TITLE: "WHAT IS A GOOD PRORAM SPEC?"

by Tim Bryce
Managing Director
M. Bryce & Associates (MBA)
P.O. Box 1637
Palm Harbor, FL 34682-1637
United States
Tel: 727/786-4567
E-Mail: timb001@attglobal.net
WWW: http://www.phmainstreet.com/mba/
Since 1971: *"Software for the finest computer - the Mind"*

"Whenever you see a ratio of 25:75 analysts:programmers you will find systems analysis being performed at the wrong time and by the wrong person"
- Bryce's Law

INTRODUCTION

Since the industry is preoccupied with producing software faster (and not necessarily better), let's stop and consider how we typically approach programming and allow me to put my spin on it. There are fundamentally three aspects to any program development effort: defining the program's specifications, designing and writing the program itself, and testing it. The software engineering gurus in the industry are primarily concerned with the internal design of the program, but there is now a raft of consultants trying to determine the best way to approach the program externally. Why? Because there is now many ways for producing software than just writing source code using a common text editor; e.g., visual programming aids/prototyping tools, workbenches, 4GL's, program generators, etc. Such tools take the need for writing precise source code out of the hands of the programmers and allows them to concentrate on basic screen and report layout. They are excellent tools for most programming assignments, but they cannot do 100% of all of the programming for all applications. We still require professional software developers with an intimate knowledge of programming languages and design techniques. Regardless if we write a program by hand, or use some sort of interpreter/generator, we still need to provide the programmer with precise specifications in order to perform their work.

Seldom do companies make use of a uniform approach for producing program specifications. It is not uncommon for programmers to receive specs in obscure ways, such as a memo from an end-user (the back of a cocktail napkin is my personal favorite). Rarely are specifications given in a consistent manner that can be evaluated for completeness. A standard approach would improve productivity and communications within the programming staff alone.

What should a good program spec include? Actually, its not too difficult to figure out...

ELEMENTS OF A PROGRAM SPECIFICATION

Each program should be defined in terms of:

1. **Input Descriptions** (to collect data or request an output) - be it implemented by a GUI, command line interface, verbal, optical, or through some other screen interface. All inputs should include:
   a. Name, alternate ID, program label, description.
   b. Defined layout and examples.
   c. Input transaction specifications, including default values and editing rules for data to be collected.
   d. Messages; e.g., data validation, and general processing.
   e. Panels (for screens).
   f. Relationship of inputs to outputs.

2. **Output Descriptions** (to retrieve data) - be it implemented by a GUI, printed report, audio/video, or through some other screen interface. All outputs should include:
   a. Name, alternate ID, program label, description.
   b. Defined layout and examples.
   c. Panels (for screens), maps (for reports).
   d. Messages; e.g., general processing and program specific information/warning/error messages.

3. **Data Structure Descriptions** (data bases, files, records, and data elements). NOTE: Programmers should NOT be in the business of designing data bases as they will only do what is convenient for their application, not others (thereby missing the opportunity for a company to share and re-use data). Physical files should be defined by Data Base Administrators.
   a. All data structures should include: Name, alternate ID, program label, description. They should also include...
   b. Data Bases - organization, key(s), labels, volume/size, backup requirements, internal structure.
   c. Files (both primary and working) - organization, key(s), labels, volume/size, backup requirements, internal structure, file-to-file relationships.
   d. Records - form, length, key(s), contents, record-to-record relationships.
   e. Data Elements - class, justification, fill character, void state, mode, picture, label, size, precision,

(continued on page 2)

(continued from page 1)

scale, validation rules. If generated data, rules for calculation. If group data, rules for assignment.

4. **Program Description:**
   a. Name, alternate ID, program label, description.
   b. Characteristics: Required processing speed, memory requirements.
   c. Dependencies to other programs externally (e.g., batch job stream).
   d. Dependencies to modules internally (e.g., DLLs, subroutines, etc.)
   e. Functions to be performed with Inputs, Outputs, and Data Structures (create/update/reference).
   f. Special processing rules (logic for processing)
   g. Command language required to execute the program (e.g., command files, JCL, etc.)
   h. Physical environment where program will be executed.
   i. Test Plan and how to assemble test data.
   j. Method of implementation - programming language(s) to be used, design techniques to be observed, tools to be used.

