the current status of the items.

C2.3.2.6. The using DoD Component shall provide application data to DoD IMMs in a timely fashion or update codes previously provided when criterion change or when items or weapon systems become obsolete. Rejected transactions transmitting application data shall be researched, corrected, and resubmitted on a timely basis.

C2.3.2.7. The DoD Components shall annually reconcile the Service weapon system application file and the DLA Weapon System Support Program database.

C2.3.2.8. To identify how items are being managed at retail supply activities, the DoD Components shall use the reasons for stockage categories delineated in Appendix 3.

C2.4. ITEM SUPPORT GOALS

C2.4.1. Requirements

C2.4.1.1. Item support goals shall be established for all DoD secondary items to ensure that the supply system optimally uses available resources to meet weapon system and equipment performance objectives and personnel readiness objectives at the least cost. Establishing these goals is required regardless of the source or method of support, e.g., organic, inter-governmental, private contractor, or partnership.

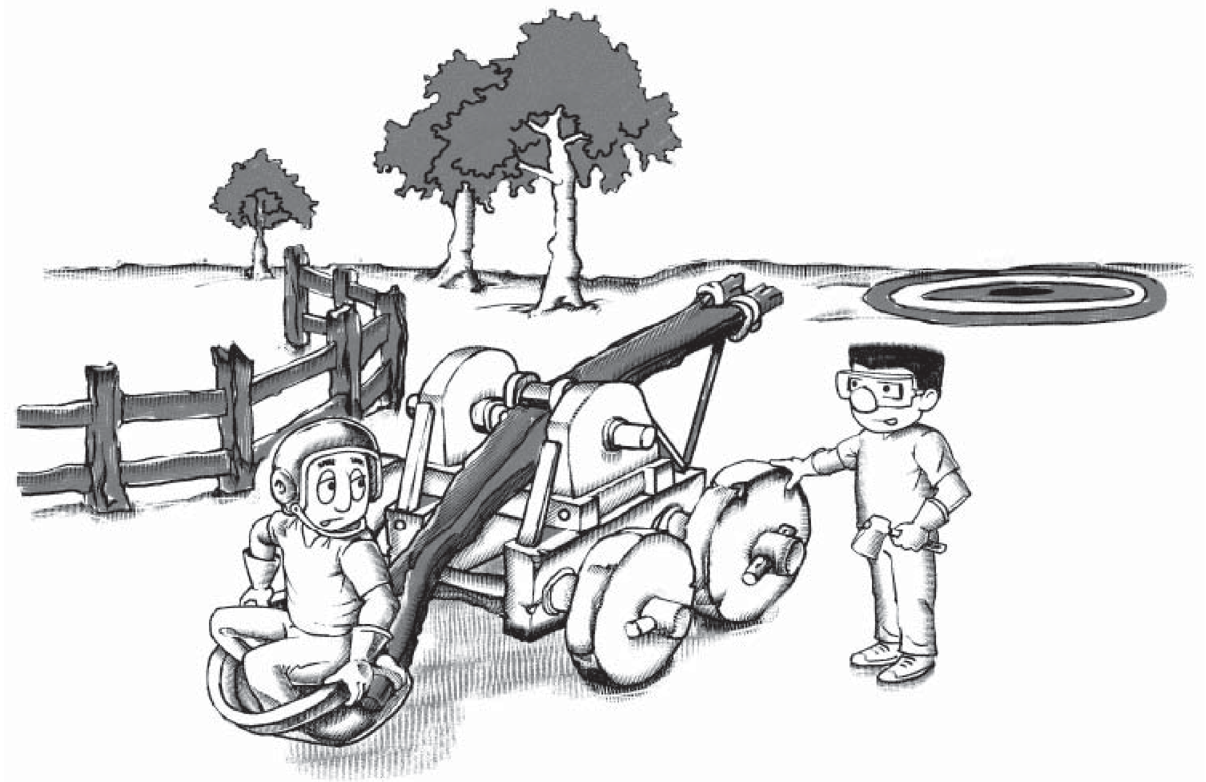
C2.4.1.2. Item support goals should be based on the performance agreements negotiated with customers or, where no agreement exists, on the enterprise metrics that the DoD Components have adopted for supply support.

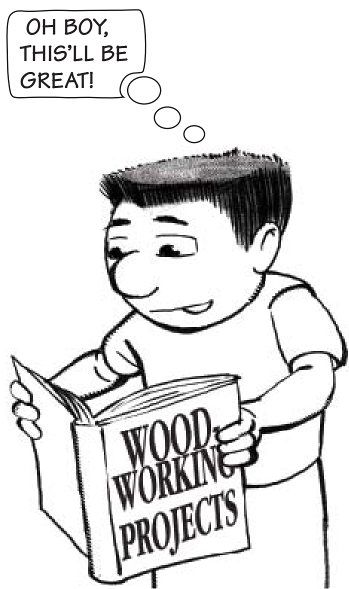
C2.4.1.3. The objective in establishing item support goals is to provide logistics managers with quantitative targets that they may use to improve supply planning, asset allocation, and the contribution of limited inventories and limited

32 CHAPTER 2

THAT MAY BE WHAT YOU WANT, BUT IT AIN'T WHAT YOU'RE GONNA GET . . .

Requirements




OH BOY, THIS'LL BE GREAT!
WOOD WORKING PROJECTS

WHAT ARE YOU DOING, RICKY?
I'M BUILDING A DOGHOUSE! C'MON, STICK, GIVE ME A HAND.

ALMOST THERE, STICK. NOW ALL WE NEED TO DO IS MAKE THE TWO STEPS.
TWO STEPS???

DARN! THESE STEPS DON'T FIT! NOW WHAT?
WHY DOES IT NEED TWO STEPS, ANYWAY?

BECAUSE IT SAYS SO HERE IN THE DIRECTIONS. THIS DOGHOUSE HAS GOT TO FOLLOW THE PLAN EXACTLY!
WOOD-WORKING PROJECTS

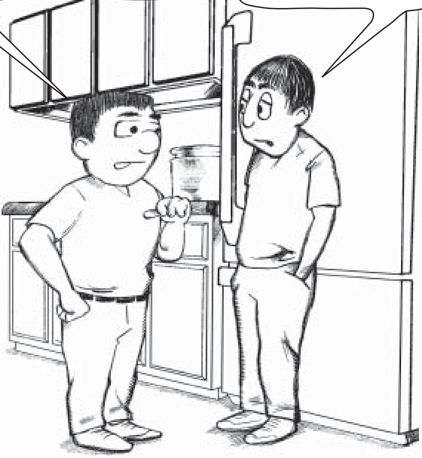
STILL NOT THERE YET, STICK. WE'VE GOTTA KEEP TRYING!
UH, RICKY...
Build a Dog Hous
in 2 Ste

Sometimes requirements that don't make any sense creep into a program. Usually, no one knows where they came from, nor why they're there. They may be based on misunderstandings, or on conditions that have become obsolete. Or maybe they were just a nasty gift from the Bad Requirements Demon. In any case, these are often the very requirements that twist a program into a pretzel.

So it's perfectly reasonable to periodically reconsider the validity of requirements, either to be sure that they're still operative, or to verify that their respective interpretations by the builder and end user is consistent: it's amazing how often such a reinspection will turn up a surprise or two.

It may seem obvious, for instance, that Ricky's notion of front steps for a doghouse was based on a misreading of the plans. But Ricky wasn't being any sillier than many real-world counterparts: some software requirements specs have sternly dictated versions of COTS products that are several releases out of date, and more than a few requirements have been diametrically opposite to what the end user has requested.

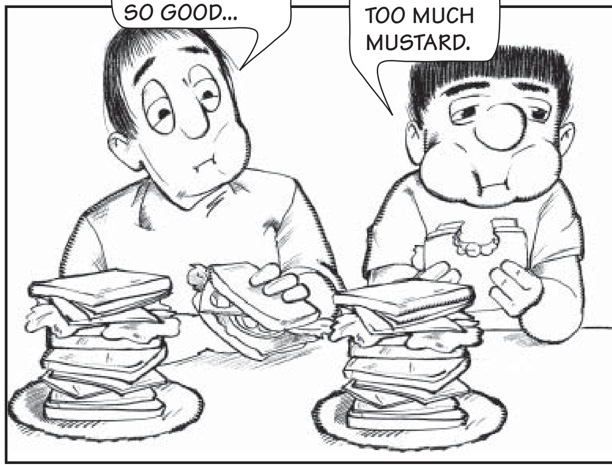
Bottom line: A periodic review of the requirements asking: "Do each of these still apply? Has anything changed?" is a valuable exercise that can help discover obsolete requirements as early as possible. By doing so, you'll avoid expending (and wasting) a huge amount of effort in needlessly trying to meet them.

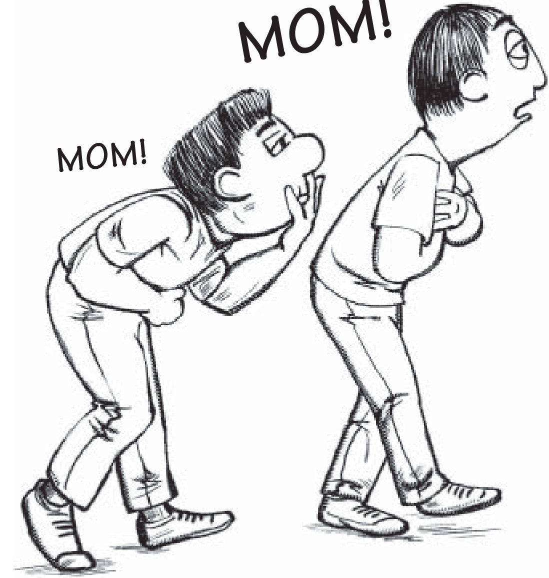

HEY, STICK, I'M GETTING HUNGRY AND MOM'S NOT AROUND. I'M GOING TO MAKE US SOME SANDWICHES.
WHAT KIND OF SANDWICHES, RICKY?

LET'S SEE. WHAT DO I FEEL LIKE EATING? HOW ABOUT SOME HAM, CHEESE, TUNA FISH, MUSTARD, PASTRAMI, SALAMI, HORSERADISH, MAYONNAISE, TURKEY, AND KETCHUP?

ARE YOU SURE THAT'LL ALL GO TOGETHER?
SURE, STICK, WHY NOT? EVERYTHING TASTES GREAT, SO THEY'LL ALL TASTE GREAT TOGETHER!

