How To Select a Programmer

Alfred J. Barchi
ajb@ajbinc.net
http://www.ajbinc.net/

Synopsis

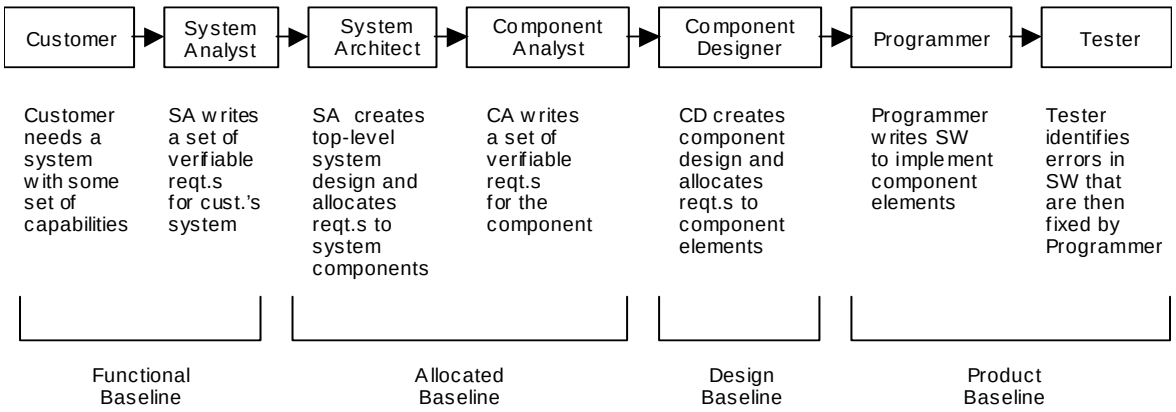
Recruiters and recruiting agencies constantly assess and attempt to quantify the skill sets and capabilities of potential employees for various kinds of positions. Hiring managers do the same thing. Additionally, with the Internet, automated resume services and large resume databases, much of this activity is becoming automated. It is, therefore, important to be able to recognize and quantify those characteristics and capabilities that are essential to performing a particular task.

The author's primary area of expertise is the implementation of software-based systems, so the author has a keen interest in the recruiting task as it relates to programmers. This paper explores some of the elements that the author considers essential to the programming task and discusses how they relate to the recruiting effort.

Introduction

A lot goes into the development of any system, including a software system. For example, the man-millennia that went into the development of Microsoft Windows XP, all 60 to 80 million lines-of-code of it, has got to rival some of the most ambitious projects ever attempted by man. And, naturally, a lot has been written about the development process.

Consider, for example, the following diagram:



This diagram represents the typical waterfall development cycle with multiple iterations and feedback from later steps to earlier steps. It is drawn here with emphasis on identifying the personnel responsible for performing each step. Of course, there are also a lot of ancillary elements to this process, such as program management, quality assurance, configuration management, verification and validation (really a component of configuration management), technical writing (user and maintenance documents), etc. Here, however, we are primarily interested in the design and implementation aspects of the system development process.

In small systems, a single person may perform some or all of the functions. In larger systems, however, there is simply too much work to be performed by one person and division of labor becomes essential. In such systems, it is essential to identify the proper skill set for each job function. In this paper, we will concentrate of the skill set necessary for the programmer function.

Ability to Understand and Comprehend the System Design Process

One of the first things that come to mind when looking at the diagram above is this: When you were a child, did you ever play "Whisper Down the Lane?" You can't just have your system analysts write a set of functional requirements and toss them over the wall and expect that somewhere down the line people are going to understand and interpret them in the same context in which they were written.

On many large projects, there is a chief engineer who performs the task of making sure that all elements of the development team, from the system analysts to the testers, are on the same page. The chief engineer is typically a very senior, very experienced individual who knows enough about each task in the development process that he can understand, communicate with, and oversee each task. Typically, as part of the configuration management process, an engineering review is conducted whenever a change is made to one of the system's baselines. The chief engineer chairs this review. It is essential that parties from each task are involved in the review. One of the chief engineer's functions is to stimulate and sustain the discussion so that he can listen carefully to each party to ensure that everyone has the same understanding of what is being built.

Conversely, each party must understand enough about the functions of the other party to communicate with them, to understand the system that they are helping to build, and to understand their part in the process.

