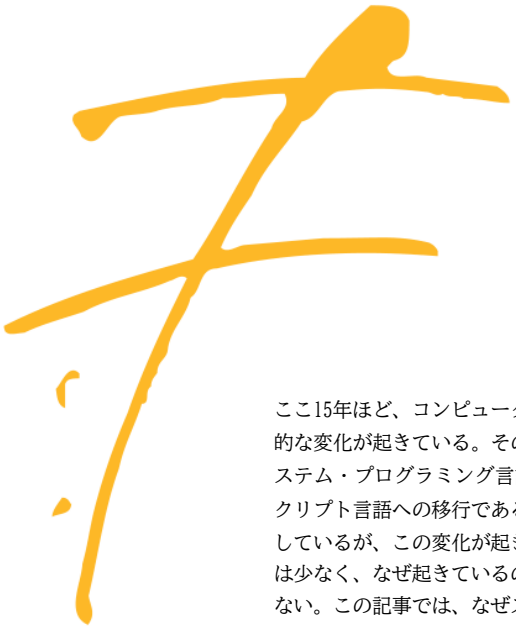
Cybersquare

John K. Ousterhout
Sun Microsystems Laboratories

Scripting: Higher-Level Programming for the 21st Century



Increases in computer speed and changes in the application mix are making scripting languages more and more important for the applications of the future. Scripting languages differ from system programming languages in that they are designed for "gluing" applications together. They use typeless approaches to achieve a higher level of programming and more rapid application development than system programming languages.

ここ15年ほど、コンピュータ・プログラムの書き方に根本的な変化が起きている。その変化とは、CやC++のようなシステム・プログラミング言語から、PerlやTclのようなスクリプト言語への移行である。多くの人がこの変化に参加しているが、この変化が起きていることに気づいている人は少なく、なぜ起きているのかを知っている人はさらに少ない。この記事では、なぜスクリプト言語がシステム・プログラミング言語よりも次の世紀のプログラミング・タスクの多くをうまく処理できるようになるのかを説明する。

スクリプト言語はシステム・プログラミング言語とは異なるタスクのために設計されており、これが言語の根本的な違いにつながっている。システム・プログラミング言語は、データ構造やアルゴリズムをゼロから構築するために設計された。これとは対照的に、スクリプト言語はグルーイングするために設計されている： 強力なコンポーネントの集合の存在を前提とし、コンポーネントを接続することを主な目的としている。システム・プログラミング言語は複雑さを管理するために強く型付けされているが、スクリプト言語はコンポーネント間の接続を単純化し、迅速なアプリケーション開発を実現するために型付けされていない。

スクリプト言語とシステム・プログラミング言語は相補的であり、1960年代以降の主要なコンピューティング・プラットフォームのほとんどには、この2種類の言語が含まれている。これらの言語は通常、コンポーネントフレームワークで併用され、コンポーネントはシステムプログラミング言語で作成され、スクリプト言語で接着される。

しかし、マシンの高速化、より優れたスクリプト言語、グラフィカル・ユーザー・インターフェイス（GUI）やコンポーネント・アーキテクチャの重要性の増大、インターネットの成長など、最近のいくつかの傾向によって、スクリプト言語の適用範囲は大きく広がっている。こうした傾向は今後10年間も続き、スクリプト言語だけで書かれた新しいアプリケーションや、主にコンポーネントの作成に使われるシステム・プログラミング言語がますます増えていくだろう。

システム・プログラミング言語

スクリプト言語とシステム・プログラミング言語の違いを理解するには、システム・プログラミング言語がどのように発展してきたかを理解することが重要である。システム・プログラミング言語は、アセンブリ言語に代わるものとして導入された。アセンブリ言語では、マシンのほぼすべての側面がプログラムに反映される。各ステートメントは1つのマシン命令を表し、プログラマーはレジスタの割り当てやプロシージャ呼び出しシーケンスといった低レベルの詳細を扱わなければならない。その結果、アセンブリ言語で大規模なプログラムを書いたり保守したりするのは難しい。

1950年代後半になると、Lisp、Fortran、Algolといった高級言語が登場し始めた。これらの言語では、ステートメントはもはやマシン命令には正確に対応しない。

