# Scripting: Higher Level Programming for the 21st Century

**John K. Ousterhout**

Sun Microsystems Laboratories
901 San Antonio Rd., MS UMTV29-232
Palo Alto, CA 94303-4900
`john.ousterhout@eng.sun.com`

(This article will appear in *IEEE Computer* magazine)

**Abstract**

Scripting languages such as Perl and Tcl represent a very different style of programming than system programming languages such as C or Java™. Scripting languages are designed for "gluing" applications; they use typeless approaches to achieve a higher level of programming and more rapid application development than system programming languages. Increases in computer speed and changes in the application mix are making scripting languages more and more important for applications of the future.

**Keywords:** component frameworks, object-oriented programming, scripting, strong typing, system programming.

## 1 Introduction

For the last fifteen years a fundamental change has been occurring in the way people write computer programs. The change is a transition from system programming languages such as C or C++ to scripting languages such as Perl or Tcl. Although many people are participating in the change, few people realize that it is occurring and even fewer people know why it is happening. This article is an opinion piece that explains why scripting languages will handle many of the programming tasks of the next century better than system programming languages.

Scripting languages are designed for different tasks than system programming languages, and this leads to fundamental differences in the languages. System programming languages were designed for building data structures and algorithms from scratch, starting from the most primitive computer elements such as words of memory. In contrast, scripting languages are designed for gluing: they assume the existence of a set of powerful components and are intended primarily for connecting components together. System programming languages are strongly typed to help manage complexity, while scripting languages are typeless to simplify connections between components and provide rapid application development.

Scripting languages and system programming languages are complementary, and most major computing platforms since the 1960's have provided both kinds of languages. The languages are typically used together in component frameworks, where components are created with system programming languages and glued together with scripting languages. However, several recent trends, such as faster machines, better scripting languages, the increasing importance of graphical user interfaces and component architectures, and the growth of the Internet, have greatly increased the applicability of scripting languages. These trends will continue over the next decade, with more and more new applications written entirely in scripting languages and system programming languages used primarily for creating components.

## 2 System programming languages

In order to understand the differences between scripting languages and system programming languages, it is important to understand how system programming languages evolved. System programming languages were introduced as an alternative to assembly languages. In assembly languages, virtually every aspect of the machine is reflected in the program. Each statement represents a single machine instruction and programmers must deal with low-level details such as register allocation and procedure calling sequences. As a result, it is difficult to write and maintain large programs in assembly language.

By the late 1950's higher level languages such as Lisp, Fortran, and Algol began to appear. In these languages statements no longer correspond exactly to machine instructions; a compiler translates each statement in the source program into a sequence of binary instructions. Over time a series of *system programming languages* evolved from Algol, including such languages as PL/1, Pascal, C, C++, and Java. System programming languages are less efficient then assembly languages but they allow applications to be developed much more quickly. As a result, they have almost completely replaced assembly languages for the development of large applications.

System programming languages differ from assembly languages in two ways: they are higher level and they are strongly typed. The term "higher level" means that many details are handled automatically so that programmers can write less code to get the same job done. For example:

- Register allocation is handled by the compiler so that programmers need not write code to move information between registers and memory.
- Procedure calling sequences are generated automatically: programmers need not worry about moving arguments to and from the call stack.
- Programmers can use simple keywords such as `while` and `if` for control structures; the compiler generates all the detailed instructions to implement the control structures.

On average, each line of code in a system programming language translates to about five machine instructions, compared to one instruction per line in assembly language (in an informal analysis of eight C files written by five different people, I found that the ratio ranged from about 3 to 7 instructions per line[7]; in a study of numerous languages Capers

Jones found that for a given task, assembly languages require about 3-6 times as many lines of code as system programming languages[3]). Programmers can write roughly the same number of lines of code per year regardless of language[1], so system programming languages allow applications to be written much more quickly than assembly language.

The second difference between assembly language and system programming languages is typing. I use the term “typing” to refer to the degree to which the meaning of information is specified in advance of its use. In a strongly typed language the programmer declares how each piece of information will be used and the language prevents the information from being used in any other way. In a weakly typed language there are no *a priori* restrictions on how information can be used: the meaning of information is determined solely by the way it is used, not by any initial promises.[1]

Modern computers are fundamentally typeless: any word in memory can hold any kind of value, such as an integer, a floating-point number, a pointer, or an instruction. The meaning of a value is determined by how it is used: if the program counter points at a word of memory then it is treated as an instruction; if a word is referenced by an integer add instruction then it is treated as an integer; and so on. The same word can be used in different ways at different times.

