RECORD OF SOCIETY OF ACTUARIES
1993 VOL. 19 NO. 4A

SYSTEMS MANAGEMENT

Moderator: ROGER W. SMITH
Panelist: GARY THOMAS
Recorder: ROGER W. SMITH

- Why is it so difficult to forecast delivery times and resource requirements?
- Should I "build" or "buy?"
- What project management techniques have worked for you?
- Why do systems development projects often go out of control?
- What is the systems development life cycle?
- Why does testing always prove inadequate?

MR. ROGER W. SMITH: Gary Thomas is the president of Wessick Research. Gary has been a software consultant in insurance and banking for ten years. He has spent the last several years as an employee benefits consultant, specializing in postretirement health and corporate uses of life insurance.

I am president of PolySystems in Chicago. My company has been providing software solutions for 20 years. We're here to talk about systems management issues. Gary and I have a bad example that we'd like to illustrate as a systems development project gone sour. Then we will follow up with comments on the management process and some ideas for improving the management of a systems development project. Gary will start with his bad example.

MR. GARY THOMAS: I'd like to share with you an experience I had during the mid-1980s at a medium-sized insurance company on the East Coast. I was hired originally as a computer consultant to help on a project to consolidate the various group administrative systems via a centralized, mainframe database.

The existing systems had been developed and modified over many years. As a result, they suffered from the inevitable "spaghetti code," and many data items served multiple, hidden purposes. Although the systems were documented, much undocumented information was carried around in people's heads. Program maintenance was becoming an increasing headache. The decision was made to scrap these systems and rewrite them from scratch, rather than attempt to patch them over one more time. A two-phase approach was selected, with the first phase consisting of the database and billing, inquiry, and policy maintenance functions, together with hookups to the existing policy issue, commissions and claims systems.

All software development projects at this company were required to adhere to a well-defined, life-cycle methodology. For those of you not familiar with this concept, it is a fairly common approach used by many corporations in some form or another. The idea is to break the design and development effort into several stages, with each stage going into a little more detail than the one before it. The completion of each stage marks a checkpoint at which cost projections are refined and revisions debated.

I guess the entire project lasted for about three years, and it got bigger and bigger until eventually there were as many as 60 or 70 programmers, analysts, designers, and support staff. We were starting to slip further and further behind our deadlines.

Communication problems were becoming acute, and senior management began to get restless. Several attempts were made to reduce the scope and salvage what we could, but for all intents and purposes, it was dead. About a year later the whole thing was mothballed. Subsequently, I decided it was going to be the last big software project I was ever going to work on, and I quit and became an actuary.

I've listed about six or seven factors that contributed to the demise of this project. They were by no means unique to this project or even this particular company. I'm sure that you will recognize many of them from your own experiences.

The first thing I should say is that sometimes I think we are overambitious in what we expect from our management information systems (MIS) departments. Most software development projects are generally nonroutine, in contrast to bridgebuilding and tunnel-digging for example, where all parties have a good idea of what the finished product is supposed to look like. In addition, software design is still an evolving art, with new tools, techniques and methodologies continually appearing. As a result, even when there is full agreement on what is being delivered, it has been notoriously difficult to come in on time and within budget. In addition, the purchasers of the software have historically lacked the computer know-how to play any more than a passive role in the process, once it is under way. In retrospect, it was an impressive achievement that the project got as far as it did.

Even with a phased approach, this project was, in retrospect, ambitious. The rapid growth in staff created management problems. Furthermore, over half of the new staff were, like myself, contract computer specialists, and they had less insurance knowledge and less of a long-term stake in the project than an internal person might have had. One reason that there were so many external contractors on this project is that software specialists tend to be compensated based directly on their technical skills. The technical skills that are in demand are continually shifting and can take a long time to acquire. Consequently, it is not always practical to develop them in house. Contract programmers can also bring a breadth of experience that can be difficult to acquire in a single-company environment.

The more ambitious your goal, the more potential there is for entanglement. In trying to develop four or five distinct systems all working from the same database, we found it was a full-time job simply coordinating the database design changes. The further along we got, the more we saw of these change requests; and correspondingly, the more resistance we would see from the other design teams, as they faced the prospect of having to recode what they thought they had already finished.

Another difficulty with large projects is that as the delivery dates are pushed off into the future, the requirements are likely to shift. The reasons you wanted the system in the first place are going to change. Your external environment is going to change. Even the types of products that the system will support will change.