HERE WE ARE, STICK. POP OPEN A COUPLE OF CANS OF SODA AND WE'LL HAVE A FEAST.


RICKY, THIS DOESN'T TASTE SO GOOD...
MAYBE A LITTLE TOO MUCH MUSTARD.


MOM!
MOM!

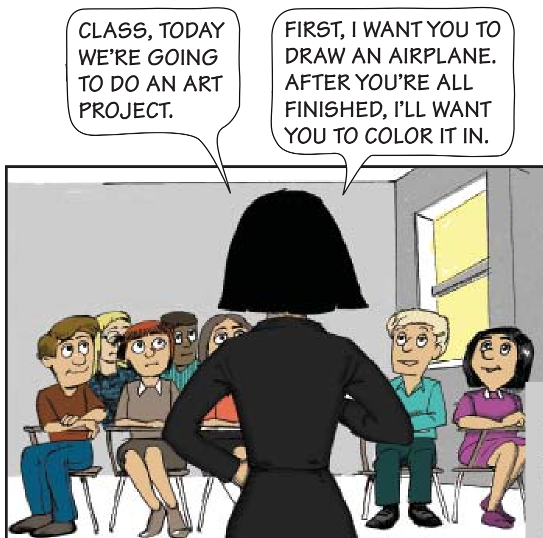
The process of software system development often involves an ongoing process of refining requirements. And that refinement process is very susceptible to requirements creep, growth, and explosion. It's the same danger that Mom is always warning Ricky about: "Your eyes are too big for your stomach," she says, which he usually ignores.

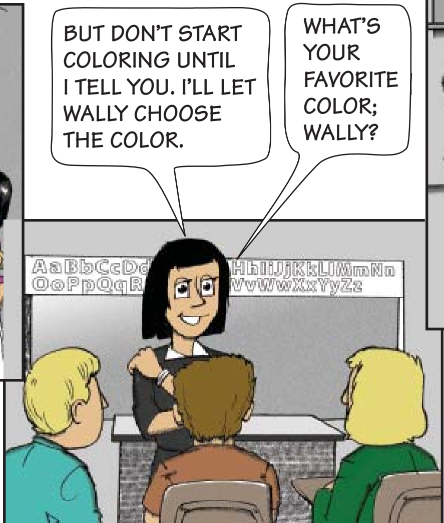
Yet the rest of us are often deaf to that same warning. It all starts when a bunch of people start out with a good idea: "Let's get rid of these overlapping, obsolete, redundant systems!" So a project begins, and the early requirements are defined. Then everyone goes over to the Dark Side: "Now that we see what we've started, why don't we reengineer ALL of our seventy-three thousand processes into one central, unified, joint, all-purpose, galactic, do-it-all, never-have-to-worry-again INTEGRATED SYSTEM!"

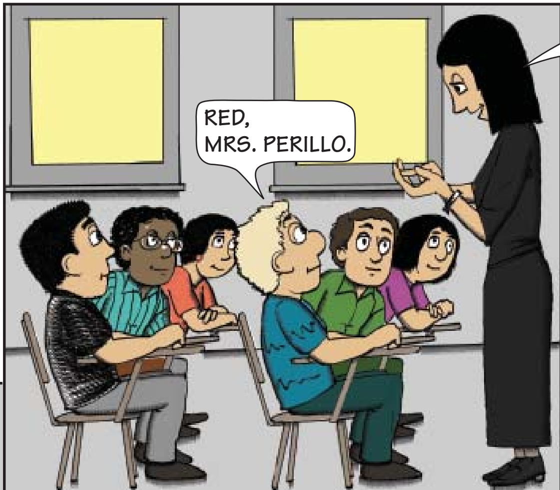
What's happened is that the understandable desire to eliminate redundancy and incompatibility has transformed itself into a greedy desire for something that ignores practicality and precedent. We get giddy with possibilities, and imagine a cosmically large system whose humongous list of functional requirements makes *any* development effort prone to failure. And in those giddy moments, it's easy to forget that the pages of acquisition history are littered with tales of failed programs slain by impossible requirements.

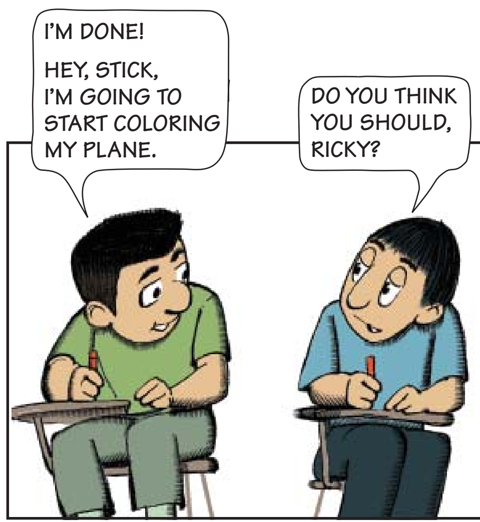
(And this same scenario also shows there are lots of things that are excellent when taken individually, but awful when put together willy-nilly. A different moral, perhaps, but one worth noting.)

The lesson is that, even taking into account the incredible flexibility of software, a coherent system needs some internal integrity and boundedness to it. More important (from the viewpoint of the poor soul who has to manage its development), a system's requirements should ideally reflect some comprehension of whether those requirements can be satisfied.

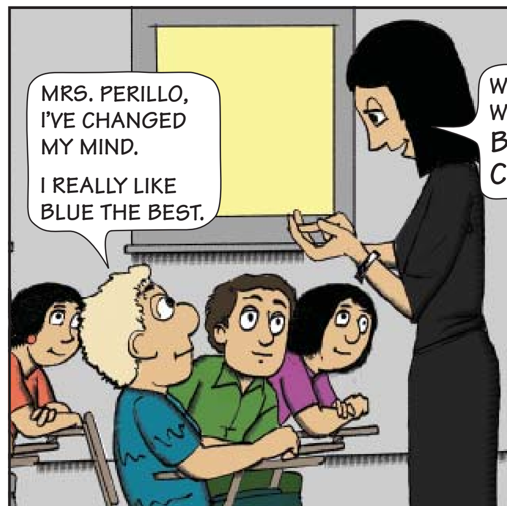

CLASS, TODAY WE'RE GOING TO DO AN ART PROJECT.
FIRST, I WANT YOU TO DRAW AN AIRPLANE. AFTER YOU'RE ALL FINISHED, I'LL WANT YOU TO COLOR IT IN.

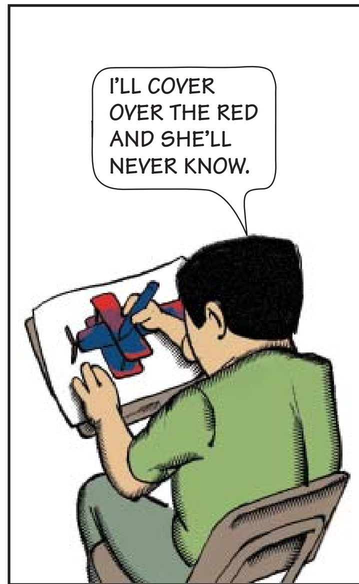

BUT DON'T START COLORING UNTIL I TELL YOU. I'LL LET WALLY CHOOSE THE COLOR.
WHAT'S YOUR FAVORITE COLOR; WALLY?


RED, MRS. PERILLO.
EXCELLENT, NOW START DRAWING, BUT DON'T START COLORING YET!


I'M DONE!
HEY, STICK, I'M GOING TO START COLORING MY PLANE.
DO YOU THINK YOU SHOULD, RICKY?

YEAH, STICK, I'LL BE THE FIRST ONE FINISHED . . .


MRS. PERILLO, I'VE CHANGED MY MIND.
I REALLY LIKE BLUE THE BEST.
WHY, THAT'S FINE, WALLY. KEEP DRAWING, BUT DON'T START COLORING YET!



People have the annoying habit of changing their minds. When these people are end users of software systems, then requirements have the annoying habit of mutating.

Given this reality, some of the best advice about dealing with requirements for today's information systems is to maintain flexibility as long as feasible. Sometimes the ideal strategy is lots of prototyping, to gain buy-in from end-users. Sometimes it's possible, using iterative cycles, to delay freezing the requirements until some fairly late point. But it's almost always a poor idea to make a ton of early commitments if there's no compelling reason to do so.

Because if we do make some early commitment, we often don't (or can't) be sure what that commitment implies. And then when things change, which they always do, recovery is sometimes possible, but sometimes it's not. For Ricky, the first time things changed, he got away with it, by switching from red to dark blue. But when things changed again, and he had to convert that dark blue to pale yellow, he was lost. All he had was a useless picture of a blue airplane—there was no way to erase the blue crayon, and no recovery was possible.

The lesson is that while it's attractive to make early choices and "nail down the requirements," it's not always the wisest course. That approach can sometimes save a lot of time and money, true. But as you're thinking of taking that step, you might also take the trouble to determine from your stakeholder community whether all the assumptions that underpin the requirements are still applicable. Because it may be that there's a Gloria in your future, unseen right now, but just waiting for a chance to say: "Yellow, Mrs. Perillo."

Department of Defense

DIRECTIVE

NUMBER 4630.5
May 5, 2004

ASD(NII)/DoD CIO

SUBJECT: Interoperability and Supportability of Information Technology (IT) and National Security Systems (NSS)

References: (a) DoD Directive 4630.5, "Interoperability and Supportability of Information Technology (IT) and National Security Systems (NSS)," January 11, 2002 (hereby canceled)
(b) Subtitle III of title 40, United States Code, as amended
(c) Sections 2223 and 2224 of title 10, United States Code, as amended
(d) DoD Directive 5000.1, "The Defense Acquisition System," May 12, 2003
(e) through (n), see enclosure 1

1. REISSUANCE AND PURPOSE

This Directive:

1.1. Reissues reference (a) to update DoD policy and responsibilities for interoperability and supportability of Information Technology (IT), including National Security Systems (NSS), and implements DoD Chief Information Officer's (DoD CIOs) responsibilities contained in references (b) and (c).