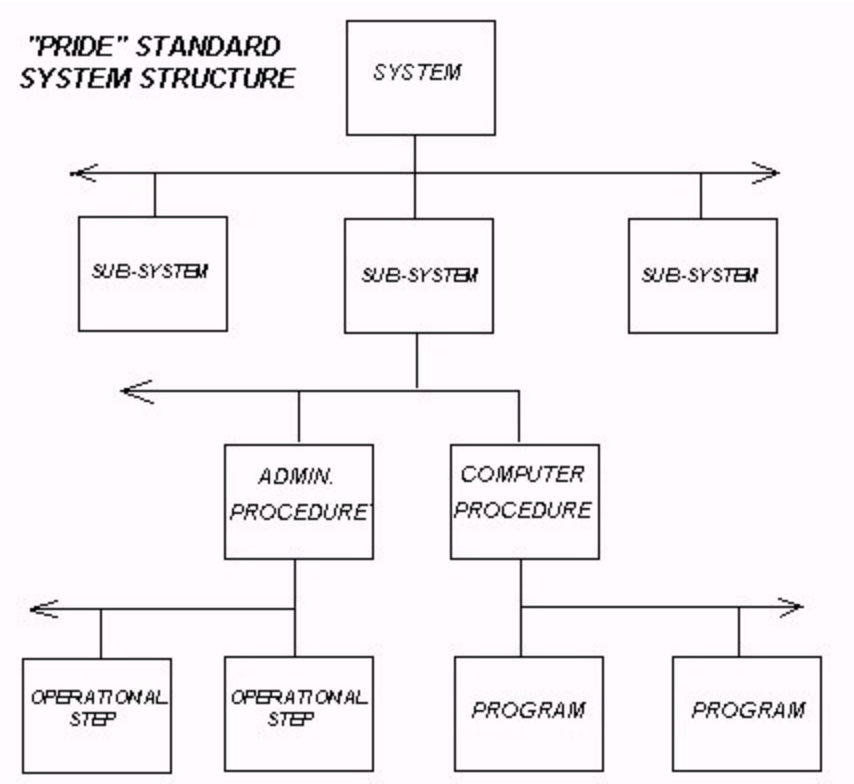
In-house software engineering standards complements any program specification (and should provide guidelines for writing the specification). Such standards define "best practices" for design and conventions to be observed during programming. As an aside, the objective of software engineering should be: Maintainability (easy to correct and update), Performance, Design Correctness (proof), International support (to accommodate languages and cultures), Integration (sharing and re-using code), and Portability (platform independence).

Between the programming spec as listed above and a good set of programming standards, it becomes rather easy to implement any program, be it by hand or through the use of a generator. As a matter of policy, specifications should be written under the assumption that a program generator will be used. This forces us to be more precise in our specifications.

OKAY, SO HOW DO WE GET THERE?

When it comes to assembling a program spec, I am of the philosophy that *"You eat elephants one spoonful at a time."* It is difficult to gather the specs for a single program in one fell swoop. Plus, when we consider most development projects today involve more than one program, the problem is further complicated. For major development efforts, I am of the opinion that "layers" of documentation are required. For example, under "PRIDE-ISEM, we view a system as a collection of subsystems (business processes), implemented by procedures (administrative and computer), administrative procedures consist of operational steps (tasks), and computer procedures consist of programs (which can be sub-

(continued on page 3)



(continued from page 2)