Management difficulties were exacerbated by a lack of clear authority, making it difficult to resolve conflicts and misunderstandings. Responsibility was split between project teams from the MIS department and a business analysis department, whose role was to put together requirements and manage the project. At this particular company there was a lack of trust between these two departments and an ignorance

of the other's fields of expertise. Both departments were ultimately responsible, but neither had real authority.

Large projects require good communication. I said earlier that most programmers are not compensated based on their knowledge of the insurance industry or the banking industry. As a consequence, you find that the systems people have little idea of what the requirements actually mean. They have a tendency to get hung up on the precise wording of a request, when what they should be doing is stepping back and asking themselves what is really needed, as opposed to what is being asked for. Conversely, many nonprogrammers tend to be ignorant of the concerns of their MIS departments. The resulting communication problems can strain even the best relationships. Communication can be a problem even within MIS departments between applications developers and the telecommunication and database experts.

The life cycle methodology adopted by this company for systems development was, in part, an attempt to formalize the development process and minimize the potential damage resulting from miscommunication. In retrospect, we might have followed it too rigidly. If the MIS department was approached, it had a tendency to answer, "we can't tell you what we'll do unless you tell us exactly what you want." The inevitable reply was, "we don't know what we want because we don't know what you can do." Back and forth this would go.

If a company adheres too rigidly to a life cycle methodology for systems development, you find that once you reach a certain checkpoint, people are very reluctant to go back and redesign or rethink something that has previously been agreed to. It looks too much like a costly admission of error. I don't believe that this approach works very well with nonroutine projects. As I said earlier, I think most software development projects are nonroutine. After all, if your software requirements were routine and standardized, it would be much cheaper and easier to go out and buy a package.

Finally, these problems were compounded by the cynicism of many people connected with the project. A significant number of people doubted that the project would be completed. This prophecy became self-fulfilling.

MR. SMITH: I want to talk about a project that I have some first-hand knowledge of and projects that I have some second-hand knowledge from observations.

Let's take a step back. The example I have comes from about ten years ago. I have prepared some of the relevant facts about the technological environment from that time.

The year that this project took place was 1983. Now the personal computers (PC) marketplace was evenly shared by three vendors. Radio Shack models were affectionately known as "trash 80s." These have been replaced. At that time the PC environment was low-powered. The development tools were definitely primitive in contrast to where they are today.

The hardware we had back then was definitely, by today's standards, very low-powered. There's probably a couple of you in this room who have wristwatches with greater processing capacity than some of those machines back then.

We wanted to do something that sounded relatively simple. We had a mainframe program that had been around for a long time. It computed cash values, reserves, and asset shares. We wanted to do what that program did. This was not an overly ambitious thing to do.

A few of the things that we saw later really focused on some of the very first decisions that were made, which were made quickly and with very limited knowledge. We may not have even realized all the assumptions that we were making, and as a result, all our adjustments did not question these initial decisions.

These critical questions seemed harmless enough. We had to select a machine environment. We needed to select programming languages. We had to decide how the user would interact with the system.

Here's what we decided. These numbers may surprise you, but back then they were reasonable. The project team decided that we had to run with just a little bit of memory, 128(k), although I am not sure why. Maybe anything much bigger than that was too rare or too expensive.

It also had to work with two floppy disks. Hard drives were very rare. They may have been unavailable in many instances. We wanted to stick with commonly available technology.

Pascal was selected as the language. I have no idea why. We also decided that screens were a must.

What did we experience as the project unfolded? We discovered that we were restricted from using the natural design of the algorithms we wanted to use to compute the numbers. We quickly realized that the decision on the memory was just too limiting. It just wasn't going to work.

Similarly, the two floppy disks you were also too limiting. We were used to dealing with larger quantities of data coming from a mainframe environment. The capacity of two floppies was too much of a toy system.

Writing screens back then was a ground-up activity. You really needed to develop everything by yourself.

What happened after we realized these restrictions? The project team decided to get very clever. There were many programming techniques that could be employed to get around some of these limitations. When you're dealing with brand new technology, it's really not a good idea to get clever in terms of programming techniques. We had people that came from mainframe environments and were used to things like memory overlays.

Understanding what should be done was difficult because the people who were asked to correct the course were people assigned to the project. These were technical people who were confident in their abilities and really wanted to prove that their ideas could work, rather than whether it's the best commercial decision. We continued to slug our way through the technical problems.

As a result, we found ourselves doing portions of the system over and over again. We were forced to write difficult data-handling techniques to squeeze all the capacity out of the machine. These were written several times. During the development, we encountered very unpleasant things about the compiler technology we were using. We had to go back and redo many aspects of the project.

We would get an idea, develop, do the work, hit a brick wall, try to get past the wall, realize we couldn't get over the brick wall, and finally retrench and try something else.