0018-9162/98/$10.00 © 1998 IEEE

スクリプト言語は、有用なコンポーネントのコレクションが他の言語ですでに存在していることを前提としている。ゼロからアプリケーションを書くためではなく、むしろコンポーネントを組み合わせるためのものである。

ソースプログラム中の#mentをバイナリ命令のシーケンスに変換する。時間の経過とともに、PL／1、Pascal、C、C++、Javaなど、一連のシステム・プログラミング言語がアルゴルから発展した。システム・プログラミング言語は、アセンブリ言語よりも効率は悪いが、アプリケーションをより迅速に開発することができる。その結果、システム・プログラミング言語は、大規模なアプリケーションの開発において、アセンブリ言語にほぼ完全に取って代わった。

高水準言語

システムプログラミング言語がアセンブリ言語と異なる点は2つある。高水準」という言葉は、多くの詳細が自動的に処理されるため、プログラマーは同じ仕事をするためのコードを書くことができるということを意味する。例えば

less

- レジスタの割り当てはコンパイラによって処理されるため、プログラマは レジスタとメモリの間で情報を移動させるコードを書く必要がない。
- プログラマーは、引数の出し入れを気にする必要がない。
- プログラマーは、whileやifのような単純なキーワードを制御構造に使うことができ、 制御構造を実装するための詳細な命令はすべてコンパイラが生成する。

システム・プログラミング言語のコード1行は、平均して約5つのマシン命令に変換される。(5人の異なる人が書いた8つのCファイルを非公式に分析したところ、その比率は1行あたり3命令から7命令の範囲でした。[1] 多数の言語を調査した結果、Capers Jones氏は、あるタスクに対して、アセンブリ言語はシステム・プログラミング言語の3倍から6倍のコード行数を必要とすることを発見しました。[2] プログラマは、言語に関係なく、1年間にほぼ同じ数のコード行数を書くことができます。[3] したがって、システム・プログラミング言語は、アセンブリ言語よりもはるかに速くアプリケーションを書くことができます。

Typing

アセンブリ言語とシステム・プログラミング言語の2つ目の違いは、型付けである。型付けという用語は、情報の意味がその使用前に特定される度合いを意味する。強く型付けされた言語では、プログラマーは各情報がどのように使用されるかを宣言し、言語はその情報が他の方法で使用されるのを防ぐ。弱く型付けされた言語では、情報の使用方法に関する先験的な制限はない。

情報の意味 は、最初の約束ではなく、情報の使用方法によってのみ決定される。

現代のコンピュータは基本的に型がない。メモリ上のどの単語も、整数、浮動小数点数、ポインタ、命令など、あらゆる種類の値を保持できる。値の意味は、それがどのように使われるかによって決まる。プログラム・カウンタがメモリのワードを指していれば、それは命令として扱われ、ワードが整数加算命令によって参照されていれば、それは整数として扱われる、といった具合である。同じワードを異なるタイミングで異なる方法で使用することができる。

対照的に、今日のシステムプログラミング言語は強く型付けされている。例えば：

- システムプログラミング言語の各変数は、整数や文字列へのポインタのような 特定の型で宣言され、その型に適した方法で使用されなければならない。
- データとコードは分離されており、その場で新しいコードを作成することは不可能ではないにせよ、難しい。
- 変数は、明確に定義された部分構造を持つ構造体やオブジェクトに集められ、それらを操作するための手続きやメソッドを持つことができる。ある型のオブジェクトを、異なる型のオブジェクトが期待される場所で使用することはできない。

型付けにはいくつかの利点があります。第一に、どのように使用されるかを明確にし、異なる扱いが必要なも のを区別することで、大規模なプログラムをより管理しやすくします。第二に、コンパイラーは型情報を使って、浮動小数点値をポインタとして使おうとするような、ある種のエラーを検出することができる。第3に、型付けは、コンパイラが特殊化されたコードを生成できるようにすることで、パフォーマンスを向上させる。例えば、ある変数が常に整数値を保持することをコンパイラが知っていれば、その変数を操作するための整数命令を生成することができる。コンパイラが変数の型を知らない場合は、実行時に変数の型をチェックするための追加命令を生成しなければならない。

プログラマーは、引数を呼び出しスタックに移動したり、呼び出しスタックから移動したりする心配はない。 # 図1は、プログラミングのレベルと型付けの強さに基づいて、さまざまな言語を比較したものである。

