©2008 Microsoft Corporation. All rights reserved.

Choosing a Programming Language

[http://www.skyscrapr.net/]

Chris Britton

January 2008

Summary: This article discusses a practical approach to comparing programming languages and deciding the programming language to use in a new project. (9 printed pages)

Contents

Introduction

Once, I went to a company and found that looking at their system reminded me of archaeology. On the top layer was a Web services application that ran on PCs. The next layer down was a Java server. Scraping that layer away, you found a Unix server that ran Tuxedo with applications that were written in C. Finally, near the bedrock, was a mainframe that ran COBOL. Every time that there was some functionality to add, the question arose: In which layer shall we put it? A key aspect of answering that question was another question: What programming language shall we use? Almost every organization of (as they say in polite company) a certain age has a similar dilemma.

Technical Characteristics

There are plenty of technical characteristics with which to compare programming languages: how many key words, maximum length of identifier, type-checking facilities, polymorphism, overriding, and so on. It is almost infinitely perplexing. Instead, I suggest that you focus on the following comparison criteria, which are centered on outcomes:

- Ease of learning
- Ease of understanding
- Speed of development
- Help with enforcement of correct code
- Performance of compiled code
- Supported platform environments
- Portability
- Fit-for-purpose

Ease of Learning

Clearly, the easier that the programming language is to learn, the quicker that programmers become productive. Java is a lot easier to learn than C++; C is probably even easier. But you learn once and

program for a long time, so that ease of learning is of only limited value.

About the most difficult language to learn that I know is C++. One of the weird things about C++ is that the language is so complex and flexible that you develop your own style and work mostly within a subset of the language. This is strikingly illustrated by the two Microsoft object libraries: MFC (Microsoft Foundation Classes) and ATL (Active Template Library), which have a large overlap in functionality, but look completely different. Learning to be proficient in C++ is best seen as a three-step process: basic understanding, developing your own style, and learning to read someone else's style. Classes and books teach you only step one; but it is only when you have passed step two that you are useful. You will be of more help to others when you have completed step three. It is probably the same for all languages; but the time that is taken in steps two and three is much shorter in languages that are not as profuse in features as C++.

Ease of Understanding

Most code is written once and read many times—usually, to focus on a particular point (for instance, to fix a bug). Thus, it is important that the reader quickly grasp the essence of what's happening. COBOL can usually be read easily; but, because it is verbose, you have to read many lines of code to get anywhere. Old-fashioned COBOL (in contrast to OO COBOL) tended to use PERFORMs, instead of procedure calls, and that means that the logic and the data are miles apart. C does not have this problem; but C can still be hard to understand. This is partly the fault of the language. For instance, I always have to think twice to remember that:

<code>int* ar [20]

is an array of pointers, and not a pointer to an array of integers.

In theory, object-oriented (OO) languages allow you to write more compact code (because of code reuse), and the structure of the objects can allow you to mirror more closely the structure of the problem; thus, in theory, they should be easier to understand. In practice, you might find many abstract classes; and you have to spend time trying to figure out what they are all for, before you can identify all of the parts of the program that are relevant to the topic that you are investigating.

Programmers can make any program hard to understand. Unfortunately, there is a breed of programmers who think that cryptic code is good code—especially, C and C++ programmers. It used to be the case that cryptic code produced fast, compact applications; but compiler optimizers are so good these days that this is no longer an excuse.

Speed of Development

If you look at speed of development in the round, you must consider not only how long it takes you to write code, but also how long it takes you to find a solution to the problem at hand and find the bugs. Factors other than the programming language—for instance, platform facilities, development tools, experience and skill of the programmers, and testing regime—are so significant that it is hard to pin down any difference in development speed that is actually due to use of different programming languages. For instance, a quick calculation will show you that the physical act of typing code makes practically no difference to the speed of development. Verbose, therefore, does not mean unproductive.

There is a programming language called APL (initialism for A Programming Language) that takes compactness to the extreme by using a host of additional graphic symbols, and was best used if you had a specialized keyboard. It was described to me once as a "write-only programming language"; having been written, the code is immediately incomprehensible. Not surprisingly, the language has fallen out of fashion.