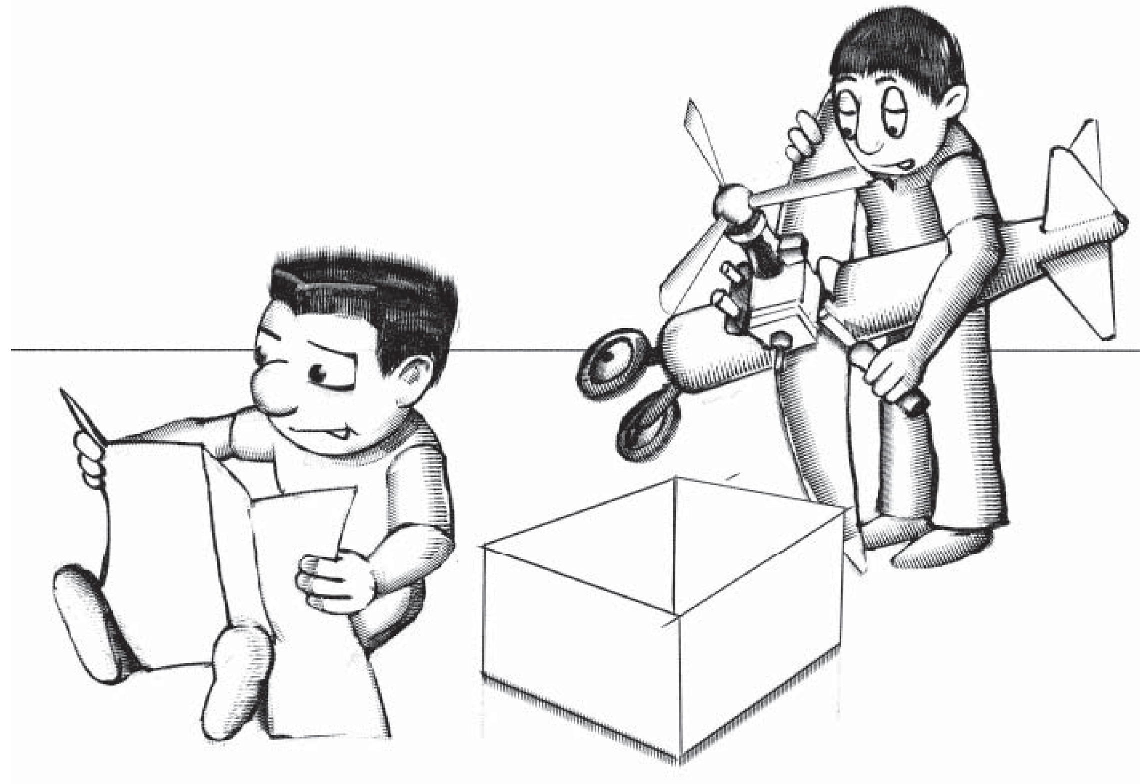
1.2. Defines a capability-focused, effects-based approach to advance IT and NSS interoperability and supportability across the Department of Defense.

1.3. Establishes the Net-Ready Key Performance Parameter (NR-KPP) to assess net-ready attributes required for both the technical exchange of information and the end-to-end operational effectiveness of that exchange. The NR-KPP replaces the Interoperability KPP and incorporates net-centric concepts for achieving IT and NSS interoperability and supportability.

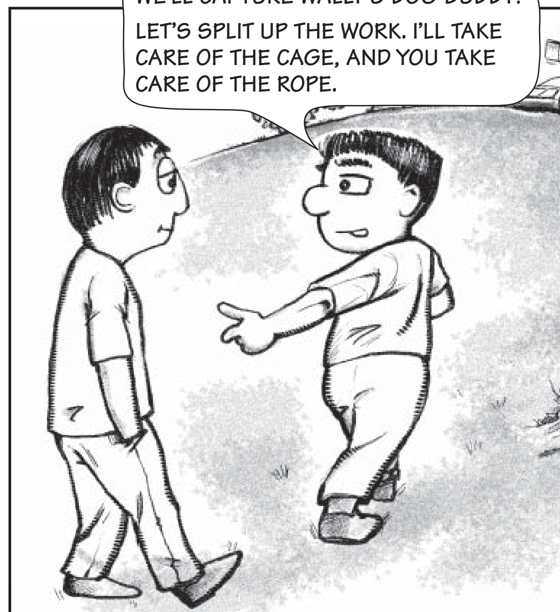
1

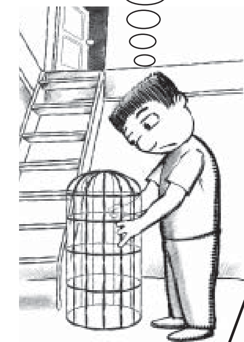
SOME ASSEMBLY REQUIRED

Integration and Interoperability



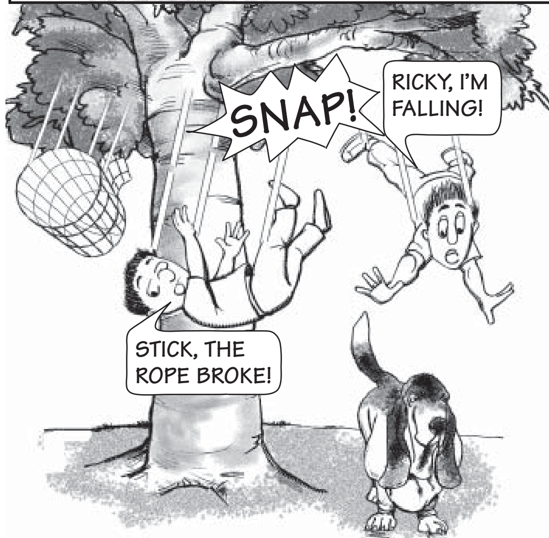
HEY, STICK, I'M BORED. LET'S GO AND CAPTURE A WILD ANIMAL.
HUH? WHAT WILD ANIMAL?


WE'LL CAPTURE WALLY'S DOG BUDDY!
LET'S SPLIT UP THE WORK. I'LL TAKE CARE OF THE CAGE, AND YOU TAKE CARE OF THE ROPE.


THIS OLD BIRD CAGE SHOULD WORK FINE!

THIS THICK TWINE SHOULD BE FINE!

HE'S ALMOST UNDERNEATH, STICK!
LET'ER GO!"
RICKY, THE CAGE LOOKS AWFULLY HEAVY . . .


SNAP!
RICKY, I'M FALLING!
STICK, THE ROPE BROKE!

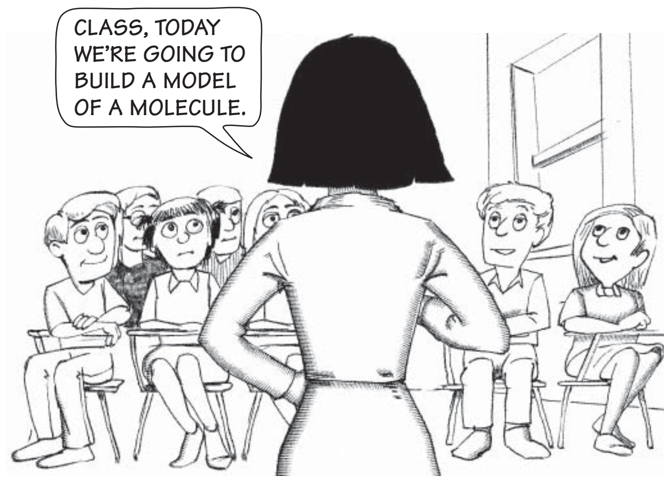
I SAID 'ROPE', NOT TWINE!
YOU DIDN'T TELL ME IT WOULD WEIGH A TON!

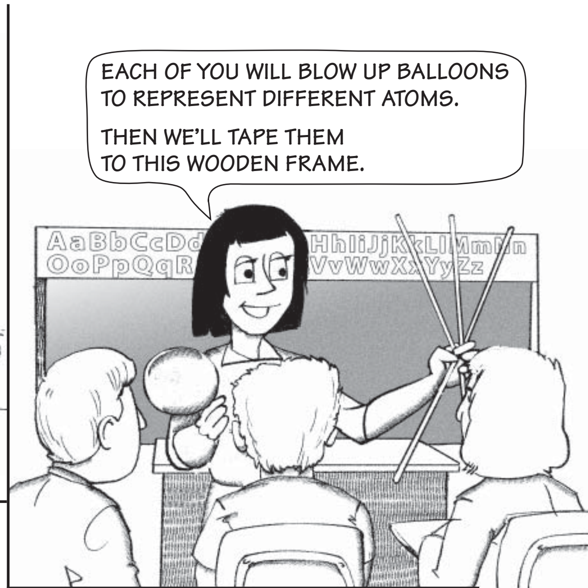
When multiple parts have to work together—anything from twine and birdcages to collections of complex information systems—and those parts are constructed independently, then there isn't a prayer of succeeding without rigorous and careful planning about how everything is supposed to fit together.

Just about everybody has a favorite story about the pitfalls of poor integration planning. And yet, over and over, during decades of software acquisition, project after project has made the same mistake. It is still being repeated today. With awful regularity, we see some group of people get together and someone says: "Hey! Let's build a Big Integrated System! I'll build the frammis and you build the jimjam. We'll get Bob to build a few claptraps!" But no one worries too much about the integration part of it. And, sooner or later, the integration turns out to be far more difficult than anyone had realized, and the twine breaks, the birdcage falls, the whole project smashes to the ground, and everyone else points fingers. Then, a few months later, a different group of enthusiastic, hopeful people gets together and someone says, "Hey! Let's go build a Really Big Integrated System..." and so forth.

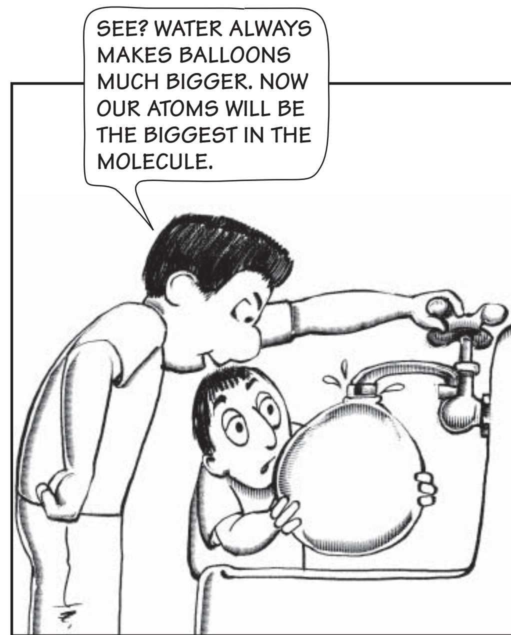
And that's the moment of truth, when somebody (perhaps you, Gentle Reader) has to pipe up and say "Hey, let's stop for a minute! Let's see if the plans for the frammis and the plans for the jimjam are consistent with each other. And let's be sure that Bob's claptraps will fit" or some comparable bit of caution. Because if *somebody* doesn't say something to that effect, and if that caution isn't shared by everyone in the room throughout the whole life of the project, then it's a virtual certainty that lots of people will work very hard for a while, but the frammis and the jimjam won't be compatible, and the claptraps won't fit at all. And, sooner or later, everyone will fall out of the tree yet one more time.