スクリプト言語

Perl、[4] Python、[5] Rexx、[6] Tcl、[7] Visual Basic、Unixシェルなどのスクリプト言語は、システム・プログラミング言語とは全く異なるスタイルのプログラミングを表しています。スクリプト言語は、有用なコンポーネントのコレクションが他の言語ですでに存在していることを前提としています。それらは、ゼロからアプリケーションを書くためのものではなく、むしろコンポーネントを組み合わせるためのものである。例えば、TclやVisual Basicは、ユーザー・インターフェース・コントロールのコレクションを画面上に配置するために使われ、Unixのシェル・スクリプトは、フィルター・プログラムをパイプラインに組み立てるために使われる。

スクリプト言語はコンポーネントの機能を拡張するために使われることが多いが、複雑なアルゴリズムやデータ構造に使われることはほとんどなく、通常はコンポーネントが提供する。スクリプト言語は、グルー言語やシステム統合言語と呼ばれることもある。

スクリプト言語は一般的に型がない

この例では、20以上のプロパティが未指定のままになっている。 # コンポーネントを接続する作業を簡略化するために、スクリプト言語は型がない 傾向がある。すべてのものが同じように見え、同じように動作するため、互換性がある。例えば、TclやVisual Basicでは、変数がある瞬間には文字列を保持し、次の瞬間には整数を保持することができる。コードとデータは交換可能であることが多く、プログラムが別のプログラムを書き、それをその場で実行することができる。スクリプト言語は文字列指向であることが多い。

型付けのない言語は、コンポーネントをフックするのが非常に簡単である。どのように使用できるかに先験的な制限はなく、すべてのコンポーネントと値は統一された方法で表現される。ある目的のために設計されたコンポーネントが、設計者の予期しないまったく異なる目的に使用されることもある。例えば、Unixのシェルでは、すべてのフィルター・プログラムが入力からバイト・ストリームを読み込み、出力にバイト・ストリームを書き込む。一方のプログラムの出力を他方のプログラムの入力に接続することで、任意の2つのプログラムを接続することができる。次のシェル・コマンドは、3つのフィルタをスタックして、「スクリプティング」という単語を含む選択行の数をカウントする：

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

selectプログラムは、現在ディスプレイ上で選択されているテ キストを読み、そのテキストを出力に表示する。grepプログラムは入力を読み、 「scripting」を含む行を出力に表示する。wcプログラムは入力の行数を数える。これらのプログラムはそれぞれ、異なるタスクを実行するために、他にも多くの状況で使用することができる。

システムプログラミング言語の強く型付けされた性質は、再利用を妨げる。システム・プログラミング言語の強く型付けされた性質は、再利用を妨げます。 # プログラマーは、互換性のないさまざまなインターフェースを作成するようになり、それぞれのインターフェースは特定の型のオブジェクトを必要とします。コンパイラーは、たとえそれが便利なものであったとしても、そのインターフェイスで他の型のオブジェクトを使うことを妨げる。そのため、新しいオブジェクトを既存のインターフェイスで使うには、プログラマーはオブジェクトの型とインターフェイスが期待する型を変換するコードを書かなければならない。これには、アプリケーションの一部または全部を再コンパイルする必要があります。今日、多くのアプリケーションはバイナリ形式で配布されているので、これは不可能です。