In contrast, today’s system programming languages are strongly typed. For example:

- Each variable in a system programming language must be declared with a particular type such as integer or pointer to string, and it must be used in ways that are appropriate for the type.
- Data and code are totally segregated: it is difficult or impossible to create new code on the fly.
- Variables can be collected into structures or objects with well-defined substructure and procedures or methods to manipulate them; an object of one type cannot be used where an object of a different type is expected.

Typing has several advantages. First, it makes large programs more manageable by clarifying how things are used and differentiating between things that must be treated differently. Second, compilers can use type information to detect certain kinds of errors, such as an attempt to use a floating-point value as a pointer. Third, typing improves performance by allowing compilers to generate specialized code. For example, if a compiler knows that a variable always holds an integer value then it can generate integer instructions to manipulate the variable; if the compiler doesn’t know the type of a variable then it must generate additional instructions to check the variable’s type at runtime.

To summarize, system programming languages are designed to handle the same tasks as assembly languages, namely creating applications from scratch. System programming lan-

---

[1] A more precise characterization would use the term “static typing” where I say “strong typing” and “dynamic typing with automatic conversion” for scripting languages that I describe as weakly typed or untyped. I use the term “typing” in a general sense to describe the degree to which the usage of data is restricted in advance.

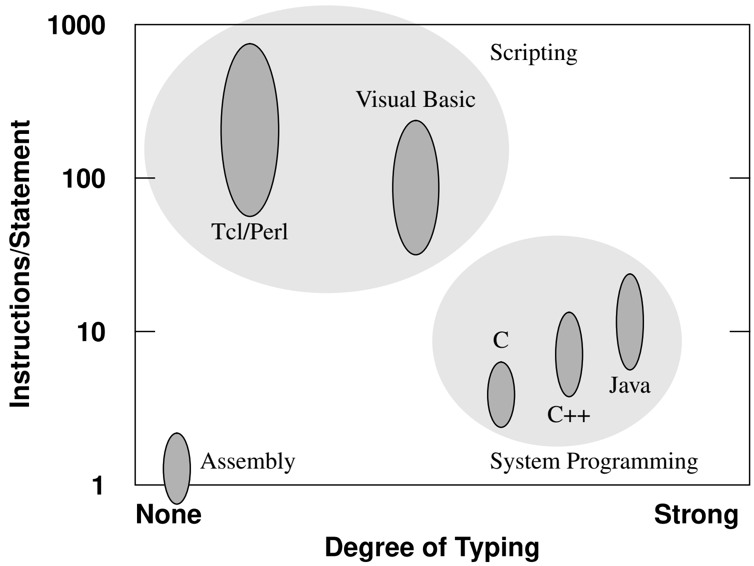


**Figure 1.** A comparison of various programming languages based on their level (higher level languages execute more machine instructions for each language statement) and their degree of typing. System programming languages like C tend to be strongly typed and medium level (5-10 instructions/statement). Scripting languages like Tcl tend to be weakly typed and very high level (100-1000 instructions/statement).

guages are higher level and much more strongly typed than assembly languages. This allows applications to be created more rapidly and managed more easily with only a slight loss in performance. See Figure 1 for a graphical comparison of assembly language and several system programming languages.

# 3 Scripting languages

Scripting languages such as Perl[9], Python[4], Rexx[6], Tcl[8], Visual Basic, and the Unix shells represent a very different style of programming than system programming languages. Scripting languages assume that there already exists a collection of useful *components* written in other languages. Scripting languages aren't intended for writing applications from scratch; they are intended primarily for plugging together components. For example, Tcl and Visual Basic can be used to arrange collections of user interface controls on the screen, and Unix shell scripts are used to assemble filter programs into pipelines. Scripting languages are often used to extend the features of components but they are rarely used for complex algorithms and data structures; features like these are usually provided by the components. Scripting languages are sometimes referred to as *glue languages* or *system integration languages*.

In order to simplify the task of connecting components, scripting languages tend to be typeless: all things look and behave the same so that they are interchangeable. For example, in Tcl or Visual Basic a variable can hold a string one moment and an integer the next. Code and data are often interchangeable, so that a program can write another program and then execute it on the fly. Scripting languages are often string-oriented, since this provides a uniform representation for many different things.