Bottom line: Interoperability doesn't happen just because you want it to. It takes effort and resources to make systems successfully interoperate in a useful way. So whenever someone asserts that "our systems will talk to each other..." or something like that, you might ask: "How much are we each budgeting for the interoperability aspect? Let's see that plan for how each of us will ensure we're keeping our side of the agreement. Hey, now that I think of it, let's see the agreement!" You might just find that the "agreement" is nothing more than a vague hope for a miracle.

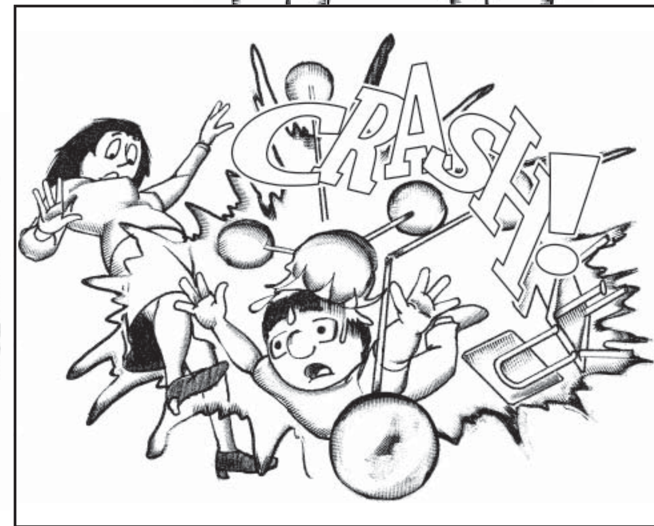

CLASS, TODAY WE'RE GOING TO BUILD A MODEL OF A MOLECULE.


EACH OF YOU WILL BLOW UP BALLOONS TO REPRESENT DIFFERENT ATOMS.
THEN WE'LL TAPE THEM TO THIS WOODEN FRAME.

GEE, RICKY, WE GOT THE SMALLEST BALLOONS. WE'LL LOOK LIKE DORKS.
DON'T WORRY, STICK, I HAVE AN IDEA.


SEE? WATER ALWAYS MAKES BALLOONS MUCH BIGGER. NOW OUR ATOMS WILL BE THE BIGGEST IN THE MOLECULE.

RICKY, IS THAT A WATER BALOON?
STOP! IT WILL BE TOO HEAVY...


CRASH!

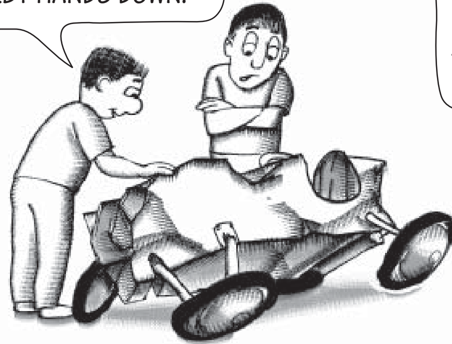
Looking at a single system gives a very different perspective from looking at several interconnected systems. What's optimal for the single system may not be so for the group, and vice-versa. The success of any collection of interoperating systems depends on just how these different perspectives are negotiated and resolved.

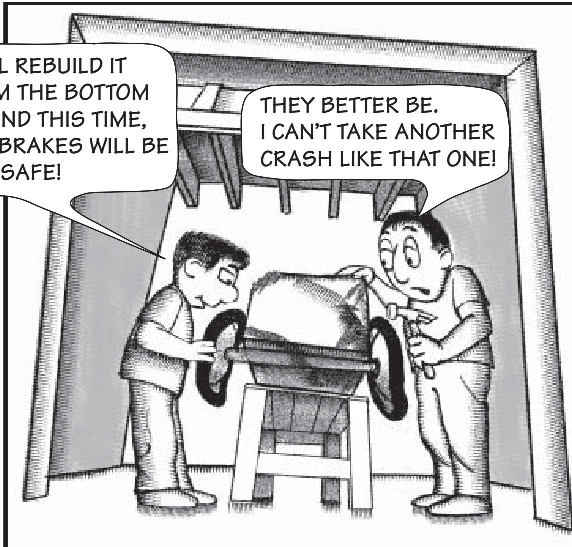
Ricky and Stick, for instance, saw no reason why they shouldn't make their dorky little balloons bigger. They looked better, and probably felt better. (And who *doesn't* like the feel of a good water balloon?) But neither of them considered that their balloons weren't independent, but were going to be in a collaborative relationship with a lot of other balloons.

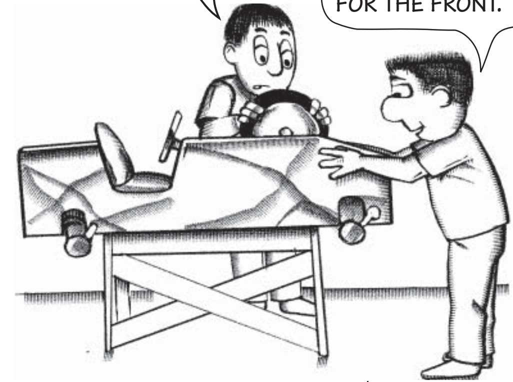
It's not that different for software managers. Software is *so* easy to tweak and change, and the owner of one system sometimes sees no reason why he shouldn't make just one *little* fix here or there, to make his own system a bit better. But when this happens, the change, however small, might disturb something about the agreements with other systems, and can potentially have a serious impact on the whole system of systems, perhaps even destroying it.

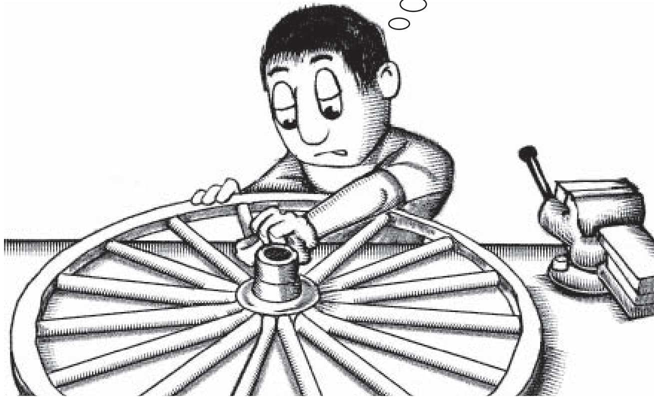
When systems are in relationships with other systems, the success of the whole depends on assumptions and agreements that each system adheres to; this is especially true for software systems. The agreements are sometimes specified, but not always. In fact, many of today's interoperating systems don't really have a clear agency that is responsible for the whole; instead they depend entirely on unwritten assumptions that everyone adheres to voluntarily. In Mrs. Perillo's case, there was certainly at least one unwritten assumption: she never expected that anyone would add balloons that were much too heavy, and thus saw no need to say "Don't use water balloons!"

The lesson is that if you're the manager of a system that's an element in a system of systems, you need to be proactive in preserving agreements, written and implicit. Before making any change, even a seemingly trival one, you might consider asking everyone (and that means everyone, those nearby and those light-years away) whether the change will affect their systems' operation. Otherwise, you might unintentionally change something that breaks the whole shebang. Then the system stops running, molecules fall down and everybody gets soaked.

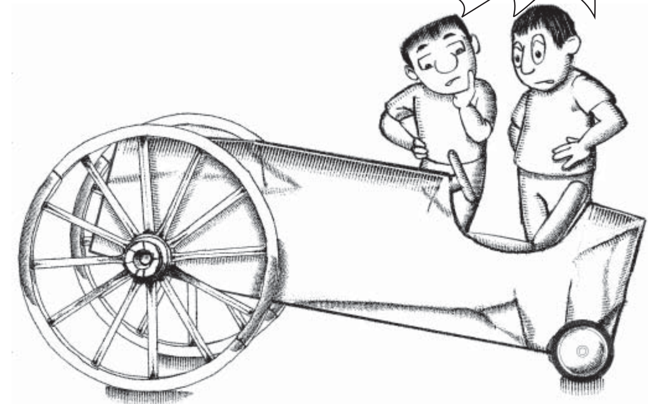

HEY, STICK, LET'S GET THIS RACER INTO TIPTOP CONDITION, AND WE'LL WIN THE NEXT SOAPBOX DERBY HANDS DOWN.


WE'LL REBUILD IT FROM THE BOTTOM UP. AND THIS TIME, THE BRAKES WILL BE FAIL-SAFE!
THEY BETTER BE. I CAN'T TAKE ANOTHER CRASH LIKE THAT ONE!


IT LOOKS A LOT BETTER. BUT WHAT ABOUT THESE OLD WHEELS, RICKY? THEY'RE ALL BENT OUT OF SHAPE.
WE NEED NEW WHEELS, STICK. I'LL GET TWO FOR THE REAR AND YOU WORK ON THE TWO FOR THE FRONT.


THESE BIG WHEELS WILL BE GREAT. WE'LL BE HIGHER OFF THE GROUND, SO WE'LL SEE MUCH BETTER.

WITH THESE BABIES, WE'LL BE LOW TO THE GROUND AND GO A LOT FASTER.


#?!#%! . . .

When a system is upgraded with new parts, it generally needs to be done with an overall understanding of the goal of the upgrade. But when upgrades to different pieces are done independently (as often happens with systems of systems, each of which may follow a separate evolutionary path), the upgrades can sometimes be at odds with each other.

For instance, separate upgrades can follow very different evolutionary goals. The upgrade to System A may aim toward greater internal effeciency while that of System B may aim at a better user interface. (Or, as in the case of our hapless heroes, Stick wanted to see better, Ricky wanted to make the racer faster.) Each upgrade might separately represent an improvement. But considered from the perspective of the whole, the aggregate system may not be improved at all; it might not even be operable. (Truth to tell, Ricky's racer, even with mismatched wheels, could still roll. But it would probably be slower, not faster, and the driver wouldn't see where he was going. While that wouldn't bother Ricky all that much, it's more serious when it describes how some actual systems evolve.)