型なし言語の利点を理解するために、以下のTclコマンドを考えてみましょう：

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

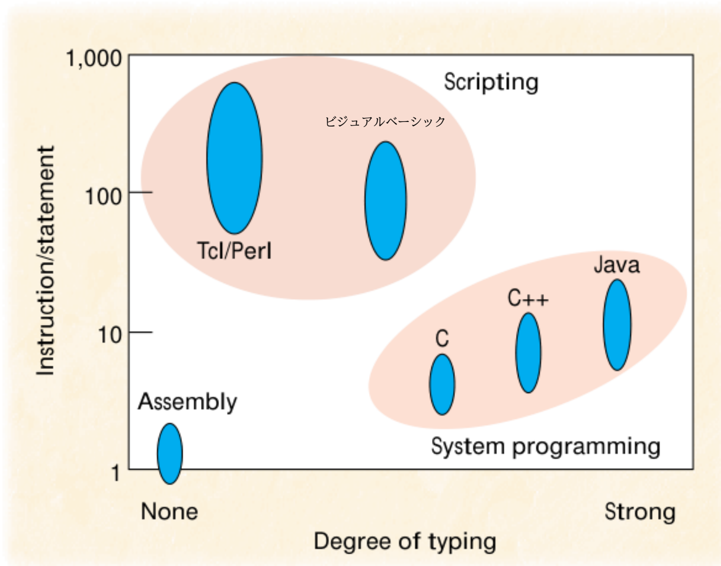


Figure 1. A comparison of various programming languages based on their level (higher-level languages execute more machine instructions for each language statement) and their degree of typing. System programming languages such as C tend to be strongly typed and medium level (five to 10 instructions per statement). Scripting languages such as Tcl tend to be weakly typed and very high level (100 to1,000 instructions per statement).

このコマンドは、16ポイントのTimesフォントで文字列を表示し、ユーザーがコントロールをクリックすると短いメッセージを表示する新しいボタンコントロールを作成します。コマンド名（button）、ボタンコントロール（.b）、プロパティ名（-text、-font、-command）、単純な文字列（Hello!とhello）、書体名（Times）とポイントサイズ（16）を含むフォント名（Times 16）、Tclスクリプト（puts hello）である。Tclはこれらすべてを文字列で統一的に表現する。この例では、プロパティは任意の順序で指定でき、指定されていないプロパティにはデフォルト値が与えられます。

同じ例をJavaで実装すると、2つのメソッドで7行のコードが必要になる。C++とMicrosoft Foundation Classes (MFC)では、3つの手続きで約25行のコードが必要です。 [1] フォントを設定するだけでも、MFCでは数行のコードが必要です：

```
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);
```

このコードの多くは、強い型付けの結果です。ボタンのフォントを設定するには、そのボタンのSetFontメソッドを呼び出す必要がありますが、このメソッドにはCFontオブジェクトへのポインタを渡さなければなりません。そのためには、新しいオブジェクトを宣言して初期化する必要がある。

スクリプト言語は型がないため、エラーが検出されないように思えるかもしれません。

practice scripting languages are just as safe as system programming languages.

CFontオブジェクトを初期化するには、そのCreateFontメソッドを呼び出さなければならないが、CreateFontは14の異なる引数を指定しなければならない硬直したインターフェースを持っている。Tclでは、フォントの本質的な特性（書体Times、サイズ16ポイント）を宣言や変換なしですぐに使うことができる。さらに、Tclでは、ボタンの動作を、ボタンを作成するコマンドに直接含めることができるが、C++とJavaでは、別途宣言されたメソッドに配置する必要がある。（実際には、このような些細な例は、基礎となる言語の複雑さを隠蔽するグラフィカルな開発環境で処理されるでしょう。ユーザーはプロパティ値をフォームに入力し、開発環境はコードを出力する。しかし、条件付き代入のような複雑な状況では、スクリプト言語はシステム・プログラミング言語よりも効率が悪い。プログラムによって生成されたプロパティ値やインタフェースのメン トは、開発者が基礎となる言語でコードを書かなければならない）。

スクリプティング言語は型がないため、エラーが発見されないように思えるかもしれ ませんが、実際にはシステムプログラミング言語と同じくらい安全です。例えば、上記のボタンの例で指定されたフォントサイズがxyzのような非整数の文字列の場合、エラーが発生します。スクリプト言語のエラーチェックは、値が使われる最後の可能な瞬間に行われます。強い型付けによって、コンパイル時にエラーを検出することができるため、実行時のチェックにかかるコストを避けることができます。しかし、効率化の代償として、情報の使用方法が制限される。その結果、コード量が増え、プログラムの柔軟性が低下する。