A typeless language makes it much easier to hook together components. There are no *a priori* restrictions on how things can be used, and all components and values are represented in a uniform fashion. Thus any component or value can be used in any situation; components designed for one purpose can be used for totally different purposes never foreseen by the designer. For example, in the Unix shells, all filter programs read a stream of bytes from an input and write a string of bytes to an output; any two programs can be connected together by attaching the output of one program to the input of the other. The following shell command stacks three filters together to count the number of lines in the selection that contain the word "scripting":

```
select | grep scripting | wc
```

The `select` program reads the text that is currently selected on the display and prints it on its output; the `grep` program reads its input and prints on its output the lines containing "scripting"; the `wc` program counts the number of lines on its input. Each of these programs can be used in numerous other situations to perform different tasks.

The strongly typed nature of system programming languages discourages reuse. Typing encourages programmers to create a variety of incompatible interfaces ("interfaces are good; more interfaces are better"). Each interface requires objects of specific types and the compiler prevents any other types of objects from being used with the interface, even if that would be useful. In order to use a new object with an existing interface, conversion code must be written to translate between the type of the object and the type expected by the interface. This in turn requires recompiling part or all of the application, which isn't possible in the common case where the application is distributed in binary form.

To see the advantages of a typeless language, consider the following Tcl command:

```
button .b -text Hello! -font {Times 16} -command {puts hello}
```

This command creates a new button control that displays a text string in a 16-point Times font and prints a short message when the user clicks on the control. It mixes six different types of things in a single statement: a command name (`button`), a button control (`.b`), property names (`-text`, `-font`, and `-command`), simple strings (`Hello!` and `hello`), a font name (`Times 16`) that includes a typeface name (`Times`) and a size in points (`16`), and a Tcl script (`puts hello`). Tcl represents all of these things uniformly with strings. In this example the properties may be specified in any order and unspecified properties are given default values; more than 20 properties were left unspecified in the example.

The same example requires 7 lines of code in two methods when implemented in Java. With C++ and Microsoft Foundation Classes, it requires about 25 lines of code in three procedures (see [7] for the code for these examples). Just setting the font requires several lines of code in Microsoft Foundation Classes:

```
CFont *fontPtr = new CFont();
fontPtr->CreateFont(16, 0, 0,0,700, 0, 0, 0, ANSI_CHARSET,
    OUT_DEFAULT_PRECIS,CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
```

```
        DEFAULT_PITCH|FF_DONTCARE, "Times New Roman");
    buttonPtr->SetFont(fontPtr);
```

Much of this code is a consequence of the strong typing. In order to set the font of a button, its `SetFont` method must be invoked, but this method must be passed a pointer to a `CFont` object. This in turn requires a new object to be declared and initialized. In order to initialize the `CFont` object its `CreateFont` method must be invoked, but `CreateFont` has a rigid interface that requires 14 different arguments to be specified. In Tcl, the essential characteristics of the font (typeface Times, size 16 points) can be used immediately with no declarations or conversions. Furthermore, Tcl allows the behavior for the button to be included directly in the command that creates the button, while C++ and Java require it to be placed in a separately declared method.

(In practice, a trivial example like this would probably be handled with a graphical development environment that hides the complexity of the underlying language: the user enters property values in a form and the development environment outputs the code. However, in more complex situations such as conditional assignment of property values or interfaces generated programmatically, the developer must write code in the underlying language.)

It might seem that the typeless nature of scripting languages could allow errors to go undetected, but in practice scripting languages are just as safe as system programming languages. For example, an error will occur if the font size specified for the button example above is a non-integer string such as `xyz`. The difference is that scripting languages do their error checking at the last possible moment, when a value is used. Strong typing allows errors to be detected at compile-time, so the cost of run-time checks is avoided. However, the price to be paid for this efficiency is restrictions on how information can be used: this results in more code and less flexible programs.

Another key difference between scripting languages and system programming languages is that scripting languages are usually interpreted whereas system programming languages are usually compiled. Interpreted languages provide rapid turnaround during development by eliminating compile times. Interpreters also make applications more flexible by allowing users to program the applications at run-time. For example, many synthesis and analysis tools for integrated circuits include a Tcl interpreter; users of the programs write Tcl scripts to specify their designs and control the operation of the tools. Interpreters also allow powerful effects to be achieved by generating code on the fly. For example, a Tcl-based Web browser can parse a Web page by translating the HTML for the page into a Tcl script using a few regular expression substitutions. It then executes the Tcl script to render the page on the screen.