And conflicting evolutionary goals are not confined to huge systems; they can pop up in small, isolated systems just as easily, and they can occur whether you're dealing with COTS products or custom-written code.

Bottom line: The evolution of *any* separate part has to be done with an awareness of how that evolution affects the integration of the whole. So if you (or your contractor) are contemplating an upgrade to a system, you might aim to explicitly answer such questions as: What is the goal of this upgrade? How does it match with upgrades to other systems with which our system interoperates? Because if multiple evolutionary goals are at odds, the integrated working of the whole might well be destroyed.

including contracted carriers, Military Service and Defense Logistics Agency (DLA) integrated materiel managers (IMMs), weapon system program offices, commercial distributors and suppliers including manufacturers, commercial and organic maintenance facilities, and other logistics activities (e.g., engineering support activities (ESAs), testing facilities, cataloging services, reutilization and marketing offices).

C1.2.2. This Regulation presents DoD logistics personnel with a process-based view of materiel management policy within a supply chain framework. This structure underscores the fundamental changes and collaborative initiatives that are occurring to meet warfighter sustainment needs and the operational requirements of the National Military Strategy.

C1.2.3. Those needs and requirements required that the DoD Components provide supplies and services that support:

C1.2.3.1. Rapid power projection;

C1.2.3.2. Improved readiness through performance-based logistics; and

C1.2.3.3. World-class standards for customer responsiveness. The guidance in this Regulation encourages the DoD Components to:

C1.2.3.3.1. Transform their support of weapons systems through total life-cycle management, increased partnering, and adoption of modern information technologies.

C1.2.3.3.2. Establish end-to-end processes that are focused on maximizing customer service or warfighter support.

C1.2.3.3.3. Implement contemporary business systems and practices that enable the integration of people, information, and processes.

C1.3. DoD SUPPLY CHAIN MATERIEL MANAGEMENT GOALS

C1.3.1. Policy from Directive. According to DoD Directive 4140.1 (reference (a)), all DoD Components shall:

C1.3.1.1. Structure their materiel management to provide responsive, consistent, and reliable support to the war fighter during peacetime and war. That support should be dictated by performance agreements with customers to the furthest extent. For weapon system materiel, those agreements should be negotiated with

WE'LL WORRY ABOUT THAT WHEN THE TIME COMES.

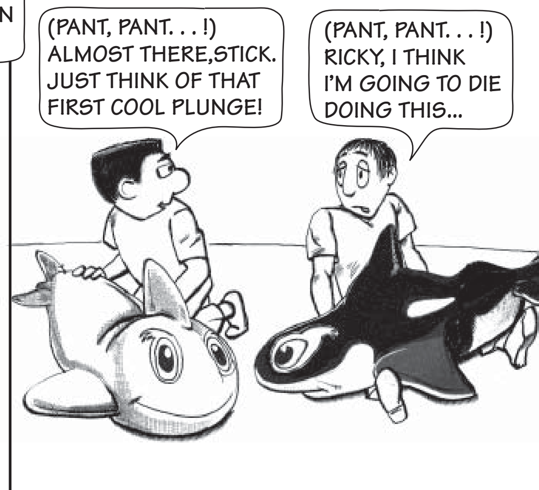
Deployment

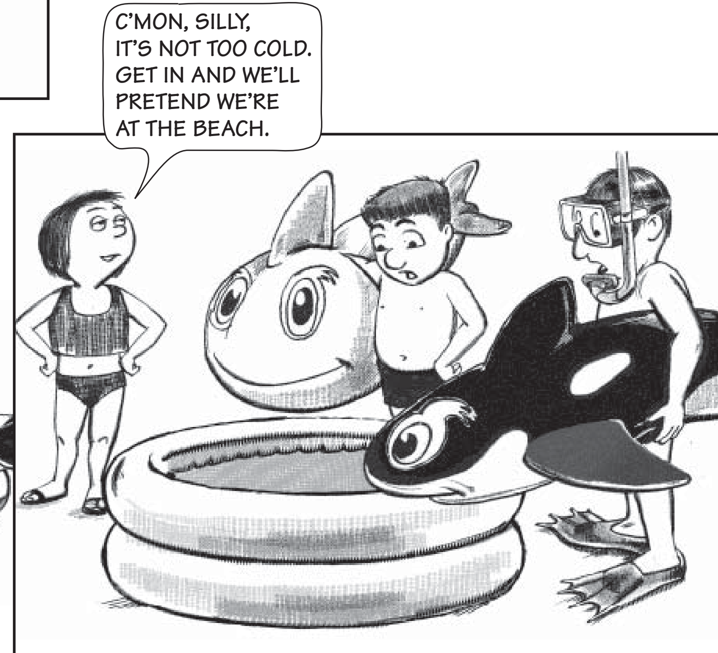
GOSH, STICK,
I'M DYING IN
THIS HEAT!
ME TOO, RICKY.
I WISH WE WERE
AT THE BEACH!

HEY, YOU TWO -- WHY DON'T
YOU COME OVER AND PLAY
IN MY NEW POOL!
THANKS,
GLORIA!
THAT WILL
BE GREAT!

STICK, LET'S GET GOING. WE'LL
BLOW UP THE FLOATS WE GOT
AT THE BEACH LAST SUMMER!
I CAN'T WAIT
TO DO MY FIRST
CANNONBALL

HURRY, STICK,
WE HAVEN'T
GOT ALL DAY!!
GEE, RICK, THIS
IS HARDER THAN
I THOUGHT.


MUCH LATER . . .
(PANT, PANT. . . !)
ALMOST THERE,STICK.
JUST THINK OF THAT
FIRST COOL PLUNGE!
(PANT, PANT. . . !)
RICKY, I THINK
I'M GOING TO DIE
DOING THIS...


C'MON, SILLY,
IT'S NOT TOO COLD.
GET IN AND WE'LL
PRETEND WE'RE
AT THE BEACH.

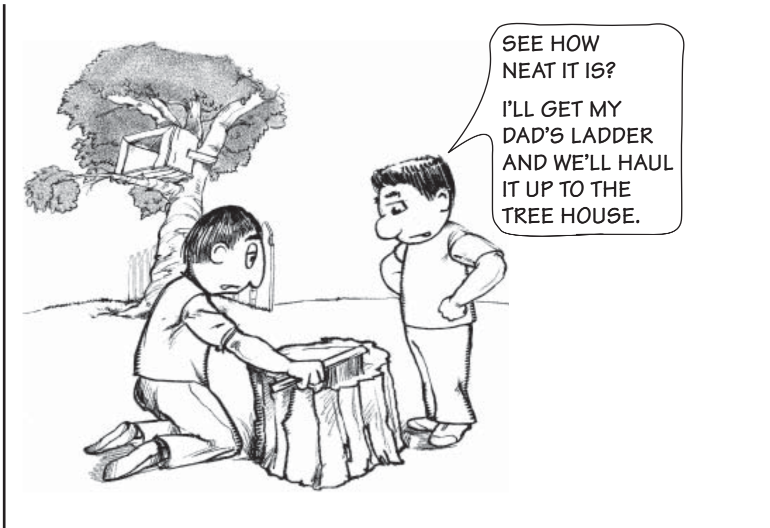
Chronologically speaking, deployment of a system comes late in the life cycle. But knowledge about where the system will actually be installed and run is needed *way* upfront, when the requirements are being decided.

And it's painful to observe how often this kind of experience really occurs—how often everyone concentrates only on the system and forgets to think ahead about deployment. And when that happens, it's not all that different from poor Ricky and Stick, who worked so hard blowing up their huge floats, not realizing that the vast Olympic pool they expected to find was nothing like the dinky wading pool they eventually found.

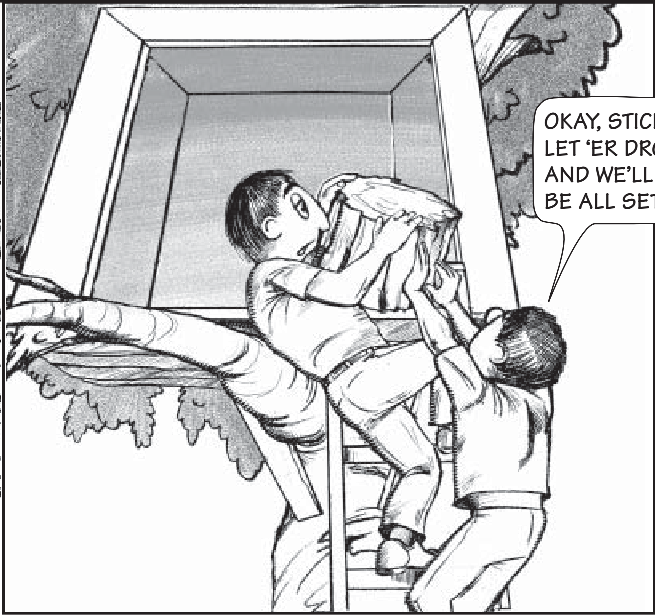
Bottom line: Find out early as many grisly deployment details as you can. Get explicit answers to such questions as: "Where is the system going to be deployed? What is the physical location? What hardware will it run on? What else will share the operating environment?" Such knowledge is no less critical than any of the other requirements, and getting this knowledge early can only help.

HEY, STICK WE NEED A TABLE UP HERE.
WE DO?

SURE WE DO.
LOOK, STICK, THAT OLD STUMP WILL BE GREAT!


SEE HOW NEAT IT IS?
I'LL GET MY DAD'S LADDER AND WE'LL HAUL IT UP TO THE TREE HOUSE.

ALMOST THERE, STICK.
WHEN WE GET IT TO THE TOP STEP, WE CAN JUST LET IT DROP INTO THE TREE HOUSE!


OKAY, STICK, LET 'ER DROP AND WE'LL BE ALL SET!

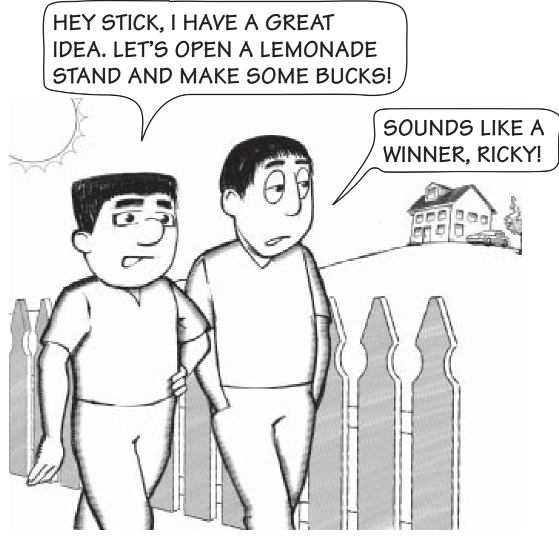
SNAP!!
CRAAAK!!
CRASSH!!