This, of course, includes the programmer. He must understand the system, how the elements that he is implementing function in the system, what functional and design requirements they satisfy, and how they will be verified. Of course, the programmer's work will be reviewed before it is accepted as part of the baseline. And in a perfect world, all of the parties reviewing the programmer's work product will understand enough about programming to understand why the programmer implemented it the way that he did.

Unfortunately, the world isn't perfect, and the programmer is responsible for communicating his implementation decisions in a way that other parties will comprehend, and for being able to understand and discuss changes that are proposed to his implementation. All too often, the author has witnessed programmers loudly proclaiming, "But it will take 50,000 lines of code to make that change!" for any change that they don't want to make.

The programmer must have a knowledge and understanding of the system development process.

Of course, we aren't born with this knowledge, and it isn't generally taught in schools – it's acquired on the job. Therefore, we have senior programmers and junior programmers.

Ability to Communicate with Domain Experts

The author once worked on a project that had an orbit analyst. This person was a true domain expert. The author got the job of being his liaison because nobody else on the project was able to communicate with him. The author had to have enough understanding of orbital mechanics to be able to take this person's work product and reduce it to software.

The alternative was to teach the orbit analyst how to write software (he already knew how to write simple programs in APL, but his knowledge of data structures and programming paradigms was non-existent).

It is important to point out that knowing enough about orbital mechanics to communicate with the domain expert isn't the same thing as being able to do the orbit analyst's job. This author humbly admits that he is no orbit analyst. However, the author does have an engineering background, and was able to at least comprehend the math involved and the concepts that the orbit analyst was trying to convey.

We can generalize from this. *The programmer has to have the background and intelligence necessary to comprehend the concepts being conveyed by any domain expert with whom he interfaces.*

Does this mean that you have to be a chemist to write software for a drug company? Well, no, it doesn't. But if the system you are developing models chemical processes, then you had better have enough knowledge of chemistry to comprehend what the chemists are saying and to ask the right questions in order to clarify things that you don't understand.

Knowledge of Programming Paradigms and Data Structures

To borrow an aphorism from a completely different field, "Programming languages don't write software, people write software." You might think this would be obvious.

However, the author can recall a conversation that he once had long ago with a senior software development manager regarding the programming language Ada. This manager insisted that the Ada language, with its strict data typing and type checking, would prevent his programmers from making many of the common mistakes that they were currently making, and that therefore, he could reduce costs by hiring less senior, less competent programmers at lower salaries. He went on to bid a very large development project at 3.5 lines-of-code per hour, even though his historical productivity measurements showed a productivity rate of only 1.5 lines-of-code per hour, since Ada was reputed to enhance productivity. The author was unable to persuade him against this.

Naturally, his proposal was the low bid, so he got the job. Unfortunately, with his lower cost, less competent staff, his actual productivity rate for this job was only 0.9 lines-of-code per hour. Even worse, when his product was finally "good enough" to ship, it was so slow that the servers that it was supposed to run on had to be replaced by the fastest, most powerful (and most expensive) servers that the server manufacturer made in order to get marginally acceptable performance. The company took a major bath on that one.

What went wrong? Well, just about everything you can imagine. For example, there was one component that loaded several hundred variables from text files. It then used these variables periodically. But, rather than using a hashing algorithm or an insertion sort to order these variables as they were loaded, the component did a sequential search of the variables each time it needed to access one. This resulted in major performance problems. Why was the design implemented this way? Well, the less expensive programmer who implemented this component didn't know anything about hashing or sorting, and didn't have the knowledge or experience necessary to realize that it mattered.

A truly competent programmer has to have knowledge of a wide variety programming paradigms and the data structures necessary to implement them. Different programming languages embody different paradigms and are designed to facilitate their use.

For example, C++, Objective C, Java and Smalltalk all facilitate the use of 'objects', an important paradigm in programming. Lisp implements list processing and supports the concept of lists of heterogeneous elements. Pascal was used at one time to teach students about complex data structures, unions and sets. Ada was based on Pascal. C implements pointer operations and linked lists. And so on.

Also, we mustn't forget assembly language. Software runs on computers, and it essential to a thorough knowledge of programming paradigms that we understand machine architecture. This also facilitates an understanding of operating systems and networks.