Scripting languages are less efficient than system programming languages, in part because they use interpreters instead of compilers but also because their basic components are chosen for power and ease of use rather than an efficient mapping onto the underlying hardware. For example, scripting languages often use variable-length strings in situations where a system programming language would use a binary value that fits in a single machine word, and scripting languages often use hash tables where system programming languages use indexed arrays.

Fortunately, the performance of a scripting language isn't usually a major issue. Applications for scripting languages are generally smaller than applications for system programming languages, and the performance of a scripting application tends to be dominated by the performance of the components, which are typically implemented in a system programming language.

Scripting languages are higher level than system programming languages, in the sense that a single statement does more work on average. A typical statement in a scripting language executes hundreds or thousands of machine instructions, whereas a typical statement in a system programming language executes about five machine instructions (see Figure 1). Part of this difference is because scripting languages use interpreters, which are less efficient than the compiled code for system programming languages. But much of the difference is because the primitive operations in scripting languages have greater functionality. For example, in Perl it is about as easy to invoke a regular expression substitution as it is to invoke an integer addition. In Tcl, a variable can have traces associated with it so that setting the variable causes side effects; for example, a trace might be used to keep the variable's value updated continuously on the screen.

Because of the features described above, scripting languages allow very rapid development for applications that are gluing-oriented. Table 1 provides anecdotal support for this claim. It describes several applications that were implemented in a system programming language and then reimplemented in a scripting language, or vice versa. In every case the scripting version required less code and development time than the system programming version; the difference varied from a factor of 2 to a factor of 60. Scripting languages provided less benefit when they were used for the first implementation; this suggests that any reimplementation benefits substantially from the experiences of the first implementation and that the true difference between scripting and system programming is more like a factor of 5-10x than the extreme points of the table. The benefits of scripting also depend on the application. In the last example of the table the GUI part of the application is gluing-oriented but the simulator part isn't; this may explain why the application benefited less from scripting than other applications.

To summarize, scripting languages are designed for gluing applications. They provide a higher level of programming than assembly or system programming languages, much weaker typing than system programming languages, and an interpreted development environment. Scripting languages sacrifice execution speed to improve development speed.

## 4 Different tools for different tasks

A scripting language is not a replacement for a system programming language or vice versa. Each is suited to a different set of tasks. For gluing and system integration, applications can be developed 5-10x faster with a scripting language; system programming languages will require large amounts of boilerplate and conversion code to connect the pieces, whereas this can be done directly with a scripting language. For complex algorithms and data structures, the strong typing of a system programming language makes programs easier to manage. Where execution speed is key, a system programming lan-

| Application (Contributor) | Comparison | Code Ratio | Effort Ratio | Comments |
|---|---|---|---|---|
| Database application (Ken Corey) | C++ version: 2 months<br>Tcl version:1 day | | 60 | C++ version implemented first; Tcl version had more functionality. |
| Computer system test and installation (Andy Belsey) | C test application: 272 Klines, 120 months.<br>C FIS application: 90 Klines, 60 months.<br>Tcl/Perl version: 7.7K lines, 8 months | 47 | 22 | C version implemented first. Tcl/Perl version replaced both C applications. |
| Database library (Ken Corey) | C++ version: 2-3 months<br>Tcl version: 1 week | | 8-12 | C++ version implemented first. |
| Security scanner (Jim Graham) | C version: 3000 lines<br>Tcl version: 300 lines | 10 | | C version implemented first. Tcl version had more functionality. |
| Display oil well production curves (Dan Schenck) | C version: 3 months<br>Tcl version: 2 weeks | | 6 | Tcl version implemented first. |
| Query dispatcher (Paul Healy) | C version: 1200 lines, 4-8 weeks<br>Tcl version: 500 lines, 1 week | 2.5 | 4-8 | C version implemented first, uncommented. Tcl version had comments, more functionality. |
| Spreadsheet tool | C version: 1460 lines<br>Tcl version: 380 lines | 4 | | Tcl version implemented first. |
| Simulator and GUI (Randy Wang) | Java version: 3400 lines, 3-4 weeks.<br>Tcl version: 1600 lines, < 1 week. | 2 | 3-4 | Tcl version had 10-20% more functionality, was implemented first. |

**Table 1.** Each row of the table describes an application that was implemented twice, once with a system programming language such as C or Java and once with a scripting language such as Tcl. The **Code Ratio** column gives the ratio of lines of code for the two implementations (>1 means the system programming language required more lines); the **Effort Ratio** column gives the ratio of development times. In most cases the two versions were implemented by different people. The information in the table was provided by various Tcl developers in response to an article posted on the comp.lang.tcl newsgroup; see [7] for details.

guage can often run 10-20x faster than a scripting language because it makes fewer run-time checks.