From these bad examples I came away with several guiding principles that have shaped many of my decisions since then. Number one, when you lay out a project or a systems development project, you need to insist that the hardware environment is sufficiently robust to do the job for you. Rather than plunging into something totally new and unproven, I will try to get something simple working first.

I will be wary of new technology. I don't think there is anybody who has worked in systems very long who hasn't been bitten by some aspect of new technology.

MR. THOMAS: You can see from both of our war stories that these problems occurred about ten years ago. Since then, plummeting hardware costs have altered the dynamics of where and how software is to be selected, developed, and used. Tools are certainly much better now. With the computer industry changing at such a rapid pace, we will not continue to face these same challenges in the future as newer tools and methods emerge. I'd like to explore how some of the challenges have changed from the mid-1980s and where I see improvement for the future.

The communication problems that seem to plague most development projects are primarily a result of specialization. I think that this situation is improving primarily because more and more business people have become computer literate. The PC has clearly been the catalyst here. The PC has triggered an explosion in third-party software providers, who clearly could not survive without a knowledge of the business in which they are operating. We will continue to see more and more people who are comfortable working in both spheres.

Tools are constantly improving, both in terms of power and in ease of use. This allows all of us to work at a higher level of abstraction. Powerful spreadsheets, programmable databases, and visual programming tools allow us to push much of the programming needed for ad hoc analysis away from MIS departments and into the hands of the users.

I spoke earlier about the systems development life-cycle methodology and the rigidity with which it is often applied. A phased approach to implementation helps somewhat, but it still does not address the central problem, which is that software requirements are usually fairly amorphous. I was fortunate to be given the opportunity at

the same company to initiate a development project using a rapid, prototyping approach. In this case, we put together something simple and took it out to show to the potential users, glitches and all. We got their feedback and went back and improved it, repeating this process several times. All the while, we were eliciting our own internal feedback based on evaluations of different hardware platforms, databases, and programming tools. In a fairly short time, we had agreement on a core system and implemented it.

This kind of constant feedback builds confidence. You're no longer putting people off for three to six months before you let them see what you've done. This approach provides real, as opposed to artificial, monitoring. If the project starts to go off track, it's a lot easier to make it right. It's also easier to cancel the project, and this is sometimes the best alternative. After all, once a large sum of money has been sunk into a project, nobody wants to see it fail. They'll do all they can to salvage something from it.

Prototyping also allows you to experiment with newer technology at much lower risk. Roger mentioned that new technology is something that you really have to be careful with, particularly when investing large amounts of time and money. For our particular prototyping project, we were looking at relational databases and minicomputers for the first time and were not sure what we would be able to accomplish.

I'd like to talk about some specific improvements in software development tools. One of the primary advances in software in recent years has been in databases. Ten years ago it was not possible to put up a database without extensive input from a database expert. In addition to considerations of how the data model should be structured, you had to worry about internal indices, performance tuning, and access patterns. It was a major project just getting the database up and running, let alone developing the actual application.

Over the last ten years, the relational database model, exemplified by Oracle, DB2 and Sybase, has become ubiquitous across all hardware platforms. It allows you to test your logical data model independently of the underlying physical implementation. In fact, because of the high degree of uniformity among the various vendors, it is possible to design and test a database model under Oracle and then go on to implement it under DB2, for example.

Relational databases can be viewed as a group of data tables together with a set of commands for manipulating and extracting data across multiple tables. The architecture is quite elegant in contrast to older databases, which required complex navigation paths through the data. Because you can often extend a table without having to rewrite your existing programs, these databases are ideal for prototyping and incremental development.

Data retrieval is fairly straightforward, with many vendors providing query-by-example screens, which allow computer-literate business users to directly access data themselves and design simple reports. This also frees up the MIS department, allowing them to concentrate on what they do best.

Another factor in the proliferation of relational databases is the adoption of SQL, which is a common language for accessing and updating data. Entire systems can now be ported from one database package to another, with little more than a recompile.

Another software technique that I've used a lot in the last several years is object-oriented programming. I believe this is a huge advance in the methodology used to develop software. It allows you to manage complexity, which is really the core problem.

From a base library of object class definitions, you can create your own general-purpose object classes. Each class definition describes its internal data together with its internal procedures for operating on that data. Class definitions serve as templates for the various objects that make up your system. Because each object is completely self-contained, in a building-block fashion, you can build objects from other objects without worrying about how they work internally. Examples of object classes include tables, files, windows, collections, matrices, integer, and character strings. Once you get over the steep learning curve, you can start developing some sophisticated software without too much effort.

