DESIGNING FRIENDLY INTERACTIVE SYSTEMS

A presentation on financial systems design to be given at the HPGSUG North American meeting in Orlando, Florida, in April 1981, by Mr. Jack Damm of The Palo Alto Group.

DESIGNING FRIENDLY INTERACTIVE SYSTEMS

By Jack Damm, Principal, The Palo Alto Group, Sunnyvale, Calif. (408) 735-8490

Good afternoon. I am going to talk about financial planning on the HP3000 with the Dollar-Flow planning language. My discussion will focus on three areas: 1) What financial planning is, and why there is a need for computerized planning; 2) Design considerations for "friendly" user-oriented applications; and 3) How the language Dollar-Flow is used for applications like profit planning.

THE NEED FOR FINANCIAL PLANNING

First, let's start with two questions: What is financial planning? And why is it necessary? Financial planning is making decisions about allocating the scarce resources of an organization so as to best achieve its goals. In the private sector, this usually means how best to allocate money and people to achieve profitability goals. In the public sector, it may mean how best to allocate people and dollars to provide a desired level of service. The main idea here is that the resource is scarce and, as a manager, hard decisions have to be made about how to use it. More specifically, financial planning is setting budgets, making pricing decisions, and estimating future demand for products and services, in order to achieve profit and/or performance goals.

Why is formal planning necessary? First, of course, because a scarce resource (typically money) is involved. If we had enough money for everything, then we could simply raise our salaries and retire early. Secondly, it is very important to have general agreement within an organization about how goals are to be achieved. No assumptions should be made without clearly stating and documenting them. With a good financial plan, trouble signs can be spotted earlier and corrective action taken sooner. Businesses which fail to plan effectively are the best illustration of the need for planning.

Let me offer one last reason why planning is important. For many companies, planning is a necessity because of the complexity of their operations. A typical manufacturing company may purchase thousands of parts for use in a vast array of products, and assemble them in many different locations. They cannot wait until there is no money in the till to decide that it's time to raise prices. And the current rates of inflation make this an even more important consideration.

THE TYPICAL PLANNING PROCESS

Okay, let's assume that one accepts the need for financial planning. So what's the big deal? Well let's look at the typical planning process and I'll show you.

First, planning involves lots of numbers. And these numbers change often. Financial planning involves projections into the future and is a very uncertain process. When you're uncertain, then you have to do contingency planning. Play "what if" games. What if sales are 20% higher than planned? What if the cost estimates are too optimistic? What if our product sales mix is different? Because of uncertainty, alternative plans are necessary, increasing the amount of work required to plan several times over.

And that's not all. The attempt to reach a targeted objective such as profit adds to the work. It may take several passes before all of the budgets combined with the sales estimates, cost estimates, and so forth, sum up to the desired results. The task soon becomes monumental.
company? Try changing every format statement in the model in an hour. And add to that the bother of documentation.

To summarize, manually prepared plans can be flexible, but they take a long time to do and lots of effort, especially if several passes are done. They often lack documentation. Planning with traditional programming languages takes too long to set up, is inflexible, and requires the services of a programmer.

PROBLEM ORIENTED LANGUAGES

Let me digress for a moment. For several decades now, computer scientists have been searching for a "universal" programming language. ALGOL? PL/I? APL? PASCAL? The search goes on. Each has its merits, each its disadvantages. But these "procedure oriented" languages have one thing in common: You have to be a programmer to use them. And it is altogether too easy to include bugs in even the simplest of programs. As long as there is a programmer acting as middleman between the user (or analyst) and the computer there are going to be communication problems. Maintenance problems. Resource and priority problems.

What's the answer? A planning oriented application language which incorporates the good aspects of traditional programming, but eliminates the problems. Where plans can be set up and revised easily, without having to be a programmer. What I am describing here is one example of another class of programming languages, "problem oriented" languages. Languages which have been designed to provide solutions in a general way to classes of problems. Simple enough to be used by non-programmers. Easier to debug. Self-documenting. QUERY is an example of a problem oriented language. It provides access to IMAGE data bases in a fashion simple enough to be used by non-programmers. Dollar-Flow is a problem oriented language, designed as a tool for non-programmers who want to set up tabular planning reports.