In deciding whether to use a scripting language or a system programming language for a particular task, consider the following questions:

- Is the application's main task to connect together pre-existing components?
- Will the application manipulate a variety of different kinds of things?
- Does the application include a graphical user interface?
- Does the application do a lot of string manipulation?
- Will the application's functions evolve rapidly over time?

- Does the application need to be extensible?

“Yes” answers to these questions suggest that a scripting language will work well for the application. On the other hand, “yes” answers to the following questions suggest that an application is better suited to a system programming language:

- Does the application implement complex algorithms or data structures?
- Does the application manipulate large datasets (e.g. all the pixels in an image) so that execution speed is critical?
- Are the application’s functions well-defined and changing slowly?

Most of the major computing platforms over the last 30 years have provided both system programming and scripting languages. For example, one of the first scripting languages, albeit a crude one, was JCL (Job Control Language), which was used to sequence job steps in OS/360. The individual job steps were written in PL/1, Fortran, or assembler language, which were the system programming languages of the day. In the Unix machines of the 1980’s, C was used for system programming and shell programs such as `sh` and `csh` for scripting. In the PC world of the 1990’s, C and C++ are used for system programming and Visual Basic for scripting. In the Internet world that is taking shape now, Java is used for system programming and languages such as JavaScript, Perl, and Tcl are used for scripting.

Scripting and system programming are symbiotic. Used together, they produce programming environments of exceptional power: system programming languages are used to create exciting components which can then be assembled using scripting languages. For example, much of the attraction of Visual Basic is that system programmers can write ActiveX components in C and less sophisticated programmers can then use the components in Visual Basic applications. In Unix it is easy to write shell scripts that invoke applications written in C. One of the reasons for the popularity of Tcl is the ability to extend the language by writing C code that implements new commands.

# 5 Scripting is on the rise

Scripting languages have existed for a long time, but in recent years several factors have combined to increase their importance. The most important factor is a shift in the application mix towards gluing applications. Three examples of this shift are graphical user interfaces, the Internet, and component frameworks.

Graphical user interfaces (GUIs) first began to appear in the early 1980’s and became widespread by the end of the decade; GUIs now account for half or more of the total effort in many programming projects. GUIs are fundamentally gluing applications: the goal is not to create new functionality, but to make connections between a collection of graphical controls and the internal functions of the application. I am not aware of any rapid-development environments for GUIs based on a system programming language. Whether the environment is Windows, Macintosh Toolbox, or Unix Motif, GUI toolkits based on languages like C or C++ have proven to be hard to learn, clumsy to use, and inflexible in the results they produce. Some of these systems have very nice graphical tools for designing screen layouts that hide the underlying language, but things become difficult as soon as the designer has to write code, for example to provide the behaviors for the interface elements. All of the best rapid-development GUI environments are based on scripting languages: Visual Basic, HyperCard, and Tcl/Tk. Thus scripting languages have risen in popularity as the importance of GUIs has increased.

The growth of the Internet has also popularized scripting languages. The Internet is nothing more than a gluing tool. It doesn't create any new computations or data; it simply makes a huge number of existing things easily accessible. The ideal language for most Internet programming tasks is one that makes it possible for all the connected components to work together, i.e. a scripting language. For example, Perl has become popular for writing CGI scripts and JavaScript is popular for scripting in Web pages.

The third example of scripting-oriented applications is component frameworks such as ActiveX, OpenDoc, and JavaBeans. Although system programming languages work well for creating components, the task of assembling components into applications is better suited to scripting. Without a good scripting language to manipulate the components, much of the power of a component framework is lost. This may explain in part why component frameworks have been more successful on PCs (where Visual Basic provides a convenient scripting tool) than on other platforms such as Unix/CORBA where scripting is not included in the component framework.