It's all too easy to focus only on the benefits you'll get from a new system: its hoped-for functionality, the ROI it will bring, or whatever other great things were the selling points that got the program approved in the first place.

And this can mean that you ignore thinking about context, and about whether the deployment environment is capable of supporting the new system (in much the same way as whether a flimsy tree house can support the weight of a very heavy stump). If it can't, then you may find yourself expending a huge effort getting the system into place, as did our heroes, only to come to grief.

Nor is this necessarily a hardware issue; the moral is no less applicable (in fact is very applicable!) to a large, complex software system. Lots of questions are apt: What *additional* software resources are needed for system deployment? Who has the responsibility to supply them? How much will deployment *cost*? Where are those dollars in the budget? Will it deploy in stages? and so forth.

The moral is that you and your contrator need to know explicit details about the deployment environment—load factors for instance—and then be sure that the system will operate properly in that environment. And you need to know it way upfront: though the deployment *process* may be far in the future, deployment *planning* should be done at the earliest part of the project.

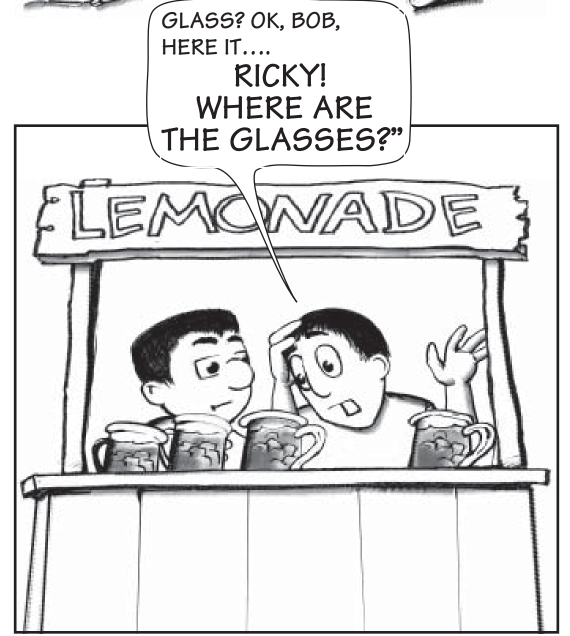

HEY STICK, I HAVE A GREAT IDEA. LET'S OPEN A LEMONADE STAND AND MAKE SOME BUCKS!
SOUNDS LIKE A WINNER, RICKY!

OK, STICK, LET'S GET BACK TO THE KITCHEN AND BRING EVERYTHING OUT. WE'LL BE RICH IN A COUPLE OF HOURS!
LEMONADE

RICKY, WHY DID WE BUILD THE STAND SO FAR AWAY FROM THE KITCHEN?
FORGET IT STICK, WE'RE READY TO GO.

RICKY, WE FORGOT THE ICE!
DARN, STICK. OK, LET'S GO GET IT.

(PANT, PANT) OK, STICK, WE'RE FINALLY SET TO MAKE THE BIG BUCKS.
HI, GUYS, I'LL TAKE A GLASS!
JUST IN TIME– HERE COMES OUR FIRST CUSTOMER.


GLASS? OK, BOB, HERE IT…. RICKY! WHERE ARE THE GLASSES?"
LEMONADE

There are dozens of stories about glitches in deploying software systems: everything from insufficient memory or too-slow hardware to incompatible disk drives and the glare from fluorescent light bulbs. And there really have been such errors.

Some of these glitches are truly difficult to see in advance, at least until you've been burned once or twice. There are, for instance, some thorny logistical issues, things like the length of supply chains, or the time needed to replenish needed items. You may think, for instance, that selling lemonade is your real job; but you can't sell it without glasses and ice. And for Ricky and Stick, their task was made significantly more difficult because of how far they had to run back to get those glasses and ice, while the lemonade sat in the sun and poor Bob stayed thirsty.

As with almost any story about deployment, the culprit is focus, since we all tend to focus on the system being built, and on its requirements, its features, its design. In so doing, it's all too easy to neglect many things that are inherently boring to most software engineers —a lot of things that software depends on are not really software things. But *someone* has to worry whether the extension cord is long enough, and *someone* needs to think about whether the chairs are too small.

Bottom line: No matter how spiffy the software is, if the people in the field aren't able to use it, it does them no good. What else is necessary? Have the users been given all the additional tools they need? Have they been trained properly? and other similar questions. As with poor Bob, there may be some great-looking lemonade right in front of them, but they'll be thirsty until the glasses arrive.

E7. ENCLOSURE 7

HUMAN SYSTEMS INTEGRATION (HSI)

E7.1. GENERAL

The PM shall have a comprehensive plan for HSI in place early in the acquisition process to optimize total system performance, minimize total ownership costs, and ensure that the system is built to accommodate the characteristics of the user population that will operate, maintain, and support the system. HSI planning shall be summarized in the acquisition strategy and address the following:

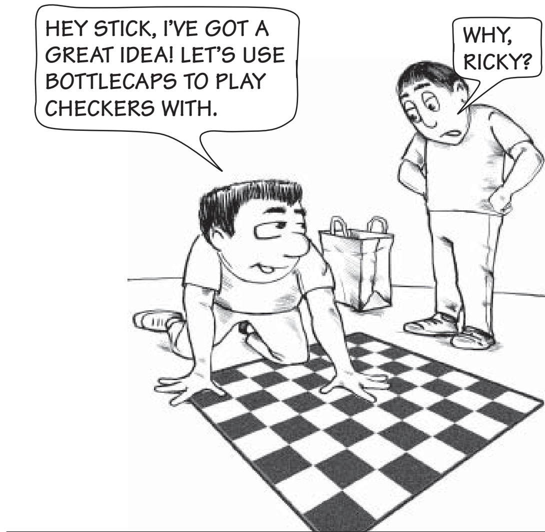
E7.1.1. Human Factors Engineering. The PM shall take steps (e.g., contract deliverables and Government/contractor IPT teams) to ensure human factors engineering/cognitive engineering is employed during systems engineering over the life of the program to provide for effective human-machine interfaces and to meet HSI requirements. Where practicable and cost effective, system designs shall minimize or eliminate system characteristics that require excessive cognitive, physical, or sensory skills; entail extensive training or workload-intensive tasks; result in mission-critical errors; or produce safety or health hazards.

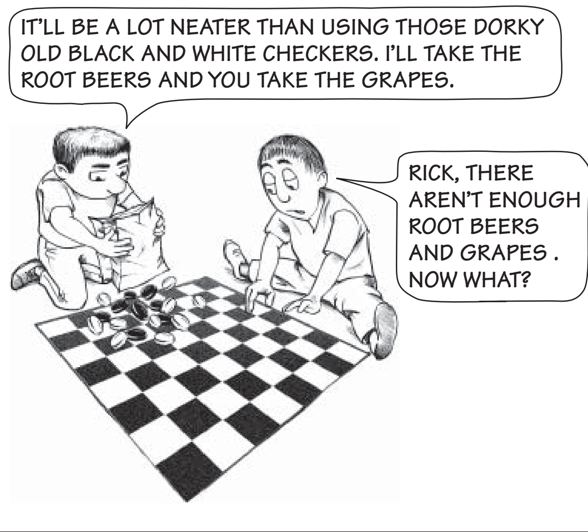
E7.1.2. Personnel. The PM shall work with the personnel community to define the human performance characteristics of the user population based on the system description, projected characteristics of target occupational specialties, and recruitment and retention trends. To the extent possible, systems shall not require special cognitive, physical, or sensory skills beyond that found in the specified user population. For those programs that require skill requirements that exceed the knowledge, skills, and abilities of current military occupational specialties or that require additional skill indicators or hard-to-fill military occupational specialties, the PM shall consult with personnel communities to identify readiness, personnel tempo (PERSTEMPO), and funding issues that impact program execution.

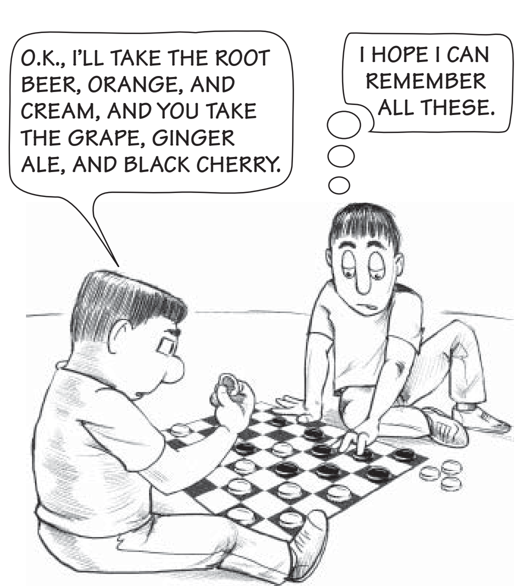
E7.1.3. Habitability. The PM shall work with habitability representatives to establish requirements for the physical environment (e.g., adequate space and temperature control) and, if appropriate, requirements for personnel services (e.g., medical and mess) and living conditions (e.g., berthing and personal hygiene) for conditions that have a direct impact on meeting or sustaining system performance or that have such an adverse impact on quality of life and morale that recruitment or retention is degraded.

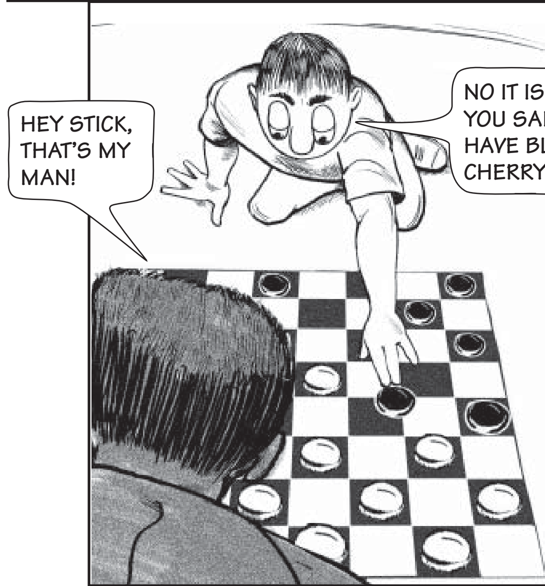
YOU DO IT YOUR WAY, AND I'LL DO IT MINE . . .