Financial planning is an area well suited to problem oriented languages. There is a considerable amount of generality in what planners do, although no two plans are the same. A financial plan typically involves mathematical operations on rows and columns of numbers. With well defined rules for the calculations. And the burden of planning in any other way gives the financial planner considerable incentive to try new appproaches.

This is a good start. But we still have to get the planner onto the terminal and communicating with the computer. How is this done? By giving him an effective tool. One which is both friendly and enables him to get the job done in a way that he understands.

DESIGNING FRIENDLY SYSTEMS

This leads us to the next point: What makes a system "friendly"? How can a system be designed so the novice or non-computer type feels comfortable with it? I ofer here a few of my ideas and techniques for developing friendly systems.

SIMPLICITY

Keep the system simple at all cost. Do not let the internal structure on the computer dictate how a system looks to the user. Let him express his ideas in his own terms. For example, the original design for the Dollar-Flow language was based on a set of documentation which I prepared for a group of accounting types. This documentation described the workings of a particular customized model on a line by line basis. I figured: What could be a better set of design specifications for a language than actual documentation? As you document your model you are also writing your program! Another example. Dollar-Flow re-orders calculation rules automatically. Thus, line 1 on a report can reference data on line 10, which, in turn, can reference data on line 20. Dollar-Flow automatically figures out the proper sequence for calculations (calculate 20, then 10, then 1) without any intervention by the user.

The following is not an uncommon occurrence: You work many hours preparing budgets and doing sales forecasts. With a board meeting just a few days away, you finish your plan. The company president takes one look at the results of the combined numbers and gives it back, requesting a 15% cut in the budget. You prepare a revised budget, repeat all of the calculations, this time under increasing pressure to get the job done fast. The day before the board meeting, marketing revises the forecast. All of the budgets must be revised again. And now it is getting late into the evening the day before the meeting. The planning process finally ends. With a good plan? No, with exhaustion. Does this seem like a doomsday tale? It's not. I've seen this happen many times. No wonder people dread budgeting time.

Combine the sheer effort required to plan effectively with the requirements for a good plan: It must be TIMELY. In a dynamic, growing company, a plan must reflect today's expectations, not yesterday's. It must be ERROR FREE. Late-night, reworked plans suffer from simple calculation errors. Errors due to using the wrong set of estimates, because they keep on changing. Imagine the embarrassment of a summation error. And with all this, the plan must remain FLEXIBLE. I worked on a profit plan for a company a few years ago which added an entire product line between iterations of the plan. And finally, when you are all done, a good plan must be WELL DOCUMENTED. What factors were used for overhead? What was the basis for the final sales figure? How was a particular number calculated? All too often, there is little documentation on how a plan was actually prepared.

To summarize: A typical financial plan involves lots of numbers, which change often. The need for many iterations makes this process time consuming and exhausting. At the same time, the plan must be timely, error free, and well documented. In short, good financial planning is not easy.

WHAT IS THE BEST WAY TO PLAN?

Given that this is the nature of planning, what is the best way to plan? How can it be done with a minimum of difficulty? Traditionally, there have been two ways of planning. Planning by hand (and calculator) and planning using the computer. Let's take a look at both of these methods and evaluate the pluses and minuses of each.

Preparation of plans manually has several drawbacks. First, because of the amount of data involved and the number of iterations, it is slow and time consuming. After many iterations, accuracy becomes a problem. The wrong estimates may be used, particularly if they keep changing. Calculation errors seem to increase with each iteration. And documentation is usually not very good.