Another reason for the increasing popularity of scripting languages is improvements in scripting technology. Modern scripting languages such as Tcl and Perl are a far cry from early scripting languages such as JCL. For example, JCL didn't even provide basic iteration and early Unix shells didn't support procedures. Scripting technology is still relatively immature even today. For example, Visual Basic isn't really a scripting language; it was originally implemented as a simple system programming language, then modified to make it more suitable for scripting. Future scripting languages will be even better than those available today.

Scripting technology has also benefited from the ever-increasing speed of computer hardware. It used to be that the only way to get acceptable performance in an application of any complexity was to use a system programming language. In some cases even system programming languages weren't efficient enough, so the applications had to be written in assembler. However, machines today are 100-500 times faster than the machines of 1980 and they continue to double in performance every 18 months. Today, many applications can be implemented in an interpreted language and still have excellent performance; for example, a Tcl script can manipulate collections with several thousand objects and still provide good interactive response. As computers get faster, scripting will become attractive for larger and larger applications.

One final reason for the increasing use of scripting languages is a change in the programmer community. Twenty years ago most programmers were sophisticated programmers working on large projects. Programmers of that era expected to spend several months to master a language and its programming environment, and system programming languages were designed for such programmers. However, since the arrival of the personal computer, more and more casual programmers have joined the programmer community. For these people, programming is not their main job function; it is a tool they use occasionally to help with their main job. Examples of casual programming are simple database queries or macros for a spreadsheet. Casual programmers are not willing to spend months learning a system programming language, but they can often learn enough about a scripting language in a few hours to write useful programs. Scripting languages are easier to learn because they have simpler syntax than system programming languages and because they omit complex features like objects and threads. For example, compare Visual Basic with Visual C++; few casual programmers would attempt to use Visual C++, but many have been able to build useful applications with Visual Basic.

Even today the number of applications written in scripting languages is much greater than the number of applications written in system programming languages. On Unix systems there are many more shell scripts than C programs, and under Windows there are many more Visual Basic programmers and applications than C or C++. Of course, most of the largest and most widely used applications are written in system programming languages, so a comparison based on total lines of code or number of installed copies may still favor system programming languages. Nonetheless, scripting languages are already a major force in application development and their market share will increase in the future.

# 6 The role of objects

Scripting languages have been mostly overlooked by experts in programming languages and software engineering. Instead, they have focused their attention on object-oriented system programming languages such as C++ and Java. Object-oriented programming is widely believed to represent the next major step in the evolution of programming languages. Object-oriented features such as strong typing and inheritance are often claimed to reduce development time, increase software reuse, and solve many other problems including those addressed by scripting languages.

How much benefit has object-oriented programming actually provided? Unfortunately I haven't seen enough quantitative data to answer this question definitively. In my opinion objects provide only a modest benefit: perhaps a 20-30% improvement in productivity but certainly not a factor of two, let alone a factor of 10. C++ now seems to be reviled as much as it is loved, and some language experts are beginning to speak out against object-oriented programming [2]. The rest of this section explains why objects don't improve productivity in the dramatic way that scripting does, and it argues that the benefits of object-oriented programming can be achieved in scripting languages.

The reason why object-oriented programming doesn't provide a large improvement in productivity is that it doesn't raise the level of programming or encourage reuse. In an object-oriented language such as C++ programmers still work with small basic units that must be described and manipulated in great detail. In principle, powerful library packages could be developed, and if these libraries were used extensively they could raise the level of programming. However, not many such libraries have come into existence. The strong typing of most object-oriented languages encourages narrowly defined packages that are hard to reuse. Each package requires objects of a specific type; if two packages are to work together, conversion code must be written to translate between the types required by the packages.

Another problem with object-oriented languages is their emphasis on inheritance. Implementation inheritance, where one class borrows code that was written for another class, is a bad idea that makes software harder to manage and reuse. It binds the implementations of classes together so that neither class can be understood without the other: a subclass cannot be understood without knowing how the inherited methods are implemented in its superclass, and a superclass cannot be understood without knowing how its methods are inherited in subclasses. In a complex class hierarchy, no individual class can be understood without understanding all the other classes in the hierarchy. Even worse, a class cannot be separated from its hierarchy for reuse. Multiple inheritance makes these problems even worse. Implementation inheritance causes the same intertwining and brittleness that have been observed when `goto` statements are overused. As a result, object-oriented systems often suffer from complexity and lack of reuse.