Object orientation has probably made good programmers better and bad programmers worse. Usually, with OO programming, there are more ways of tackling a problem than in conventional languages—or, at least, there seems to be. This means more time up front thinking about the design. Once that is done, however, development speed should be faster—mainly, because of the reuse opportunities.

The dream of reusable class libraries from which classes can be extracted and glued together by non-experts never materialized. Creating the classes was no problem. Finding and reusing the classes was the hard part, because you had to find a class that was both a solution to the problem and did not have unwanted side effects, such as using other classes, databases, or system facilities that you did not have or did not want. One of the messages from the agile community is: Only program for reuse, when a reuse opportunity arises. In other words, do not develop classes that you think will be suitable for reuse. Instead, refactor the code to create a reusable class only when you want to reuse it. I find this to be good advice.

One of the simplest forms of reuse is to copy code from one place and paste it in another. The benefits of OO reuse kick in when you want to modify the code; now, you modify it only in one place. In a poorly designed OO program, you find that it is hard to modify the code for one subclass without having to break the functionality of another subclass. In a well-designed OO program, each piece of shared code has a clear task, and making it better benefits all subclasses. Finding the right solution can be hard and is often an iterative process, which is why refactoring is so important.

Help with Enforcement of Correct Code

The ideal programming language should turn logic errors into syntax errors. A powerful means to this end is type checking. Most standard languages, such as Java and COBOL, have good type checking. But some languages have an escape clause. An example is C++, in which you can change the type of a pointer. Figure 1 shows an example.

```
char·a[1000];¤
class·C·{int·x;·int·y;};
C*·p·=·(C*)a·+·50;¤
p->x·=·100;¤
```

Figure 1. Changing the type of a pointer

This example creates a pointer to an object of type **C**, but points it 50 characters into array **a**. Clearly, this feature completely circumvents type checking, and it is considered dangerous. But I have had hardly any trouble by using this feature—possibly, because I know it's dangerous and, so, tread carefully.

Most languages have pitfalls of their own. These are caused by being able to write two expressions that look similar on paper, but have very different effects. For instance, in C and C++:

<code>if (A = B) X();

means that A is assigned the value of B, and X is called if A is not equal to zero; while:

<code>if (A == B) X();

means to call X if A equals B.

Java is safer, because the first expression gives a syntax error.

Performance of Compiled Code

These days, performance is as much an architectural issue as a programming issue. Thinking of performance problems about which I have heard over the last five years, there was one that used a workflow tool for all of its application logic. It used far too many I/Os. There was another that did not upgrade its network when it moved from old-fashioned terminals to a Web interface. And there was another that had a program that would cache hits to searches, but, every now and then, clear its cache and grind to a halt rebuilding it.

I cannot remember a recent case in which bad performance came down to inefficient code production by the compiler. Even with a games program that is processor-bound, the chances are that a large percentage of the processing is going toward drawing the screen. That, in turn, means that performance comes down as much to how you use the graphics card as how the compiler generates code. Of course, huge, number-crunching programs will be an exception; but, even there, only a small part of a large program is performance-critical.

Supported Platform Environments

By platform environment, I mean not only the operating-system facilities, but also the middleware facilities, database facilities, and system-management facilities. Clearly, the more facilities that you have, the more work that has already been done for you. However, there is a downside. Understanding the

platform facilities to the level that you can use them wisely is more difficult, in my opinion, and takes longer than understanding the programming language. Furthermore, I find that I use a facility, make it work, and move on. I do not revisit the subject for months or years. The effect of this is that I am much more dependent on documentation, examples, and research by using the Web than I am with normal programming.

Portability

Most popular languages have been standardized by a nonvendor organization. The aim is twofold: to reduce retraining needs and enhance portability. However, portability has been found to be very difficult. The standards bodies have succeeded in making differences of syntax a relatively minor problem, but that only covers the part of the language that is standardized. Most languages are dependent on hardware constraints in some form, such as defining the maximum value that an integer can take. If code is used such as seen in Figure 1, the dependency between code and platform can be tight and hard to unravel.