Business Processes


HEY STICK, I'VE GOT A GREAT IDEA! LET'S USE BOTTLECAPS TO PLAY CHECKERS WITH.
WHY, RICKY?


IT'LL BE A LOT NEATER THAN USING THOSE DORKY OLD BLACK AND WHITE CHECKERS. I'LL TAKE THE ROOT BEERS AND YOU TAKE THE GRAPES.
RICK, THERE AREN'T ENOUGH ROOT BEERS AND GRAPES . NOW WHAT?


O.K., I'LL TAKE THE ROOT BEER, ORANGE, AND CREAM, AND YOU TAKE THE GRAPE, GINGER ALE, AND BLACK CHERRY.
I HOPE I CAN REMEMBER ALL THESE.


HEY STICK, THAT'S MY MAN!
NO IT ISN'T! YOU SAID I HAVE BLACK CHERRY!

THIS ISN'T BLACK CHERRY — IT'S COLA. THIS ONE SHOULD BE MY MAN.

YOU NEVER SAID ANYTHING ABOUT COLA! WHY SHOULDN'T HE BE MY MAN?
I HATE THIS GAME!

It's usual that the introduction of a new software system means that the users will be doing something different from their familiar tasks. As often as not, the business processes have been reengineered and improved, and the end result is that things will change for the better (or, at least, so everyone hopes). But every now and then, someone decides to introduce a software system that doesn't really change any business processes. All it does is force the users to learn some very annoying software steps that don't improve anything: the users are doing exactly what they used to do…but now it's harder.

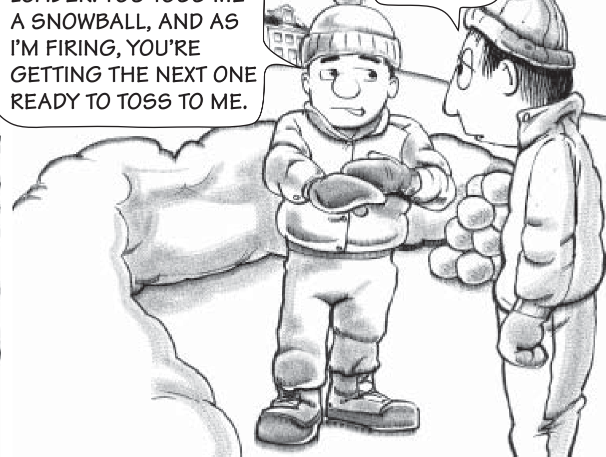
When Ricky decided to improvise a new set of checkers, he paid no heed to what that implied. He and Stick had played checkers a million times, and were still playing the same game this time around; same rules, same strategy, same everything. But now they were fighting their own tools; they couldn't even tell which were their own men, and they ended up fighting with each other. The same thing can happen with software: a simple task executed with pencil and paper can become agony when it needs three screens, forty-three keystrokes, and a trip to the printer.

There are, on occasion, good reasons to introduce new software while keeping some process unchanged. It might be a huge increase in transaction turnaround time, or a vital need for consistency with other operations, or something of that kind. But whatever the reason, it has to be enough to offset the inevitable unhappiness of the end users. More to the point, any conceptual separation between a process and the software that implements it is suspect.

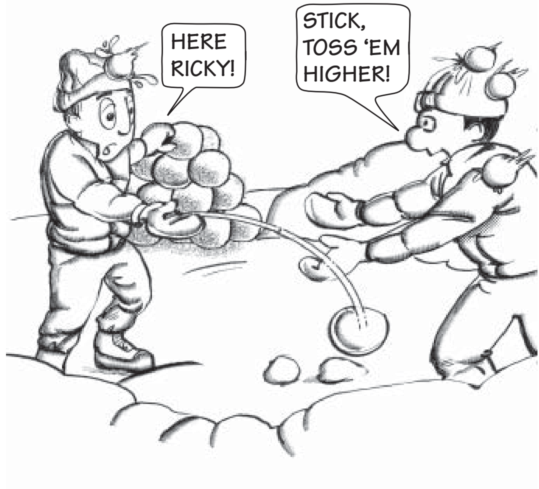
So the moral is: If new software comes into play, it's probably bogus to think that all it's doing is supporting the same old process.

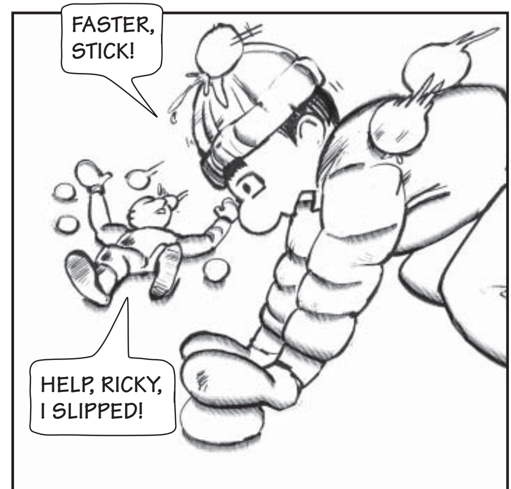
Said in reverse, if you truly must introduce some new piece of software, ask yourself: Have I considered what process reengineering is needed? Have I brought it about? If you haven't, it's worth taking a hard look to find out why.

STICK, I'VE DEVELOPED A SURE-FIRE IMPROVEMENT IN SNOWBALL WARFARE! WALLY WILL BE DEAD MEAT!


I'LL BE THE GUNNER, AND YOU'LL BE THE LOADER. YOU TOSS ME A SNOWBALL, AND AS I'M FIRING, YOU'RE GETTING THE NEXT ONE READY TO TOSS TO ME.
OK, RICKY, I HOPE IT WORKS!

OK, WALLY, PREPARE TO MEET THY DOOM! THIS IS NEXT GENERATION SNOWBALL TECHNOLOGY!
GO AHEAD, DUMBBELL. I CAN HARDLY WAIT!


HERE RICKY!
STICK, TOSS 'EM HIGHER!


FASTER, STICK!
HELP, RICKY, I SLIPPED!

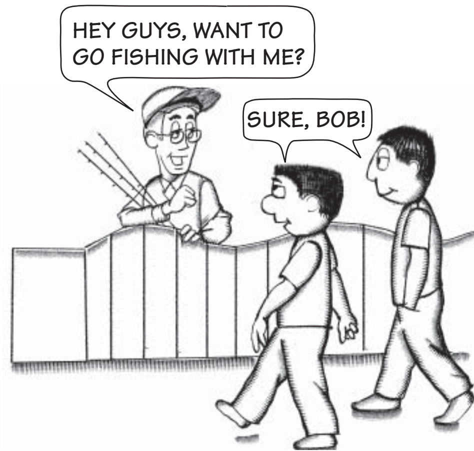
BOY, ARE YOU TWO DUMB!
SPLAAT!

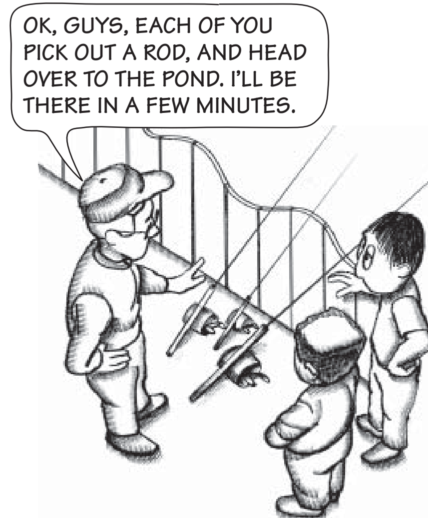
When a new software system is introduced, often with a loud public flourish, it sometimes happens that it falls flat on its face. Usually it's just embarrassing, but sometimes it's quite dangerous, with the potential for grave effect. This has led to a widespread belief that a large percentage of *all* software is seriously flawed, and that the craft of creating computer programs is unacceptably primitive.

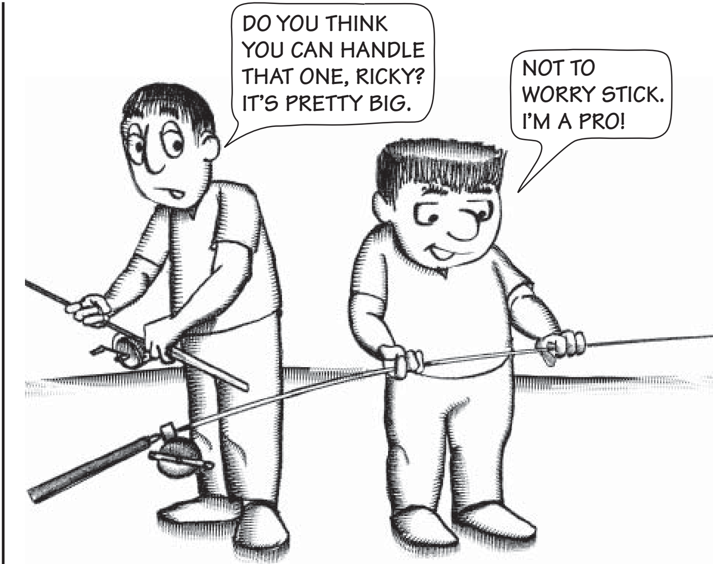
Some of the belief is justified; there's a lot of bad software out there. But an equally guilty partner, one that usually hides far from public scrutiny, may be that the folks responsible for introducing the new system didn't pay any *real* attention to the training of the new system's users. Those users likely needed significant practice with the changed and reengineered business processes the new system demands, and with that training, the system might otherwise have been a triumphant success.

Now, Ricky may have been on to something with his new mode of snowball warfare. But he and Stick didn't bother to practice it, so they were total doofuses to try out such a radically different system when they were in mortal combat with Wally. (Who knows - the whole future of the Ricky-Wally War might have been different.) For any manager whose responsibility involves bringing a *system* to IOC, the task of bringing its *users* to IOC is of equal importance.