On the other hand we have financial planning on the computer using the traditional programming languages like BASIC, FORTRAN, or COBOL. Once set up, a model written in one of these languages will run on the computer in a matter of minutes or seconds. Great! But here's the catch. The model will run very quickly once it has been set up, but it may take months to get it developed. And you need a programmer. Let's see what can happen. You start your plan well in advance of the next budgeting cycle. With six months lead time you give a precise set of specifications to an enthusiastic programmer who dutifully sets about coding your model. At the end of the first three months, he comes back to you with his first try. You patiently point out where the model is not consistent with the specifications, settle on a set of revisions, and the model is reprogrammed to your satisfaction. All set, right? No. As you begin using the model, the company president starts to change his mind (even though he reviewed the original specifications). Add a decimal place here, another line item there. Why aren't all twelve columns of data on the first page? Frustration.

What is the moral of our story? Programming a planning application with the traditional programming languages lacks flexibility. The programmer needs lead time to set up the application and has difficulty in reacting to short term changes. How about adding another division to a multi-divisional

It is important that the application be self documenting. For example, Dollar-Flow is a menu driven system. At each step of operation, the user knows his alternatives. There is little need for a "pocket guide" to the language. This is not to say that there is no need for manuals. A good manual is important. But it is a fact that few people actually read manuals. The less a system forces a user to read the manual, the more usable it will be.

Not only should the user be told what his alternatives are, the system should also help him to choose the proper response. Throughout the Dollar-Flow prompts, the most likely response is shown in brackets as the "default" response. In some cases, he can use the default response without bothering to even understand the question! For example, the prompt:

USE STANDARD OVERALL REPORT FORMAT (<Y>,N,W-WIDE PAGE)?

In one brief prompt, the user can see his options and pick one. A simple carriage return will cause the system to use the default response. And his entire report format is set up. No PRING USING or FORMAT statements. Very simple. And it can be changed easily. As the user becomes more familiar with the language, he can begin to exercise more options. With an 'N' response, Dollar-Flow leads the user through a review of the many formatting alternatives. Report formatting can even be done on a trial and error basis. Start off with the standard format, then change the column width or number of decimal places shown as needs require.

As I already mentioned, the design for the Dollar-Flow calculation rules was based on a set of user oriented documentation. Ask a user to describe how the values on the report are to be calculated in his own terms. With the addition of a few quote marks here and there, he has already written a program in the Dollar-Flow language. Self-documenting languages not only save the effort required for documentation, but make debugging easier as well.

One last comment about simplicity. Save the user concerns about internal structure through structure independent (or data base) approaches data relationships. One of the beauties of QUERY is that the user doesn't have to concern himself with all of the details of the data base to get a simple report. In Dollar-Flow, all reports are programs, all saved programs are files, and all save files contain reports. To reference data on a saved Dollar-Flow report, simply indicate the line name and the report save file name:

MARKETING BUDGET = 'BUDGET' OF 'MKTG';

There is no need for the user to know how the data is stored or even which line on the 'MKTG' report is the 'BUDGET' line which he is using.

ERROR HANDLING

Okay, so let's say you have implemented a simple system. Does this mean that users won't make mistakes? Of course not. In fact, the friendlier a system is, the greater the likelihood that the users will not be computer types. So, keep in mind that "too err is human, to forgive is good systems design." Of course, you must edit all inputs. But then use a friendly approach when the user has made an error. Because Dollar-Flow is menu driven, simple typing errors cause the system to repeat the prompt. Errors of a more complex nature, such as where a report is referenced but does not exist, generate intelligible error messages. Along with each error message give a message number. And provide a glossary with the documentation which gives even greater detail on the possible cause of the problem.

At the same time that it is informative, a system should help the user to work around problems. For example, in the case of an invalid report reference in Dollar-Flow, the user can interactively specify a different report name, or values, or zeroes. He can also indicate that computation should cease after a scan for further errors. Again, unless a particular error is extremely serious, warn the user and proceed (with his permission). Another example. As far as the mathematician is concerned, division by zero gives unworkable results. In Dollar-Flow, division by zero yields 'invalid' numbers (which print as asterisks), but doesn't stop computation. It's amazing how much more satisfying a user finds a report filled with asterisks than just a list of error messages. At least he can look at the format to see if it's to his liking.