Probably, however, the most serious problem with portability is the platform environment. For instance, the problems of moving a mainframe COBOL program to a .NET environment will almost certainly be in the area of changing from—say, a CICS interface to a .NET interface. By far, the most successful example of a popular language that has good portability is Java, which was deliberately designed for portability. It has achieved this by standardizing not only the language, but also the platform environment (J2EE and J2SE).

Fit-for-Purpose

While Java is a good language and is highly portable, it is unsuitable for some purposes, such as some game programming and system programming. Game programming often requires fast access to the screen-display hardware. This can be done by using DirectX, which is available only in C++ or C. The kind of code that Figure 1 illustrates might not be portable, but it is highly useful if you must take a buffer of information and unpick the data from it. You need this if you are writing your own database or network software. However, the majority of programmers are writing business applications in which neither of these issues applies, and Java (or C# or Microsoft Visual Basic) is just fine.

Historically, there have been many good programming languages that got nowhere in the market, simply because they did not have good interfaces to the platform environment. This was probably true of Pascal, for instance, and it helps explain why that never became a popular commercial language.

Decisions, Decisions

So, how do you decide which programming language to use? The first overriding factor is the last one in the preceding list: fit-for-purpose. For most business operational systems, you need a language that has good middleware and database facilities. For specialized work—for instance, writing a hardware driver—you might be forced to use C or C++. For some AI work, perhaps, a language such as Lisp or Prolog might be the only one that gives you access to the facilities that you need.

The second factor is platform choice. If your application must integrate with other applications, you will often find it easier to implement if it is written in the same language as those other applications. For instance, calling a Java method by using Java RMI is easier from another Java program. Calling a .NET application method is easier from another .NET application. Also, if your program must create a batch file for an existing COBOL program, it will be easier and quicker to write the new application in COBOL—especially, if you can use existing COBOL copy libraries.

The third major factor is the existing skills base of your programmers. Training an existing Java programmer to write in C# would probably take a few days, because the languages are similar. What would take much longer would be training your programmers in the .NET environment. For the first few months of their new life as C# programmers, they will be looking things up in the reference manual, asking each other how such-and-such worked, and spending time pondering (and, we hope, properly investigating) the best way to perform a new task. As time progresses, much of this additional effort disappears, and both productivity and quality creep up.

The net result of the last two factors is that choosing a programming language becomes a strategic skills issue for the IT department. You want to concentrate your resources on a few languages. You want to grow skills in languages that you think will have long-term benefits for your organization. However, you do not want to do it so fast that you are left with all of the programmers struggling with a new language.

Conclusion

The three major factors that influence the choice of programming language are the following:

• The language must be fit-for-purpose. Normal business-application development can be done in Java or

COBOL, but specialist applications might need C++ or some other language.

• The choice of platform is critical. It's better to let the platform dictate the programming language, instead of letting the programming language dictate the platform.

• The skills of the programmers. Programmers are much more productive when they are working in a language that they know well, instead of working in a new language or on a new platform.

Critical-Thinking Questions

• What are the programming skills of my department?

• What do we expect to happen to this skills base over the next five years? For instance, are key people reaching retirement age?

• What is our platform and middleware strategy over that time period?

• Do the skills match the strategy?

• What are the programming requirements for maintenances over the next five years?

• Is there—or will there be—a shortfall in programming expertise?

Further Study

There are many books on programming, but I cannot recommend any that discusses programming in general and compares different programming languages. If you are in an IT department that is facing these issues, I suggest that you have the different groups explain the language that they use, the environment, and how they use the facilities, so that decision makers and influencers have an understanding of all of the options. Get them to structure their presentations on the comparison criteria that are discussed in this paper.

Programmers tend to be very dismissive of languages and environments that they do not understand well. This attitude can be overcome in part by some cross-training.

Glossary

Abstract class—A class that is used only to define subclasses and has no objects of its own.