These objects are modular; therefore you don't have to worry about programming changes to one object class causing unforeseen repercussions elsewhere. Many of the procedure definitions take up less than ten lines of code; this makes them much easier to test. Once you've tested a procedure within an object and you're satisfied that it's correct and you're satisfied with your class design, then that piece of code will stay correct forever. You can bank it. In an age when we are continually trying to work at higher levels of abstraction, that's very helpful.

Another advantage of the modular approach is that you can easily extend and improve existing applications. That's historically been one of the biggest headaches of many MIS departments. Because of the cost of replacing software systems, they can often live for 10 or 15 years, undergoing numerous modifications, each one more painful than the one before.

Object-oriented programming is particularly suited to prototyping and iterative development. You can get something up and running very quickly. It may not do what you ultimately want, but it will give you an idea of how it's going to look in the end.

Iterative development allows you to develop software with much smaller teams, and because you can shorten your development time, it means that there are fewer worries about management, communication, systems development lifecycle methodologies. More and more companies are starting to adopt this methodology.

MR. SMITH: I'd like to talk about some of the steps that I would recommend in managing a development project. Sometimes it seems that a development project has no beginning or end. The long-term nature of a development project makes it difficult to manage. There's a series of questions that I would recommend that you ask up front and push to get written answers for each of them.

A good written plan is essential. If more than one person is involved in the project, communication will be important. People will have different ideas on the project or what the system that you're developing should do. Sometimes the necessary whys get the short treatment in the development process, and you don't examine the reasons for developing a system.

Many aspects of the user's needs affect the design. How often will it run? Who's going to use it? How sophisticated is the operator? Will it run everyday or once a year? All these questions affect the design effort.

I like to build a plan as the next part of the development cycle. At this point it's important to write down all the formulas that you're going to use. Identify all the various data relationships. I see more people get into more trouble by not paying enough attention to how the data needs to come together in a system.

I think another aspect is deciding how you are going to test. The systems that I see in development and in production are becoming far more complex than my simple example. Often I find it very difficult to look at an overall process and to understand how to test it and see that the numbers are coming out. It's a good idea to think of a test plan up front.

Next, try to identify the technologies that are going to be used. To do this well, you should be constantly monitoring the new technologies. We are constantly reviewing new products to see which ones will work for us.

There is one thing that I don't attempt to do. I don't know if you noticed, but I do not like to prepare a hard schedule. From my earliest experiences, even back before I was active in software development, I have seen managers struggle with schedules. Everybody wanted to get that schedule down. How many people in this room have seen a systems development schedule that has extended a couple of years? Is there anybody here who has not seen one of those?

My feeling on those schedules that extend a couple of years into the future is that they might be the silliest thing you could possibly do with a systems development project. I truly have a lot of trouble defining what I'm going to be doing 18 and 24 months from now. I see a lot of people do it, or attempt to do it, and write it down. One of my early systems experience took place before 1983. I was on the user team that installed a new administrative system at a life insurance company.

The first thing we did was build a two-year schedule. We assembled the project teams. Within six months, all the original project leaders had been fired. The original schedules were adjusted only as necessary. The bureaucracy that had developed just for maintaining the schedule was really quite extensive. We became a slave to the schedule. Nobody stepped back and admitted that a schedule might not be possible.

During the execution of the project, it seemed like a tough project. After the project was complete, however, the software vendor referred to our company as a flag-ship installation the way to do it. I decided if that was the best way to do it, I didn't want to be around one of the worst ways to do it.

So what do we say about schedules or goals? I feel very strongly that you need to have targets. They need to be written and communicated frequently. The risk is that people believe them when they should not.

Keep the longer-term goals soft. I then like to decompose projects into workable units such as things that should be completed within one or two weeks. Create hard schedules for smaller components. Examples of the smaller steps are writing specifications, designing formulas, arranging screens, or writing the calculation code. Whatever the component might be, try to sit down and agree that an individual is going to get it done in the next week. Sit down every Monday and do this with everybody on your development staff.

What are you going to get done next week? What do you think might be done? Agree that the work step is large enough for a week's effort. If it seems like somebody is holding back or not taking off a big enough bite, then you need to encourage them to do more. This will tell you how well the project is progressing.

By getting agreement from the people doing the work, you can tell if something is falling behind. You should be able to project progress a week at a time. In this way you will be able to adjust if you get behind. You will be able to see the problems, and you can put additional resources on them. Maybe that person isn't quite right for the job. Perhaps you've asked him to do a little bit too much. Many people will optimistically overestimate what they think they can get done. That may not seem to be bad, but you can't have people telling you that they can get something done in a week when it's going to take them months.