If you must tell the user that he has made an error, tell him as early as possible. One of the most enlightened things done by the MPE operating system is to edit the job statement when a job is being streamed from an interactive session. It sure is better to find out right away than waiting for the job to begin execution to find out that a simple error has been made on the JOB statement. Report development in Dollar-Flow is completely interactive. If a user is setting up a report and he enters a calculation rule with invalid syntax, the system responds with a message immediately, and permits him to edit his error (not unlike the BASIC interpreter). It is not necessary to go into the computation step to find many errors.

MAINTENANCE AND SUPPORT

Let us assume that as an enlightened designer of friendly systems you have now designed and implemented your masterpiece. Are you done? Of course not. This is only the first step. There are two more important aspects which are critical for good, friendly systems: Continuing improvement and good support. Let me talk about continuing development first. No system is great on the first try. I am a believer in the iterative approach to systems development, if you can afford it. I am not talking about sloppy design. I am talking about the tremendous wealth of ideas that you can get from your users, AFTER you have implemented a system. Try to be receptive to the suggestions of your users (even if they are infeasible). Never give a critical user the impression that you think he has just offered a bad idea. Go out of your way to solicit ideas from your users. If the situation merits it, get involved in several of their applications. You can learn about ways the system is being used that you never thought about. Ways in which its use may be awkward. Which messages are more annoying than useful. Which features are badly needed. I send periodic questionnaires to my users (some of them even respond). This helps to prioritize new features. And users group meetings are a great boon to information flow.

How should this wealth of new ideas be integrated into an already developed system? Carefully. Do not rush a new version of a system out to users just because they need a particular feature. You must let a new version of a system be "burned in" first by a test site. Software bugs cost you credibility. Once lost, credibility is very difficult to reestablish, so reliability is extremely important. After all, would a user prefer a system with the bells and whistles he wants but doesn't work, or one which works with a few less features?

Speaking of bugs and user suggestions leads me to the question of support. There is nothing more frustrating to a user than to get 95% of the way to his computer solution only to be stopped by the application package he is using. For any reason. If you can afford to do it, good support pays great dividends. Dollar-Flow is supported in an "on-line" fashion. This means that if a user has a problem, he picks up the telephone and calls. If his problem is with an existing report, we may even log onto his system and take a look at that report. This kind of support not only helps to find and eliminate system problems quickly, but we also find out about areas where the documentation may be confusing (or incorrect). Where another feature might simplify the user's application. In short, on-line support can be another source of good ideas from users.

Let me summarize these techniques for creating friendly systems. First KEEP IT SIMPLE. Try to think like the user instead of a computer expert. Use his terms. Assume that he won't read the manual. Try to make it self-explanatory. Second, be INFORMATIVE but FORGIVING with your error handling. Edit all inputs, but don't bother the user with minor errors. When the application merits, CONTINUING ENHANCEMENT will make a much more usable system. Respond to user suggestions. But exercise good judgment in the trade-off between adding new features and degrading SYSTEM RELIABILITY.

'IT PLANNING

I am not going to take too much time on the last part of my talk. I am just going to show you a few sample reports prepared using Dollar-Flow. At the risk of violating my agreement not to make a sales pitch, I invite you to visit the PALO ALTO GROUP's booth during the vendor exhibits for a demonstration of Dollar-Flow in action.

Let me first describe the typical company profit planning cycle and the environment in which a planning tool like Dollar-Flow is used. The typical Dollar-Flow user is the accountant or company controller who is responsible for preparing the reports. Not a programmer. Most users are working on in-house HP3000 systems. With access to CRT's and a system line printer nearby. Reports are written interactively, and manual inputs are also entered via the terminal. Usually, reports are printed on the CRT for review then saved when the user is satisfied with the report. If hard copy is desired, the reports can be routed to the line printer. For generating large numbers of reports, the "batch command mode" is used, where with very little terminal input a large number of reports can be generated.