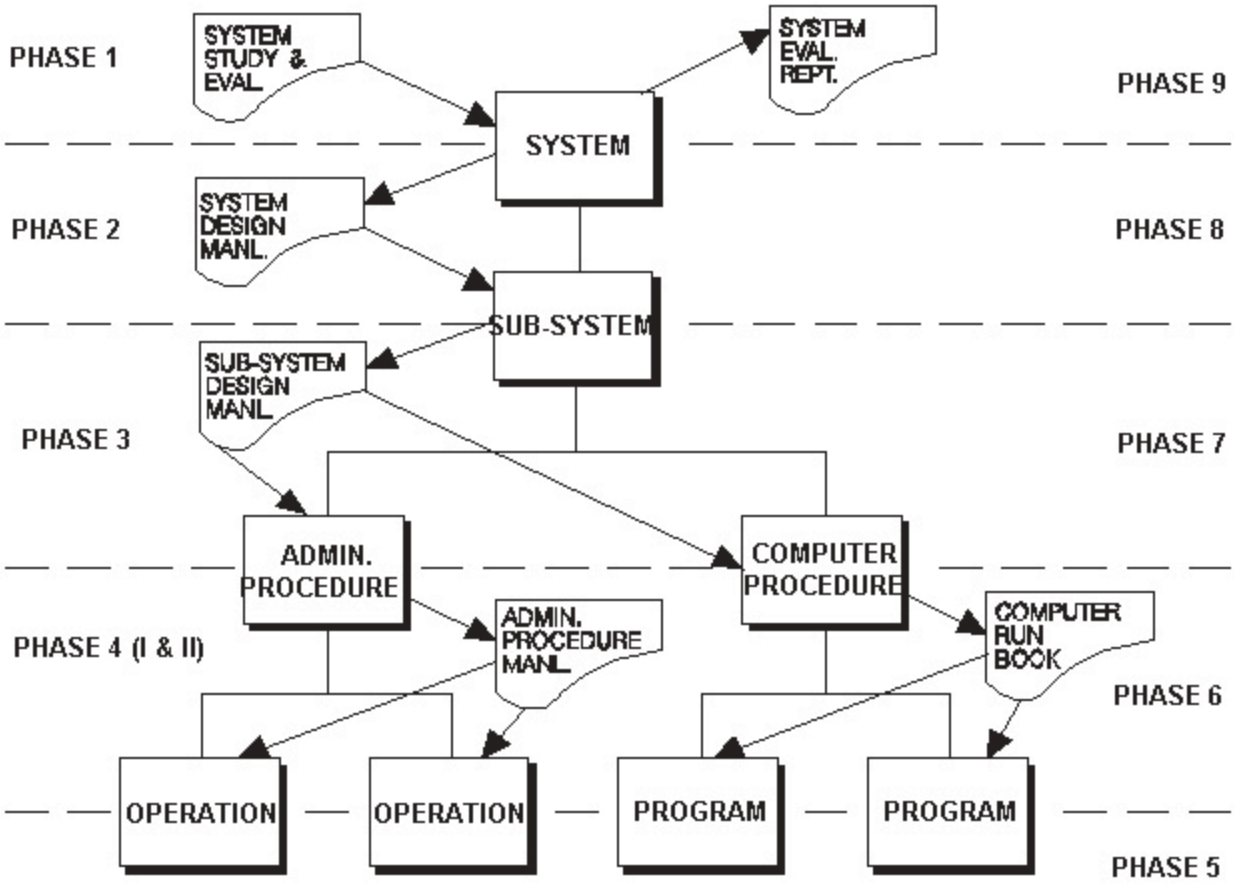
divided into modules if so desired).

Basically, "PRIDE" views a system as a product that can be engineered and manufactured like any other product. From this viewpoint, we can make use of other engineering techniques, such as a top-down blueprinting approach to documentation where levels of abstraction define the different levels in the system hierarchy. For example, the Phase 1 Information Requirements contained in the "System Study & Evaluation Manual" define what system(s) are needed (either new or existing systems requiring modification); the Phase 2 "System Design Manual" includes specifies the sub-systems; the Phase 3 "Sub-System Design Manual" specifies the procedures in the business process; the Phase 4-I "Administrative Procedure Manual" specifies the operational steps, and; the Phase 4-II "Computer Run Book" specifies the programs. This blueprinting approach allows us to progressively refine our specifications until we reach the bottom of the product structure. In other words, it is not necessary to define everything about an Input, Output, File, or Data Element all at once, but rather to initially identify the need for them, then progressively refine the details until we are ready to program.

This approach to documentation is sometimes referred to as "step-wise refinement" whereby the design of a structure, such as a product or building, is refined over various levels of abstraction. Only when we have completed these architectural designs can the product move to manufacturing/building. Imagine trying to build an automobile or skyscraper without such a technique. It would be virtually impossible. Why should systems be any different? In order for this approach to work, you must accept the concepts: a system is a product; that there are various levels of abstraction to it, and; there are standards for documenting each level. This is considerably different than a "forms driven" approach to development; e.g., fill out forms in a regimented sequence without any thought in regard to the design of the system. Instead, documentation should be a natural by-product of the design process.

This also makes a clear delineation in terms of "types" of specifications; for example "information requirements" and "programming specs" are miles apart in terms of content and purpose. Whereas the former is a specification regarding the business needs of the user, the

(continued on page 4)



(continued from page 3)

latter is a technical specification for the programmer to implement.