Then, of course, there are subjects like automata theory, compiler design theory, expert systems and natural language processing. There are distributed processing and

distributed communications models. There is queuing theory. There are relational algebra and database design paradigms. There are mathematical algorithms and numerical methods, such as Runge Kutta integration. We already mentioned sorting, searching and hashing.

This is not intended to be a complete list. It is merely presented to demonstrate that programming is not a trivial endeavor. Programming is a creative process. And like all creative processes, the human brain cannot extract, correlate, synthesize and create from that which it doesn't know. And although we may be using C++ for a particular project, that doesn't mean that the elements we implement won't involve the use of many programming paradigms besides 'objects' and 'classes' of 'objects'.

It's important to realize that every functional requirement and every design requirement are filtered through the prism of the programmer's knowledge and understanding before they can become actualized in your system.

The competent programmer must have a working knowledge of a wide variety of programming paradigms and data structures.

Again, we aren't born with this knowledge, and it isn't generally taught in schools – it's acquired through study, reading and on the job experience. Therefore, we have senior programmers and junior programmers.

Understanding of Lifecycle Costs

There are many factors that go into determining the lifecycle costs of a system and its components. Factors such as adaptability, flexibility, usability, testability, performance, maintainability and security, just to name a few. These concepts are topics for another paper, or series of papers. It is important to the programmer to understand that they exist, that they may trade off against each other, and how to weigh and evaluate them in order to minimize lifecycle cost.

Many people who profess an understanding of such things as local optima and 'simulated annealing' claim that implementing systems in such a way as to minimize lifecycle costs is "too expensive!" As bizarre as this might sound when stated this way, it makes a great deal of sense when taken in context. The reason has to do with risk. You have to discount the expected gain of a design approach by the risk that it will fail.

Approach 'A' is expensive to implement. So is approach 'B'. The cost of failure of both approaches is high. If we knew in advance that approach 'A' would work, i.e., that it would be secure, meet the performance requirements, etc., then we would choose approach 'A' over approach 'B' because it would have a lower lifecycle cost. However, due to various reasons, approach 'A' is riskier than approach 'B' (it requires more lines of code and we might not meet our deadline, it is more difficult to implement, it requires solving some poorly defined problem, etc.). We have to weigh the tradeoff between 'A'

and ‘B’ in the context of this risk and make our decision on which approach to take based on this.

The competent programmer must know how to properly assess lifecycle costs of different design and implementation approaches.

Examples – Typical Job Postings

The following was taken from an actual job posting downloaded at random off of the Internet:

> Qualifications include Degree or equivalent experience in CS, MIS, accounting, or related technical field coupled with 2 years of C#.NET experience. Additional experience desired includes SQL Server and/or other RDBMS, client server computing, multi-tiered architectures. Experience with accounting systems; mutual fund reporting/accounting; Word/Office automation; Web Services; and ASP/HTML are all pluses. Again, seeking preferably at a minimum someone with technical lead capabilities (initially over 2-3 developers) and probably windows based application development background.

Going line by line, we first find a degree requirement. It’s usually good practice to pre-qualify your job applicants by insisting that they meet certain educational requirements. In other words, someone with sufficient intelligence to complete a 4-year college degree program probably has enough intelligence to do this particular job. This job apparently involves the development of a small, web-based accounting system on a Microsoft Windows platform, or a small component of such an accounting system (notice the group size of 2 to 3 developers).

The author of this posting would probably have preferred to ask for 5 or even 10 years of experience with C# .NET. Unfortunately, as of this writing, it hasn’t been around that long. The fact is, if you know Java and C++, it won’t be difficult to pick up C#.

Experience with “SQL Server and/or other RDBMS” may mean that the poster is looking for someone with an understanding of databases, or someone with an understanding of database managers, it’s not really clear which.

Experience with “client server computing, multi-tiered architectures” suggests an architecture with application servers running on one or more platforms and interfacing with remote clients over the Internet, either through a browser or some other web-based tool, as opposed to the client-server model used by Active-X or the client-server models used in other distributed applications.

“Experience with accounting systems; mutual fund reporting/accounting; Word/Office automation; Web Services; and ASP/HTML are all pluses” suggests that the poster is looking for someone who understands the programming paradigms commonly used in such systems.