This is the technique that I like to use. I find it more realistic than saying we'll be done in a year. You must be able to determine whether you are on schedule or not. Long-range scheduling is especially difficult if the project involves a lot of invention or uses brand-new technology.

One element of systems development that is quite important and more difficult is testing. You need to design the testing as you design the system. You need to figure out how the system will be tested before you begin to develop.

As pieces of the system are built, you should test the individual pieces. Build on components that you are absolutely sure will work. Once the components work, you can proceed to the next level.

I don't know if the next step appears in any of the classical systems development techniques. Part of the test plan has to be how to decide if the system is ready. I have frequently seen development teams test things repeatedly without a sense of what is enough. They just want it to be absolutely perfect.

In one of our recent efforts, I went to our testing unit. They were comparing some reserves calculations from new software with those from an old program. I asked how the testing was proceeding. I was told that overall things looked "pretty good," but that there were a few problems. I asked what the problems were. The group was trying to match five-year, renewable-term-reserve calculations using a

continuous-premium assumption, but wanted to use a curtate expense allowance. We were within a few pennies, but we were not matching exactly.

I'm really not sure if there is a lot of demand for the continuous-premium, curtate expense-allowance-term product. I was not concerned about this very small difference for a product combination that probably does not exist.

Quality control is quite important, but it's also possible to overtest unnecessarily. You have to approach testing and quality control with a plan that should include a point at which the software is ready for release. You want the system to work well, but it is impossible to be completely sure that there are no bugs of any kind. Looking for the last bug can seriously delay the software's release.

I'd like to shift gears and discuss what I like to call the schedule killers. One of the most common schedule killers is the ability of a project to enlarge on you. The longer that a project takes, the more likely it is to expand to some extent.

In the session prior to this one, the speakers were talking about some product design with bonus features. Ted Becker was on the panel. That discussion reminded me that regulators and new regulations cause the specifications to change.

Technology is another factor that changes schedules. It is always shifting. How many operating systems are out there right now for desktop or departmental machines? There are OS/2, MS DOS, a couple of versions of Windows, and a couple forms of UNIX. Which ones will be the most common or popular five or ten years from now?

Where is technology going to take us? If you go back five years and look at the job of supporting printers, you would find a great deal of diversity. You would have a complete interface to every printer that was out there. There is more standardization now so the effort you invested in supporting a multitude of printers would be wasted.

Be wary of technology traps. I bet that many of you have stories that you can share. New technology is very enticing and inviting. We spend so much time examining new tools, both hardware and software. This is definitely a necessary overhead or a necessary investment. There are so many ideas that look good on the surface, but when you test them with your application, you find that there are shortcomings or major problems.

I don't evaluate hardware by just looking at standard performance benchmarks. Everybody quotes their million instructions per second (MIPS), million load operations per second (MLOPS), and SPEC marks. However, when I run my application, I see more variations in my application's performance that cannot be explained in terms of the hardware vendor's benchmarks. I have been surprised when two machines that have similar published performance benchmarks exhibit significant variations on a real application.

Another aspect of new technology is something I call a time bomb. By this I mean that there is one feature or one problem that is a real showstopper for you. You just

can't get around it. These are things you obviously want to find just as soon as possible so you can avoid going down a particular path.

Other things can happen to you as well. Sometimes company management can make decisions on technology that are arbitrary and do not work out well. There is not much that you can do about this.

What other things can come up? Let's say that even after you have successfully completed all aspects of a development project, you find that the system just doesn't run fast enough. I can think of examples of systems where this kind of thing happened. I'm thinking of some administrative system examples in which a new system with very good theoretical tools was brought. New technology was utilized, but it just wasn't fast enough for some reason. The system couldn't do enough work in a short amount of time. The system was very slick and did a wonderful job, but it just didn't do it fast enough. That can be very difficult to anticipate up front. The only way to avoid this is to prototype the system on a small scale. Find out what the limitations might be before you have invested seven figures in a systems development project.

The development time itself can cause unforeseen problems. Frequently it can take more than a couple of years to develop a large-scale system. You develop it with a certain hardware platform in mind. By the time you are finished, you might be forced to move to a new computer or hardware platform. It might not be possible to make this migration so easily.

One of the difficulties with actuarial software is that we're not a large enough player for a lot of general-purpose software or hardware vendors to get excited about. They will not see market potential in developing the perfect machine for actuaries. Maybe that will change at some point.

In the meantime we need to hitch our wagons to where the hardware and software vendors might be going. For example, if you use a name-brand database product, then your systems development will be affected by the future direction of that particular database product. Changes in third-party software can affect your development project and possibly set back your schedules.