For more information on *"Defining Information Requirements,"* see "PRIDE" Special Subject Bulletin #4, Dec. 27, 2004

http://www.phmainstreet.com/mba/ss041227.pdf

This blueprinting approach also highlights the need for basic systems work in the earlier phases of design, with the programmers being the beneficiaries of more precise specifications (as opposed to vague concepts), thereby simplifying their job. The Japanese use pyramids to describe the differences in this focus between the up-front systems work and the back-end programming work (see below).

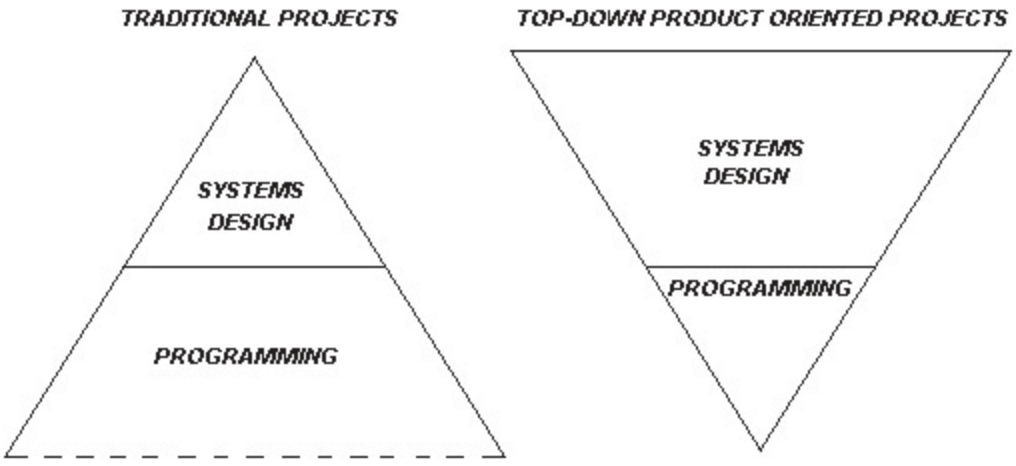
Both pyramids represent the scope of work in a project. In the pyramid on the left, very little time is spent up-front in system design. Consequently, more time is spent by programmers "second-guessing" what the software should be doing. Contrast this to the inverted pyramid on the right where more time is spent in systems design, thereby producing more detailed specifications and, as a result, less time in programming.

CONCLUSION

So, what is a good program spec? Anything that eliminates the guesswork for the programmer. Consider this: if the up-front system design work was done right, programming should be less than 15% of the entire development process. Then why does it currently command 85% of our overall time (and financial resources)? Primarily because we have shifted our focus and no longer believe we are being productive unless we are programming. After all, programming is perhaps the most visible evidence of our work effort; system design is less tangible.

Let me illustrate, back in 1976 I took an entry level COBOL training course from IBM in Cincinnati. Our class was divided into teams of three people and each team was given problems to solve. When we received an assignment, the other two programmers in my team immediately started to write code, key their entries (Yes, we used keypunch equipment back then), then compiled the program. Inevitably, there were errors and they would go back-and-forth correcting errors until they finally got it right. As for me, when I got an assignment, I would pull out a plastic template and paper, and work out the logic of the program before writing the code. I would then key and compile, and would always complete the assignment

(continued on page 5)



(continued from page 4)

before my partners. Curiosity got the better of me and I asked them, *"Why do you do it that way?"* They contended this was how they were expected to work by their superiors; that they weren't being productive unless they were producing code. I countered that even though they were faster at producing code, I was still beating them every time, simply because I was thinking the problem through.

The IBM rep who registered me for the class happened to stop by and asked me if I was learning anything. I said I was learning more about "programmers" than I was about "programming." I am still learning about programmers, but I haven't noticed any significant changes in their attitudes towards development since then. True, we now have some great tools to expedite programming. But if they are so good, why doesn't our backlog diminish? Why are we constantly in a maintenance mode? Why can we never seem to complete our major applications on time? Why? Because we are no longer doing the up-front work.

Just remember, it is always *"Ready, Aim, Fire"* - any other sequence is simply counterproductive.

END

"PRIDE" Special Subject Bulletins can be found at the "PRIDE Methodologies for IRM Discussion Group" at:

http://groups.yahoo.com/group/mbapride/

You are welcome to join this group if you are so inclined.

"PRIDE" is the registered trademark of M. Bryce & Associates (MBA) and can be found on the Internet at:

http://www.phmainstreet.com/mba/pride/pride.htm