The manager who wrote this job posting will probably get someone who can do the job that he needs done. (I'm not suggesting that this is a perfect posting, merely that it is adequate for its intended purpose.)

The following was taken from another randomly selected posting:

> Senior level Java Batch Programming in a Websphere environment.
> Senior level experience in transforming requirements into a detailed design.
> Senior level experience with design and modeling objects with UML and Rational Rose.
> Senior level experience with Java and Websphere
> Senior level Object-oriented Design and implementation
> Senior level Experience with using and analyzing programs with Jprobe
> Senior level Experience in developing unit test cases and implementing them with Junit.
> Strong Java, J2EE and Websphere development experience needed.

It might not be obvious, but the term "Senior level experience" is actually meaningless, since it means something different to each person who reads it, and there is a low probability that anyone who reads it has the same understanding of it as the person who wrote it.

"… Java Batch Programming in a Websphere environment" indicates that the poster is concerned about the "learning curve" associated with this position. The posting is for a 4 to 6 month contract. It's not clear whether this is a "contract-to-hire" position, or whether the poster just wants to fill a temporary need.

"… transforming requirements into a detailed design" – Gee, isn't that kind'a, sort'a, more-or-less what every programmer does?

"… design and modeling objects with UML and Rational Rose", when taken in the context of the rest of the posting, suggests that the poster is more concerned about a knowledge of UML and Rational Rose than an understanding of what objects are, object-oriented design and programming paradigms, or how to design and model objects.

"…Java and Websphere" – Once again, a pre-occupation with the use of the tools.

"…Object-oriented Design and implementation" – One wonders whether the poster cares about an understanding of object-oriented paradigms or merely the process of defining object classes in UML and Java.

"…using and analyzing programs with Jprobe" – Again with the tools.

"…developing unit test cases and implementing them with Junit" – Here again, the poster is interested in a knowledge of a particular tool.

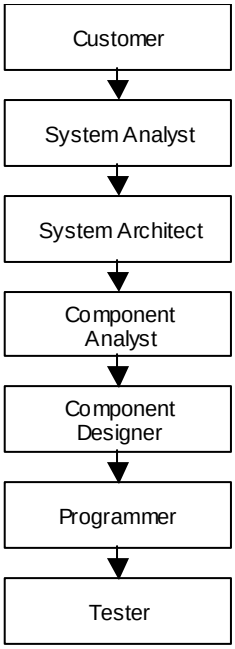
"Strong Java, J2EE and Websphere development experience needed." If the project is proceeding on a short timetable, then this kind of experience is necessary in order to avoid the learning curve.

The hiring manager who wrote this posting is focused on the mechanics of programming, not the thought processes behind it. Perhaps that is because he has a tight schedule to meet. On the other hand, maybe he views programming as a simple mechanical process that high school kids can excel at (as in all of those movies and TV shows that feature the nerdy teenage hacker/computer genius). In either case, the lifecycle of a product extends well beyond its first delivery (unless it's really wretched). Yet, his posting is totally devoid of any indicia that he understands this.

This manager and this project will probably fail.

Hierarchical Thinking

Many people think of the chart that was presented earlier in this paper this way:



In other words, they view the design team as a hierarchy, with programmers near the bottom of the food chain, just slightly above testers and pieces of rotting meat. The manager who wrote the second example job posting above probably thinks this way.

There are a couple of reasons for this kind of thinking. First, the development process is frequently depicted this way in textbooks and articles that discuss it. The second reason

is that as you move from left to right (or top to bottom) along the development process, the tasks become more and more detailed, and it requires more and more people to implement them (for example, in "The Mythical Man-Month," author Frederick Brooks opined that testing should take up fully 50% of the development effort – this was based on his personal experience and historical data on large system developments at IBM). This necessarily means that each individual assigned to the more detailed development functions has less leverage to cause problems for the development as a whole than the people closer to the customer side of the development.

Normally, you would tend to put more senior people into higher leverage positions, particularly when they may be required to communicate directly with the customer. Also, there are just so many first rate programmers available for any job, and you're lucky to get 1 or 2, much less 50 or 100.

But it's a mistake to think that your top programmers need to be any less competent at what they do than your system analysts or system architects need to be at their jobs. You can't make up for quality with quantity. It's like ionizing vs. non-ionizing radiation, or throwing stones at a target across a river. If you aren't strong enough to throw a stone across the river then it doesn't matter how many stones you throw, you'll never hit your target.