I'd like to wrap up in terms of what I think is the best or at least a good way of managing a systems project. You need long-term goals. You need to review them on a regular basis. You need to have an idea of when a delivery would make sense to the user. It would not be helpful to deliver software when your users cannot implement or use the software because of their particular business schedules.

I don't like to schedule things too far in advance because it can work against you. Frequently you don't know what you don't know. Working with short-term goals is a better way to proceed with a project. Get it in good, bite-size, meaningful pieces, things that can be accomplished in a relatively short time. This is where the most active management should take place. This is where the accountability and all the resource adjustment should logically take place on these shorter-term projects. Of course, you need a view that is headed towards that long-term goal.

As I have said, technology is always changing. We spend quite a bit of time trying to stay on top of what is new and what is on the drawing boards. It's probably impossible to do this completely. I remember the first time I picked up a PC magazine, probably ten years ago. I actually felt intimidated because I didn't even understand most of the ads. These are the types of things that you need to really stay on top of. There are a lot of new directions in processing, hardware, and peripheral devices.

There's also a lot to keep up to-date with in support software. Database products, operating systems, and development tools each offer new features and, at the same time, introduce new limitations. Sometimes it is tough to know whether new software is a net benefit or not.

When I select a particular hardware platform or a software operating system, I always like to have my next move in mind. They say that good football coaches are two or three plays ahead. You need to do this as well. Know what that next step is. This way you won't wake up surprised one day.

Finally, you need some way to validate all the assumptions that you're making and understand what the assumptions are. Some of the comments that Gary made about prototyping, rapid development techniques, and object-oriented programming can validate those assumptions. Then you can successfully and adequately complete projects and get them finished.

FROM THE FLOOR: You talked a bit about a project that failed because of the constraints of not having a powerful enough system. A successful system can run into the same problem. I found that if something works, people come out of the woodwork with new ideas and new ways of producing output from it. Very often you start out with a platform that might be sufficient, and then you realize that you don't have enough power to meet all the requirements that are needed. You want to anticipate, to look ahead to your next step, as you were just saying a few minutes ago. You might try to buy the most powerful machine you can, but somewhere along in the process you'll run into somebody who's going to try to negotiate you down. A good strategy to counter this is to ask more than you really need. But unless you feel that you need the best, you can't ask for the best. What other types of arguments have you found that work with management or controllers who are trying to limit costs. Their very logical argument is, what do you need this for? You only need 1 $\times$ X. But you feel that you might eventually need 2 $\times$ X, or 3 $\times$ X.

MR. THOMAS: I have never directly experienced that problem because I spent much of my career working in mainframe shops, where the hardware was already there. Our perspective had always been that the hardware was an integral part of any software solution that we provided, and therefore these decisions had to come from within MIS. The proliferation of networks and cheap, powerful PCs is now divorcing the hardware purchase decision from the particular needs of an application.

MR. SMITH: There are many reasons why people might want to be reined in. It could be a turf issue, and it could be cost, but it should be quite evident and straightforward in the cost-benefit analysis. The question of your need for that much hardware or that much processing speed can be a difficult question to answer. It's

rarely a question of "have to have." Nobody is going to die if you don't have it. You need to determine what makes the most business sense and try to make a case that a process should take a certain amount of time and no longer.

Generally what I find is that the cost of staff just dwarfs what hardware might cost. I normally look for the cost savings that will free up many hours of labor or make cost savings possible. If you have a tough controller, convince that person that you need the hardware to get the financial numbers a bit quicker.

MR. THOMAS: One other comment on that point is that with hardware performance advancing rapidly, you should try to look past your current limitations. You intend to use your new system for a number of years. Perhaps you could purchase hardware that can be subsequently upgraded when it is much cheaper, if you don't absolutely need all the power now.

MR. SMITH: I have seen examples of what you mentioned. As soon as you get something up that works and it's operating, it's amazing how all the restrained demand bubbles up to the surface. All these things that nobody ever asked for previously because it was pointless are now investigated.

MR. LARRY A. CURRAN: This question is primarily for Gary Thomas, and it has to do with your comments about object-oriented programming. I believe you said you've been using it for several years. Is that right?

MR. THOMAS: Yes, that's correct.

MR. CURRAN: Could you tell me if you use it on all projects, or is this a hit-or-miss thing?

MR. THOMAS: No, I've used it to put together a life insurance cash-flow projection and pricing program. As you can imagine, it's fairly complex. That's really something that I didn't start until within the last year. I spent two years or so before that really just experimenting with the technology. It wasn't something where I just dove right in and became immediately productive.