Class—The definition of an object's implementation and interface. A class defines a collection of variables and methods, and one or more interfaces.

Class library—A collection of classes.

Function—Used in C and C++ to mean both operation and method. In mathematics, a function converts input data to output data and has no side effects (that is, updates to variables). In C and C++, functions can have side effects. Function, operation, procedure, and method are often used colloquially as synonyms.

Global variables—Variables that are always present (created when the program starts and destroyed when the program finishes).

Heap—Where free-standing variables are stored. Free-standing variables are created and deleted by program command, instead of being created automatically, as is the case with global and local variables. In some languages (such as Java), all objects go in the heap, and only objects go in the heap.

Interface—The operations and variables that can be used from outside the module or class. Thus, class variables can be public (they are part of the interface) or private (they are not part of the interface, and can be used only by the object's methods). The interface also defines the object's type (see **type**). In some languages (such as Java), it is possible to define a class with multiple interfaces (that are not subtypes of one another) and, therefore, with multiple types.

Local variables—Variables that are local to a method (or block) that are created when the method/block starts and are destroyed when the method/block finishes.

Method—The implementation for an operation (that is, the program logic for the operation). Function, operation, procedure, and method are often used colloquially as synonyms.

Module—A subset of a program that can have its own local data and is accessible to the outside through

one or more interfaces. Modules can often be compiled separately and linked.

Object—Different programming languages have a different concept of object. Usually, it means a free-standing, structured variable (one that can be created and deleted) that is defined by a class.

Operation—A named piece of functionality that is provided by an interface. An operation's signature is the name of the operation, the name and type of all its parameters, and, perhaps, the output type of the data that is produced by the operation. Function, operation, procedure, and method are often used colloquially as synonyms.

Overriding—Operations that have the same name, but different methods. Polymorphism is one form of overriding; but, in languages such as C++, you can override an operation by defining an operation that has the same name but different parameters.

Parameters—The input and output variables for an operation.

Polymorphism—Overriding by defining a different method in a subclass for an operation that is defined in its superclass.

Procedure—A word that is commonly used in non-OO programming languages for the combination of operation and method. Function, operation, procedure, and method are often used colloquially as synonyms.

Refactor—A term that was first coined by the agile-programming community. It means making small changes to the implementation without changing the functionality, and it is used to improve the code quality and ready a program for the further development of a new feature.

Reuse—The act of taking existing code and variable definitions, and using them in the implementation of some new functionality. Reuse can be of many different forms—ranging from copying and pasting code examples, to creating a new subclass from an existing class to reuse the class's existing functionality.

Stack—A last in/first out (LIFO) storage mechanism. Local variables are stored in a stack, as building a stack is a natural consequence of methods calling other methods.

Subclass—A class that is defined by extending an existing class, instead of defining it anew. A subclass also normally extends the supertype's interface and, therefore, defines a subtype of the class's interface. See the definition of **type**.

Subtype—Type S can be defined as a subtype of type T—meaning that, whenever a variable of type T is called for, a variable of type S can be substituted for it.

Superclass—The opposite of subclass.

Type—A property of objects and variables that is used in type checking. Informally, a type defines how an object or variable can be used.

Type checking—Ensuring that the variables that are passed to an operation through a parameter or used in an assignment statement have compatible type. Compatible type means that the type of input variable is the same as—or a subtype of—the type that is defined for the target variable (that is, the parameter or assigned variable).

Variable—Data with a type and (with the exception of temporary variables output from an expression) a name.

About the author

Chris Britton has programmed in Algol, COBOL, Pascal, C, C++, and Java, but primarily in Algol and C++. Recently, he has designed and written the WeaverBird generic modeling program in C++, which included developing his own programming language, LOOM (Language for Object-Oriented Modeling). Chris also does architecture consultancy. With Peter Bye, he wrote *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems*, Second Edition (Boston: Addison-Wesley, 2004).

This article was published in Skyscrapr, an online resource provided by Microsoft. To learn more about architecture and the architectural perspective, please visit skyscrapr.net [http://www.skyscrapr.net/] .