Scripting languages, on the other hand, have actually generated significant software reuse. They use a model where interesting components are built in a system programming language and then glued together into applications using the scripting language. This division of labor provides a natural framework for reusability. Components are designed to be reusable, and there are well-defined interfaces between components and scripts that make it easy to use components. For example, in Tcl the components are custom commands implemented in C; they look just like the builtin commands so they are easy to invoke in Tcl scripts. In Visual Basic the components are ActiveX extensions, which can be used by dragging them from a palette onto a form.

Nonetheless, object oriented programming does provide at least two useful features. The first is encapsulation: objects combine together data and code in a way that hides implementation details. This makes it easier to manage large systems. The second useful feature is interface inheritance, which refers to classes that provide the same methods and APIs even though they have different implementations. This makes the classes interchangeable, which encourages reuse.

Fortunately, the benefits of objects can be achieved in scripting languages as well as system programming languages and virtually all scripting languages have some support for object-oriented programming. For example, Python is an object-oriented scripting language, Perl version 5 includes support for objects, Object Rexx is an object-oriented version of Rexx, and Incr Tcl is an object-oriented extension to Tcl. One difference is that objects in scripting languages tend to be typeless, while objects in system programming languages tend to be strongly typed.

## 7 Other languages

This article is not intended as a complete characterization of all programming languages. There are many other attributes of programming languages besides strength of typing and the level of programming, and there are many interesting languages that can't be characterized cleanly as a system programming language or a scripting language. For example, the Lisp family of languages lies somewhere between scripting and system programming, with some of the attributes of each. Lisp pioneered concepts such as interpretation and dynamic typing that are now common in scripting languages, as well as automatic storage management and integrated development environments, which are now used in both scripting and system programming languages.

## 8 Conclusion

Scripting languages represent a different set of tradeoffs than system programming languages. They give up execution speed and strength of typing relative to system programming languages but provide significantly higher programmer productivity and software reuse. This tradeoff makes more and more sense as computers become faster and cheaper in comparison to programmers. System programming languages are well suited to building components where the complexity is in the data structures and algorithms, while scripting languages are well suited for gluing applications where the complexity is in the connections. Gluing tasks are becoming more and more prevalent, so scripting will become an even more important programming paradigm in the next century than it is today.

I hope that this article will impact the computing community in three ways:

- I hope that programmers will consider the differences between scripting and system programming when starting new projects and choose the most powerful tool for each task.
- I hope that designers of component frameworks will recognize the importance of script ing and ensure that their frameworks include not just facilities for creating components but also facilities for gluing them together.
- I hope that the programming language research community will shift some of its attention to scripting languages and help develop even more powerful scripting languages for the future. Raising the level of programming should be the single most important goal for language designers, since it has the greatest effect on programmer productivity; it is not clear that strong typing contributes to this goal.

## 9 Acknowledgments

This article has benefited from many people's comments, including Joel Bartlett, Bill Eldridge, Jeffrey Haemer, Mark Harrison, Paul McJones, David Patterson, Stephen Uhler, Hank Walker, Chris Wright, the *IEEE Computer* referees, and dozens of others who par-

ticipated in a heated net-news discussion of an early draft of the article. Colin Stevens wrote the MFC version of the button example and Stephen Uhler wrote the Java version.

## 10 References


[1] B. Boehm, *Software Engineering Economics*, Prentice-Hall, ISBN 0-138-22122-7, 1981.

[2] S. Johnson, *Objecting To Objects*, Invited Talk, USENIX Technical Conference, San Francisco, CA, January 1994.

[3] C. Jones, "Programming Languages Table, Release 8.2", March 1996, `http://www.spr.com/library/0langtbl.htm`.

[4] M. Lutz, *Programming Python*, O'Reilly, ISBN 1-56592-197-6, 1996.

[5] Netscape Inc., "JavaScript in Navigator 3.0", `http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/atlas.html#taint_dg`.

[6] R. O'Hara and D. Gomberg, *Modern Programming Using REXX*, Prentice Hall, ISBN 0-13-597329-5, 1988.

[7] J. Ousterhout, *Additional Information for Scripting White Paper*, `http://www.sunlabs.com/people/john.ousterhout/scriptextra.html`.

[8] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, ISBN 0-201-63337-X, 1994.

[9] L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl*, Second Edition, O'Reilly and Associates, ISBN 1-56592-149-6, 1996.