Conclusion – How to Select a Programmer

As was previously stated, programming is a creative process that requires an ability to extract, correlate and synthesize concepts from a broad knowledge base. In order to select someone who can perform such a task, you must look for certain indicators in his background, both in his resume and during an interview. Aside from the normal things, such as: Does his resume focus on his duties or on his accomplishments? Is he a convicted felon? Etc. There are some specific things that you need to look for. Naturally, senior programmers need to possess these qualities to a greater degree than junior programmers.

You must look for indicia of a deep and broad understanding of programming paradigms. How many different programming languages has the programmer used? Does he understand machine and network architectures? How many different operating systems has he worked with? What types of applications has he developed: embedded, real-time, natural language processing, expert systems, data management systems, accounting systems, etc.?

You must look for indicators of native intelligence. What kind of education does he have? Is he articulate? What different kinds of domain experts has he worked with in the past? Can he talk fluently about the work that he did with them?

You must look for evidence that he understands risk, lifecycle costs and how to minimize them. Does he think in terms of lifecycle cost? Does he view a system as being

complete after the first delivery, or does he view it as a living entity that is maintained and that evolves over time?

You must look for signs that he understands the development process. Does he understand the role of configuration management, or even what it really is? Does he understand the role of quality assurance, and why the QA group should be independent of and not report either directly or indirectly to the program manager? Does he understand what baselines are, and why you have to maintain all of them simultaneously? Does he understand what requirements are (i.e., they're not something that you create, they are something that you discover through analysis and then document)? How many different kinds of development environments has he worked in? How many different business environments?

Lastly, is he fast? This characteristic hasn't been mentioned before, and it's difficult to gage without subjecting the applicant to a proficiency test (which probably won't work very well anyway and is demeaning to applicants with any experience – you don't want to start a new experience with an insult). You can, however, estimate this to some extent by asking him about the size of his previous jobs, how much he did personally, and how long it took him to do it.

Also, nothing slows a programmer down more than having an inadequate understanding of what he's doing and why he's doing it. If he has all the other essentials, then he at least has the potential to be fast.

The least important characteristic to look for, except in rare and extreme cases when you need it done yesterday, is familiarity with a particular programming language or a particular set of tools. Good programmers can learn these things virtually overnight.

If you were to think of it in terms of Chomskian Linquistics, the use of a particular data structure or operation would correspond to the 'deep structure' or 'd-structure', and the expression of that data structure or operation after it has been transformed into a particular programming language would correspond to the 'surface structure' or 's-structure'. For context-free languages, such as computer languages, this is easily done by looking things up in a reference book. The actual lookup time is trivial compared to the time required to form the underlying concept, and once you've looked up the same data structure or operation a few times, you know how to make the transformation and you don't need to look it up anymore. Forming the 'deep structure' construct is a cognitive process; forming the 'surface structure' transformation of the construct is a mechanical process.

For an extreme example, the Forth language (a threaded-interpretive language) relies heavily on pointers. But the original Forth interpreter was written in FORTRAN back in the 1970's, and FORTRAN has no pointer construct. Charles Moore (the creator of Forth) was able to translate his pointer operations and data structures into FORTRAN anyway. It wasn't a simple, straightforward translation, but it worked. It was a mechanical process of writing FORTRAN code that performed the pointer operations on

abstract pointer structures represented by arrays in FORTRAN. In order to do this, Charles Moore had to know what pointers are and how and why they are used.

For another somewhat extreme example, the author once worked on a project back in the early 1980's that used Pascal. The author selected Pascal because it was the only high-level language available at the time on both the development platform (Vax minicomputer) and the target machine (Intel 8086 cpu – we were developing our own CPU board). The idea was to develop our software in a machine-independent environment while the target environment was still under development. But, the application called for bitwise logical operations, and the Pascal language doesn't have any bitwise logical operators. So the author created some – written in Pascal. In order to do this, you have to know what bitwise logical operators are and how and why they are used.

Remember, programming languages don't write software. Tools don't write software. People write software. It's a creative process, a cognitive process – not a mechanical one. Focus on the thought processes of programming – not the mechanics, and you'll select good programmers.