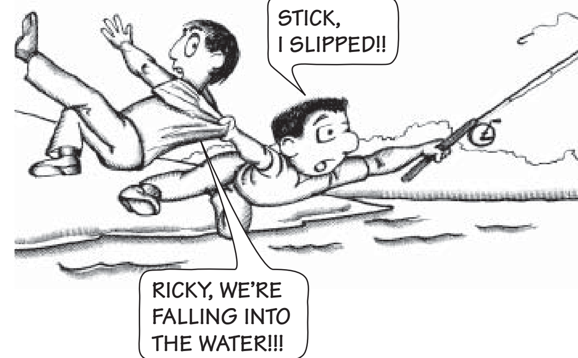
Moral: it's worth looking carefully at how training appears in the project plans. If the training appears to be an afterthought, it's probably not enough. If the training is being squeezed by the schedule, the schedule needs to be changed. And if the training isn't even on the radar screen, then you'd be wise to buy a bus ticket and get out of town. Wally's got some nasty-looking snowballs, and he's right behind you.


HEY GUYS, WANT TO GO FISHING WITH ME?
SURE, BOB!


OK, GUYS, EACH OF YOU PICK OUT A ROD, AND HEAD OVER TO THE POND. I'LL BE THERE IN A FEW MINUTES.


DO YOU THINK YOU CAN HANDLE THAT ONE, RICKY? IT'S PRETTY BIG.
NOT TO WORRY STICK. I'M A PRO!

C'MON STICK, LET'S GET STARTED. WATCH THIS!
SHOULDN'T WE WAIT FOR BOB?


STICK, I SLIPPED!!
RICKY, WE'RE FALLING INTO THE WATER!!!

LISTEN, RICKY: DON'T PICK OUT A ROD JUST BECAUSE IT'S THE BIGGEST ONE. YOU WANT A ROD THAT SUITS YOU!

Tools, whether fishing rods or software systems, should be appropriate to both their intended use and their intended users. But often, there's a mismatch somewhere along the line. Sometimes, what the users want isn't what they really need. And sometimes, regardless of what they want, what they need is far from what they get.

Ricky's tumble into the drink came from this kind of mismatch. He was dazzled by *bigger!*, and paid no attention to the fact that the huge rod was far beyond his size and strength. Sometimes, organizations are equally dazzled by other things—*newer! fancier! better! cheaper! faster!*—all of which are equally seductive and equally dangerous.

No one can doubt the DoD's need for the finest software systems possible, a need that will continue for the foreseeable future. But resources are finite, and they have been squandered too often, usually because realism somehow gets misplaced, just as happened to Ricky. So questions like the following are apt, and should be asked as early as possible: Are *these* the capabilities that we really need? Or is our true need somewhere else? What *precisely* will happen if we don't get this new system? If the acquisition is complex, or expensive, or controversial, does the system's potential benefit outweigh the risks should the acquisition fail? Is it imperative to take a large leap forward, or can there be several small steps? And are we trying to use a fishing rod that we have no business using?

When posing these questions, you will run the danger of "acting negative," or "being obstructionist," or "not thinking out of the box," or some equally vacuous accusation. Keep the faith, friend. In response, you can point to a depressingly large number of failed software programs over the past two decades. Surely, totaling the cost of those wasted programs should be answer enough.

ONE FINAL THOUGHT . . .

ONE DAY, AS SUMMER WAS COMING TO AN END, RICKY TOLD HIS MOM HOW MUCH HE WAS DREADING THE START OF SCHOOL. "I HATE THOSE DUMB TESTS, AND THOSE DUMB ASSIGNMENTS. WHEN THE SUMMER STARTED, I THOUGHT I WAS FINALLY FREE, AND NOW IT'S ALL STARTING AGAIN!" HE WAS NEAR TEARS.

HIS MOM WAS UNDERSTANDING, BUT REMINDED RICKY, "SURE, HONEY. SUMMER WAS A FUN TIME. BUT DON'T FORGET, SUMMERS ARE ALWAYS TOO SHORT, AND AUTUMN ALWAYS COMES, AND THEN YOU ALWAYS GO BACK TO SCHOOL. AND SCHOOL ALWAYS MEANS YOU'LL HAVE NEW TEACHERS, WITH NEW THINGS TO LEARN, AND TESTS AND ASSIGNMENTS FOR YOU TO DO."

THIS WAS THE FIRST TIME THAT RICKY HAD EVER CONSCIOUSLY PAID ATTENTION TO THE IDEA OF PERPETUALLY GOING INTO NEW GRADES. HE KNEW, OF COURSE, THAT THERE WERE OLDER KIDS IN HIGHER GRADES. BUT HE HAD NEVER QUITE THOUGHT ABOUT SCHOOL IN THIS NEVER-ENDING WAY.

SUDDENLY, HE GOT A HORRIBLE VISION OF THE REST OF HIS LIFE FILLED WITH NEW SCHOOL YEARS, HATEFUL ASSIGNMENTS AND TESTS, AND TEACHERS NAGGING HIM TO LEARN EVER-HARDER SUBJECTS. HE WAS BEGINNING TO UNDERSTAND THAT A NEW TERM WOULD NEVER BE A ONCE-ONLY EVENT TO BE GOTTEN THROUGH, BUT PART OF A LARGER, ONGOING PROCESS. THE PROSPECT FILLED HIM WITH A GREAT DREAD.

AFTER RICKY HAD GONE TO BED (UNUSUALLY, BEFORE BEING TOLD TO), HIS DAD ASKED HIS MOM, "WHAT'S WRONG WITH HIM?" SHE SMILED. "DON'T WORRY," HIS MOM SAID, "HE'S JUST BEGUN TO PUT IT ALL TOGETHER — REALIZING THAT HE'LL ALWAYS BE MOVING INTO ANOTHER GRADE, WITH UNFAMILIAR TEACHERS, AND NEW SUBJECTS. HE DOESN'T WANT IT TO BE TRUE, POOR KID; HE WANTS TIME TO STOP AND FOR THINGS TO STAY JUST THE WAY THEY ARE NOW. BUT HE'S STARTING TO UNDERSTAND THAT HE DOESN'T HAVE MUCH CHOICE IN THE MATTER…"

Owners of modern software systems, particularly information systems, are increasingly aware that the stability of their systems is constantly being undermined. Familiar tools disappear, new ones appear, Web services evolve, commercial products are updated, users demand new capabilities, and so forth. And the pace of this instability is only increasing. It's normal to find this unsettling, and to try to nail down islands of stability that last while the rest of the world changes around you. But that strategy will only work for a little while—about as long as the endless summer that Ricky thought he had finally found.

The real solution, difficult as it may be, is to embrace the march of technology and to make it work for you; to accept that change really will be never-ending. This means developing a strategy that somehow encompasses and expects the inevitable earthquakes to your system, and makes them opportunities for improvement and growth.

Epilogue

And so we come to the end of the adventures of Ricky and Stick. We began by calling these stories "fables," and the term is apt, since each story is intended to demonstrate some moral or useful principle. Thus, in each episode, we've suggested a few ways that the story might be applicable to problems in the real (and, sadly, no less hazardous!) world of software acquisition. We truly hope that, in addition to provoking a smile or two, some of these tales will resonate with readers who are really grappling with the situations that are only fictional here.

In the great fables of antiquity, their morals were generally stated as clear, easily-remembered mottoes, such as "One man's pain is another man's pleasure," or "Necessity is the mother of invention." Unfortunately—from this author's viewpoint at least—these well-turned phrases are remarkably difficult to come up with. (I suspect that old Aesop may have had a couple of Madison Ave. types helping him out.) But since I wish to place the overall set of "morals" in some sort of relief, I herewith append a thumbnail list of them.

That the morals are largely self-evident is obvious. That they often require restating is painfully true.

So to recap, the Morals Of Our Stories are:

It's really good to test before fielding. (p.9)

It's usually wiser to let a model model something real (p.11)

IV&V is, by and large, a good thing. (p.13)

Using the "seems OK to me" rule is often a recipe for disaster. (p.17)

Metrics aren't second-class citizens. (p.19)

Counting the right things is better than counting the wrong things. (p.21)

It can't hurt to rethink the requirements every now and then. (p.25)

A grab bag of "want-to-have's" doesn't make a requirements set. (p.27)

"Nailing down the requirements early" isn't necessarily a good idea. (p.29)

Interoperation doesn't happen just because everyone wants it to. (p.33)

Interoperation fails if someone ignores the assumptions held by everyone else. (p.35)

If separate systems evolve, somebody needs to keep an eye on preserving their interoperation. (p.37)

Knowing where a system will be deployed is as important as knowing what the system does. (p.41)

Planning for deployment generally needs to come as early in the life cycle as other system requirements. (p.43)

Deploying the software usually means deploying a lot more than the software. (p.45)

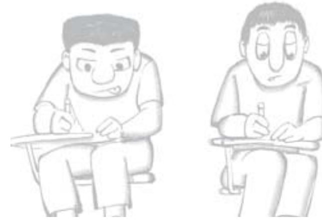
Big software change almost always means big change to the business process. (p.49)

Any new business process means some—and often a lot—of new training. (p.51)

The biggest (or most expensive, or most feature-laden) system is not necessarily what is needed. (p.53)

Continunual change is inevitable. This is true whether we wish it or not. (p.55)

Afterword



Writing this little book has been a genuinely pleasurable experience. David and I found the collaboration enlightening on several levels, not least of which was how well the little comic stories we devised had genuine bearing on topics that are really quite serious and important. And we feel that there's a lot more that could be said. We're therefore very interested in hearing from any readers who can suggest comparable situations, topics, stories, scenarios, whatever. If you can help us concoct a few more of these little fables, we'll definitely find a way to make them see the light of day. (djc@sei.cmu.edu)

Acknowledgements

A great many friends made significant contributions to this endeavor. Lisa Masciantonio and Al Evans found ways to bridge the gap between having a lot of fun and working on a real project. Tricia O., Fast Eddie, Denny D., and Big Bad John were particularly helpful and gave welcome encouragement. Claire D. corrected sloppy grammar and removed illogical and inconsistent wording. Slim Hissam kept me on the straight and narrow all through the early days, when I nearly fell into several traps; it's really due to him that Ricky and Stick came alive. And more than any words of thanks can convey, Bob Fantazier came up with unbelievable solutions to convert a large number of text files, drawings, designs, random ideas, and last-minute demands into a hugely attractive book. To all of these friends, David and I would like to say "Hey Stick, you know what? We should write a book…"

For More Information, contact—

David J. Carney

Phone
412.268.6525

Email
djc@sei.cmu.edu