Profit planning typically begins with a preliminary sales forecast. Preliminary. Sales forecasts always change. And at the last minute, too. Often the sales forecast is done on a product-by-product basis for the first year or so, then combined with overall dollar sales projections further in the future. The near term unit forecasts are sometimes adjusted based on an overall dollar figure. The forecast is iterated several times. To make a change, the product manager just runs Dollar-Flow, inputs whichever figures have changed, pushes a few buttons, and the new sales forecast is ready. Since many parts of the profit plan depend on this sales forecast, the typical plan is usually set up with reports referencing the sales forecast report. If the figures are changed on the sales forecast, these changes will be automatically reflected on the other reports the next time they are run. Some manufacturing companies even use a multi-level sales forecast step, where a build plan (or production plan) is generated from the sales forecast.

Meanwhile, departmental budgets are prepared. Some Dollar-Flow users centralize the budgeting function and only distribute budget worksheets to each department or location. This is usually done if there are only one or two budget iterations. On the other hand, some of our customers distribute the budget preparation, with each location setting up its own budget in Dollar-Flow. In this case, figures can be input to Dollar-Flow, changes can be made, and several iterations of the budget can be done all in a matter of minutes. And budget consolidations are fun! With a few simple commands to Dollar-Flow, a whole series of budgets can be consolidated into a departmental or divisional budget. When changes are made to the low level budgets, they automatically are reflected on the consolidated budget the next time it's run.

The profit/loss projection is next. Using the data from the sales forecast, the build plan, and the budgets, and adding factors for items like sales discounts and returns, a pro forma operating statement is prepared. Often, the bottom line (profit) on this report determines what (if any) changes need to be made to the budgets. With a flexible tool like Dollar-Flow, a financial executive can even do sensitivity analysis: What if sales are 20% lower then forecast? What if our discount schedule is more aggressive and our volume is larger?

Some companies that rely on substantial amounts of debt to finance their operations combine the profit/loss projection with a cash flow projection. This is because interest paid (an item of expense on the profit/loss statement) has an impact on the amount of money required to run the business. This deter-

mines the level of borrowing, which, in turn, affects the amount of interest which is paid. Dollar-Flow, and most good financial planning languages, can solve the "simultaneous equations" this circular logic represents, and determir a level of debt and debt service which are consistent with each other. This is far more difficult when done manually.

Another procedure which is laborious when done by hand is the aging of accounts receivable and accounts payable projections. Using Dollar-Flow, once the rules for aging have been set up, a change in the sales forecast or the build plan will automatically be reflected in new receipts and payables projections.

And, finally, some companies prepare pro forma balance sheets as the last step in their profit planning cycle. This is not necessarily the way all companies plan. Or even the way all Dollar-Flow users plan. In fact, many Dollar-Flow users are not even responsible for profit planning. Instead, the system is used for a wide variety of ad hoc applications involving calculations on rows and columns of numbers. It is even used as a design tool for systems which will later be hard-coded in COBOL, FORTRAN, or BASIC.

Some of the other applications of Dollar-Flow that I am aware of include:

Product pricing. Comparing alternative prices for a single product (the plotting capability is great for comparisons). Or comparing profit percentage across an entire product line. Financial ratio analysis. Comparing selected financial ratios against industry standards or company objectives. Capital budgeting. Rates of return and discounted cash flows can be calculated easily using built-in financial functions.

Performance reporting. Variance reports showing actual budgets or profits versus plan. How sales are doing against target. (One Dollar-Flow user generates 500 graphs every month showing product line sales performance for every branch of every distributor who markets his products!)

SUMMARY

Let me leave you with a few parting thoughts. Financial planning is not an easy process. Figures change. The whole approach to a plan may change. And you need your results yesterday. Traditional systems design and programming methods are not going to be effective in this kind of situation. Use a better approach. With a friendly, problem oriented planning language like Dollar-Flow, applications nightmares can become applications successes,