ESD/MITRE
Software Acquisition
SYMPOSIUM

An ESD/Industry Dialogue

Proceedings
May 6-7, 1986
Currently, software cost estimation models are at best accurate to within 20 percent of the actual real costs about 70 percent of the time. One of the reasons they are not any better is that the data to which they are calibrated isn't any better. Furthermore, I think the situation is going to get worse before it gets better because of the uncertain effect of such new technologies as rapid prototyping, Very High Level Languages, and Ada.

We need a consistent set of counting rules for Ada programs. Lines of Ada code are put through a "pretty printer" and suddenly you have a program that is twice as large. We need to define and collect data very carefully. One exception to all this pessimism is the ESD Software Management Metrics, which I think are going to lead to some very valuable data in the process of helping ESD manage its software projects.

We need better sizing primitives. Again, we all use lines of code because we haven't found anything better. We have tried function points; they work pretty well on small-to-medium business applications, but they do not work very well on real-time or people-intensive projects.

One of the things that I think was good about the STARS Business Practices Workshop was that it recommended that the Department of Defense (DOD) not standardize on a single cost model. That would freeze technology and it would reduce people's options to calibrate and to get a little bit of parallel triangulation on the problem by using more than one way of costing.

The cost models that are currently around are hard to adapt to new technology. Some of them are pretty good at addressing reusable components. Some of them are just barely getting data that provides some idea of what Ada will do to software costs. Several models are doing pretty well on incremental development. Hardly any of them do very well at costing the impact of using fourth-generation languages or prototyping, either on the cost or the schedule involved in a software development project.

I think the most important thing you can do to get a better cost estimate is to better define the software product that you are going to build. A couple of years ago I ran an experiment where seven teams of people built essentially the same product with the same requirements. At the end of this process, the seven teams produced software in the same elapsed time, but the number of man-hours it took varied by a factor of three. The same inputs and the same outputs were required. One of the requirements was to build a user-friendly interactive interface and a single-user file system. The way people interpreted those requirements really gave you the factor of three in how expensive this was.

Figure 1 is based on a curve from my book, "Software Engineering Economics," with elaboration based on data collected since the book was published. As you proceed through the life cycle and define a concept of operation, a requirements specification, and a design specification, you reduce the variance in the cost estimates. If you are estimating before you have a concept of operation, you haven't pinned down the classes of people and data sources you are supporting, and it shouldn't be surprising that your cost estimates may be off by a factor of two to four in either direction. As you get a concept of operation
EXAMPLE SOURCES OF UNCERTAINTY, MAN-MACHINE INTERFACE SOFTWARE:

- Classes of people, data sources to support
- Query types, data loads, smart-terminal trade-offs, response times
- Internal data structure, buffer-handling techniques
- Detailed scheduling algorithms, error handling
- Programmer understanding of specification

PHASES:
- Feasibility
- Plans and Requirements
- Product Design
- Detail Design
- Development and Test

Figure 1: Seven Experimental Programs with the Same Requirements

pinned down, the variance reduces, but at the time of ESD proposals, many other factors that you haven’t pinned down still give you a wide source of variation.

In the same experiment, I counted the number of lines of code in each one of these products and found a similar variation: one program had 1,300 instructions, another had 4,600 instructions, and the others were spread in between.

There was one interesting difference that accounted for a lot of the variation in the size and cost. Basically, the seven teams all built programs for COCOMO cost estimation models, and at the beginning of this activity everybody had been furnished the same inputs and outputs, equations, and variables. Some of the other things were very broadly specified: each team was to produce a “user-friendly interface” and a “single-user file system.” However, they were all equivalent in that they required user’s manuals, maintenance manuals, and well-commented code. They all worked in the same environments.

Four of the teams used a specifying approach. They wrote a requirement specification. They
wrote a design specification, and then they wrote the code. Three of the teams used a prototyping approach. They built a prototype, exercised it, and went on from there to develop the code. Uniformly, the products that went through the requirements and design specifications before writing code had more instructions and were more expensive in the number of man-hours than the products that were developed using prototyping. The average was about 40 percent less code and 40 percent fewer man-hours for the products that were prototyped.

At the end of the experiment, we had three people exercise these programs and rate them on a scale of 1 to 10 with respect to functionality, robustness, ease of use, and ease of learning. The prototyped products were about one point lower in functionality and robustness and one point higher in ease of use and ease of learning. In general, they had less of what people called “gold plating.”

I asked the people involved to critique their experience, and one of the specifiers said the big problem with the specifying approach is that “words are cheap.” Basically, in doing reviews I would give each team the same kind of feedback. I would say some users would like to put the inputs in backwards as well as forwards. The specifiers would basically view that as one more sentence in the specification, and would put it in. The prototypers, on the other hand, had more of a feel for how expensive that would be because they understood how much breakage there was and how much redesign was involved, so they were much more reluctant to add to the specification.

In using the document-driven DOD-STD-2167 approach to writing down the specifications before you think about the implications, users tend similarly to request all possible options as part of requirements. All of these things get embedded into the requirements and locked into the contract.