MR. CURRAN: Do you do formal object modeling as part of your object oriented approach?

MR. THOMAS: No, it's been informal, actually. Once I was familiar with the technology, I had a good idea of what most of my objects were going to look like. Most of the time you are just creating simple extensions to the class libraries provided by the language vendors.

For example, Smalltalk comes with some 200 or so classes together with several thousand methods for operating on these objects. It can take a long time to get familiar with these libraries. And that's where the steep learning curve comes in. Once you've mastered that and you've experimented with a few class designs yourself, you get a little better. Initially, it is very much a hit-and-miss affair. I've put together some fairly complex objects that I've had to subsequently tear down and redo. Eventually you get much faster at it. But it takes a long time.

MR. CURRAN: Smalltalk is the language you're using then?

MR. THOMAS: I do use Smalltalk, but my language of choice is Actor. Smalltalk and Actor are the only two languages I'm familiar with that are purely object-oriented. I know that C++ is object-oriented, although many people just use it for conventional programming. It probably runs a little faster than Actor and Smalltalk, although the syntax is not as elegant.

MR. CURRAN: One final question. You said that more and more people are now using object-oriented technology. Is this just a feeling you have or have you actually talked to several people in the insurance industry that are using it?

MR. THOMAS: I don't know of any insurance companies that use it beyond a little experimentation. It's probably best not to go in feet first until you've tried it out on some small pilot projects. The spreadsheet vendor, Borland, recently staked its future on it, and the results are only just beginning to pay off. A number of banks and investment houses are using Smalltalk.

One of the main problems with Smalltalk in the past was its sluggish performance. That's become much less of an issue now. I know of one bank that's running Smalltalk on some monstrous machine with over 130 megabytes of memory and acres of disk space. So I don't think they're seeing performance problems there at all.

FROM THE FLOOR: I'd like the speakers' suggestions on not getting tied to target dates that are very far out. I'm struggling a bit to apply this in my environment. I work in product development, and we're coming up with a new product and trying to install it on the system. If it's a variable product and I have a goal to release it May 1 and hold a sales meeting to introduce the product at a certain date, I don't see good ways to avoid those long projects, very tight deadlines, and very long deadlines. Do you have any suggestions for me?

MR. SMITH: Well, I would say that if you're absolutely forced to, you may not be able to avoid them. I think that you definitely need to have that large project decomposed into the workable pieces. Do it to the largest extent possible and manage each one of those very rigidly. As something slips, you ought to have some knowledge of this in advance. You should then be able to judge whether the longer-term deadline is in jeopardy, and perhaps you need more resources to be brought in to work on the effort.

There's no perfect solution. Sometimes things can proceed very rapidly. You might think it will take a week, and I've seen things happen in a day that fell into place. Other things don't do that. I would say that the decomposition would be very important in that.

MR. THOMAS: You need to set priorities on which features of the system you really do need immediately and which ones can be pushed off a little, or maybe simplified or cut back.

MR. JEFFREY T. ROBINSON: In that regard, how do you keep from slipping from a year to two, when you don't have a long term goal. I often find that I start with

when the thing is due and then work backwards. It sounds very appealing to have these short goals. But you need to look at when you have to have the job finished. And most people underestimate that time. How do you know how long the job should take if you don't really think from the total scope down to little segments?

MR. SMITH: Well, I think the answer is that you don't know how long the total job is going to take. I think that you need a target and you need a goal. You need something to shoot for. I still say that the hard schedule is not practical, but I agree that you need some goal and some target out there.

MR. ROBINSON: Usually I'm the person that is shot because I haven't met that goal. But I think I've never been involved in a project where I've left enough time to do it. And you know that's a whole other thing, just building in some time. And I see people who say that they do, and it's as if they commit heresy. How long is a project going to take? They say three years. Of course, when they finish after two years, everybody says why can't you do it in one? Generally you can't. But I think you have to set that long-term goal, and then just kind of see where you're going to go. But the short pieces are nice, because you can generally meet those deadlines, whereas you can't meet the rest. Gary, you mention on one project you used many contract programmers. Let's say you used consultants. Wouldn't you think that you could control those people more readily than you could control other employees within the organization? Because, in general, you have the payroll or the purse strings over those people. If the contract programmers don't perform, you can get rid of them. Whereas if your data processing department doesn't perform, generally they're unaccountable.

MR. THOMAS: Consultants generally have very marketable skills and are able to find new contracts. It depends on the economy. In 1987 it was a tough market. But in general, contract programmers with the right kind of skills found work anywhere.