On the other hand, prototyping wasn’t the universal winner. The specified products had such nice things as interface specifications. The prototypers basically started by dividing responsibility for the software, and four weeks later they would find they were building exactly the same pieces of software that did the error checking on the original and the modified inputs, and they were doing it in incompatible ways. They had specified the same variables with different names and different structures and had trouble integrating them.

From these experiments and similar experiences we have had at TRW, I came to the conclusion that a combination of prototyping and specifying is the best approach, and the best way to determine the mix is by using risk considerations.

Frequently, on both sides of the acquisition activity, we are involved in a situation where RFPs come out at a point in Figure 1 where there is still a factor of more than two for potential variation in the size and cost of the software. One of the things that I think forces us into adversarial relationships is that we lock on to early cost estimates. During the early planning phase of the life cycle, somebody picks a cost and an Initial Operating Capability (IOC) delivery date and your delivery budgets and schedules are fixed for all time. Then we compound the adversarial relationship by coming up with a fixed price contract at a point in time when we really have no idea of what the product is that we are building or what it will cost.

In a situation where our budgets or schedules are unavoidably fixed, I think it’s much better to design to cost. There have been a number of procurements that did this in a planned way. The one that I remember the best was the TIPS acquisition at SAMTEC where the requirements specification had a letter in parentheses after each itemized requirement indicating whether the requirement was mandatory or optional, and the bidders could do a proposed design to cost and knew what the priorities of the contract were.
Another problem that I think gets us into lose-lose situations are early Best And Final Offers. What does a Best And Final Offer mean if some of the requirements say “user-friendly interface” or “graceful degradation” or “99 percent reliability” where reliability isn’t defined? In reality, they are absolutely baseless and lock us into adversarial situations later on.

The best way to consider the cost variance relationship is that it represents a source of program risk, and the best way to address it is to come up with a risk management plan that addresses the major sources of variation. If we want a user-friendly interface, we can plan to do some prototyping that minimizes the risk. For other sources of risk, we can plan to do more mission modeling analysis, performance modeling analysis, or break the job up into increments where we may better understand what is in increment one. Then we can defer the cost/schedule/performance/functionality trade-offs of the later increments while we get more experience and information.

Another really good approach is a competitive concept definition phase. Basically it takes a couple of competitors down to the PDR so we will have a prototype, a B5 specification, software defined down to the unit level, and cost estimates that are within something about 25 percent of what they would most logically be. In general, we found in the defense software business that if you get the numbers to within 25 percent of what they should be, a good software manager can turn them into a self-fulfilling prophecy. Once you get things pretty firm they are more predictable. If you are a little bit under, the manager can usually motivate people to work a little bit harder and bring the job in on cost and schedule.

If you are going to do the risk management plan, you must actually live up to it. One of the things that we have been working on is the spiral model (Figure 2), which is an attempt to come up with a definition of the software process that is more of a risk-driven process than a document-driven process. One of the problems that we have with the waterfall model or DOD-STD-2167 is that our less experienced people tend to look at it and interpret it literally. If you ask them at any given point what they are doing, they will say, “I’m producing a document.” Getting people to focus on the risks that are implied by these documents is very important.

Basically what the spiral model (Figure 2) says is that what we really do, and I think really ought to do in software, is not a linear progression through a sequence of activities, but a cycle at increasing level of detail through a number of processes, the first of which is determining overall mission objectives at the beginning, overall alternatives in terms of centralized or distributed or federated architectures or things like that, and constraints. As you go into more detail, the objectives come down to individual objectives for a little piece of code. Sometimes you will be able to evaluate the alternatives precisely with respect to the objectives and constraints, but generally you will not. If not, you are in a risky situation, so you should go through some kind of a risk resolution activity.

If you don’t know what the user interface should be, the model says you ought to do some prototyping. You may continue to do this in each cycle and do the evolutionary development kind of approach. In a situation where you know enough about the job, you can go directly through a concept of operation document, a requirements specification, and a design specification, without having to spend extra time and money on a prototype.

Thus, the waterfall model is a special case of the spiral model, which you use if the pattern of risks in your program say that is the best way to go. Evolutionary development is another special case that you use when the risks determine that is the best way. The spiral model provides a context in which you can use either the waterfall evolutionary development. some other models.
The Spiral Model

<table>
<thead>
<tr>
<th>Determine objectives, alternatives, constraints</th>
<th>Evaluate alternatives; identify, resolve risks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commitment</td>
<td>Partition</td>
</tr>
<tr>
<td>Plan next phases</td>
<td>Develop, verify next-level product</td>
</tr>
</tbody>
</table>

or mixes of these, and use the risk considerations to determine the best mix.

It's a little bit difficult today to go immediately from DOD-STD-2167 to something completely different and not completely worked out like the spiral model. I think there are elaborations to DOD-STD-2167 that involve the specific development of a risk management plan, which can help document the risk considerations and point the acquisition in the right direction. If the risks say we ought to prototype, let's build a plan that gives us enough budget, schedule, and resources to do the prototype and learn the lessons and proceed from there.

Incorporating risk management plans for software is something that we can start doing today. I am encouraged that the upcoming revision of AFR 800-14 includes a requirement for a software risk management plan, and that Air Force programs are already employing them.