If they find a company they're comfortable with, they'd love to stay there as long as they possibly can.

MR. ROBINSON: But do you think the project leader generally has more control over them than internal resources? That's really the question.

MR. THOMAS: Well, at this particular company, many of the project leaders were also contractors. At one point they put a consultant in charge of the entire project on a day-to-day basis. Presumably, you are paying good money to contractors because they are adding value. You can certainly control them in the short term, if you feel you could do without them. In the cases where you can't do without them, control becomes a problem. But in those situations, there is probably some deeper underlying problem. But I would agree that computer contractors tend to be highly responsive.

MR. ROBINSON: That was the specific word I needed. Often I find that you can't really get an internal data processing department to do, or move up, or speed up, or slow down, because you just lack the authority. You're told they know what they're doing.

MR. THOMAS: Well, the people outside the department don't understand all the issues involved in putting software together. And as I said at the very beginning, it is a lot more difficult than people think.

MR. ROBINSON: On the other hand, the data processing department, as you said before, doesn't always understand the technology of, or the technical aspects of, the insurance part of the job.

MR. THOMAS: Exactly. And that's a serious problem in an industry where most of the systems really are highly technical.

MR. ROBINSON: For this very reason I don't like to see data processing people in charge of these projects. They know the data processing aspects, but they don't know how it integrates.

Roger, in your shop, which is really a consulting firm in a way, have you ever used project contract programmers?

MR. SMITH: Rarely, just a couple of times. I can't think of many.

MR. ROBINSON: One more question for Gary. Can you give us a quick definition of what object-oriented programming is, or how it differs from regular programming? That would probably be best.

MR. THOMAS: One common type of object that actuaries might define would be some kind of vector, for example, to represent mortality tables, survivorship tables, or a set of discount factors. You would define a vector class to be derived from the existing array class, which comes complete with a set of procedures for initializing itself, loading in data, enumerating through its elements, and so on. By defining this vector as a subclass of the array class, you can inherit all its data and behavior. Perhaps you would define some additional behavior for your new vector class, such as multiplication with another vector. This will now work for any pair of vector objects, whether they represent survivorship tables, discount rates, inflation or anything else. You only have to code that once in a place where you can easily find it again. To generate survivorship tables and discount factors, you might define one further method for this vector class that would cumulatively multiply its elements.

MR. ROBINSON: In traditional programming you would define each table separately?

MR. THOMAS: Yes. The traditional language I'm most familiar with is common business oriented language (COBOL). It requires you to define all your data in one place, with all the procedures for each table elsewhere. Each table has its own initialization, loading, and access procedures scattered all over the program. With nested tables within tables, it gets very messy. And the more complex your program gets, the bigger it gets, and the easier it is to make mistakes. With traditional programming methods, the first line of code is the easiest, and after that it gets harder and harder. Conversely, with object orientation, it is quite difficult constructing object class definitions that not only solve your immediate problem, but also will prove useful far into the future. But once you've got a nice set of object classes, it is not hard to put them together.

MR. DANIEL A. CAMPBELL: We heard about global competitiveness and poor competencies. My premise is that in order for insurance companies to be successful over the next decade, they're going to have to have world-class skills in installing systems, integrating those systems, and continuously adapting those systems as their business changes. My question for the two of you is, what steps do you think insurance companies are going to have to take, starting now, to be able to deliver those world class skills in their organizations?

MR. THOMAS: In my view, the core competency of an insurance company is simply the ability to design and market profitable products that satisfy customer needs. IBM has found that it is not competitive in the software business. Likewise, although many of them do it well, I don't think that insurance companies are naturally in the software development business. It can be justified now simply because of all the unique products out there. But once we begin to see cheap software that is flexible enough to accommodate all these variations, perhaps we will start to see more companies reorganizing themselves to fit the software rather than fitting the software to the organization.

MR. CAMPBELL: Even if they don't develop systems internally, they have to be skilled at bringing in those packages that other software vendors supply. So that may be a skill set that they need to be very adept at, installing, supporting, and helping to accommodate those packages and tie them into the rest of their operation.

MR. THOMAS: That's a very important skill.

MR. SMITH: Well, I would say that companies need to be looking at technology and systems, whether they develop them or buy them. They need to get them to do a couple of things for them in order to become more competitive. Many systems are aimed at expense control, just doing a better job of acting as an insurance company. This is one huge area of opportunity where companies can become more competitive.

How many companies have the ability to understand, track, measure, and anticipate the basic fundamentals of their profitability in a very regular, quick way. This is something I think is going to be very necessary to be a player.