スクリプト言語は解釈される

スクリプト言語とシステムプログラミング言語のもう一つの重要な違いは、スクリプ ト言語が通常インタプリタ型であるのに対し、システムプログラミング言語は通常コンパイル 型であるということです。インタプリタ言語は、コンパイル時間を省くことで、開発中の迅速なターンアラウンドを提供する。インタプリタ言語はまた、ユーザーが実行時にアプリケーションをプログラムできるようにすることで、アプリケーションをより柔軟にする。例えば、集積回路用の合成・解析ツールの多くにはTclインタプリタが含まれており、プログラムのユーザーはTclスクリプトを書いて設計を指定し、ツールの動作を制御する。インタープリターはまた、その場でコードを生成することで強力な効果を実現することもできる。例えば、Tclベースのウェブ・ブラウザは、いくつかの正規表現置換を使用してページのHTMLをTclスクリプトに変換することにより、ウェブ・ページを解析することができる。そしてTclスクリプトを実行して、画面にページを表示するかわりにインタプリタを使用するためでもあるが、基本的なコンポーネントが、基盤となるハードウェアへの効率的なマッピングよりも、パワーや使いやすさを重視して選択されているためでもある。たとえば、システム・プログラミング言語が1つの機械語に収まるバイナリ値を使うような状況で、スクリプト言語は可変長文字列を使うことが多く、システム・プログラミング言語がインデックス付き配列を使うような状況で、スクリプト言語はハッシュ・テーブルを使う。

幸いなことに、スクリプト言語の性能は通常大きな問題ではありません。スクリプト言語のアプリケーションは一般的にシステムプログラミング言語の アプリケーションよりも小さく、スクリプトアプリケーションの性能は、一般的に システムプログラミング言語で実装されているコンポーネントの性能に支配される 傾向があります。スクリプト言語は、1つのステートメントが平均してより多くの仕事をするという意味で、システム・プログラミング言語よりも高レベルである。スクリプト言語の典型的なステートメントが数百から数千のマシン命令を実行するのに対し、システム・プログラミング言語の典型的なステートメントが実行するマシン命令は5つ程度である（図1参照）。この違いの一部は、スクリプト言語がインタプリタを使用するためだが、その大部分は、スクリプト言語のプリミティブ操作がより高機能であるためである。たとえばPerlでは、正規表現の置換を呼び出すのは、整数の加算を呼び出すのと同じくらい簡単だ。Tclでは、変数にトレースを関連付けることができるので、変数を設定すると副作用が発生する。たとえば、トレースを使用して、変数の値を画面上で継続的に更新し続けることができる。

スクリプト言語は、グルーイング指向のアプリケーションを迅速に開発できる。表1は、この主張を裏付ける逸話である。表1には、システム・プログラミング言語で実装されたアプリケーションをスクリプト言語で再実装したもの、あるいはその逆のものがいくつか紹介されている。その差は、2分の1から60分の1であった。このことは、再実装は最初の実装の経験から大きな恩恵を受け、スクリプト言語とシステム・プログラミングの真の差は、表の極端な点ではなく、むしろ5倍から10倍程度であることを示唆している。スクリプティングの利点もアプリケーションに依存する。表1の最後の例では、アプリケーションのGUI部分はグルーイング指向であるが、シミュレータ部分はそうではない。

表の情報は、comp.lang.tclニュースグループに投稿された記事に対して、 様々なTcl開発者から提供されたものです。

異なるタスクには異なるツールを

スクリプト言語は、システムプログラミング言語の代わりにはならない。

表1．システムプログラミング言語とスクリプトの両方で実装されたアプリケーションの比較 anguages.

Application (contributor)	Comparison	Code ratio*	Effort ratio**	Comments
データベースアプリケーション (Ken Corey)	C++ version: 2 months Tcl version: 1 day		60	C++ version Tcl版の方が高機能だった
Computer system test and installation (Andy Belsey)	C test application: 272,000行、120ヶ月 C FISアプリケーション：90,000行、 60 months Tcl／Perl バージョン：7,700行、 8 months	47	22	C版が先に実装され、Tcl／Perl版が両方のCアプリケーションを置き換えた
Database library (Ken Corey)	C++バージョン：2-3ヶ月 Tcl version: 1 week		8-12	C++ version implemented first
Security scanner (Jim Graham)	C version: 3,000 lines Tcl version: 300 lines	10		最初に実装されたのはC言語版；Tcl言語版の方が高機能だった # 最初に実装されたのはTcl言語版だった
油井の生産曲線を表示 (Dan Schenck)	C version: 3 months Tcl version: 2 weeks		6	
Query dispatcher (Paul Healy)	C version: 1,200 lines, 4-8 weeks Tcl version: 500 lines, 1 week	2.5	4-8	Cバージョンは最初に実装され、コメントなし；Tclバージョンはコメントがあり、より多機能 Tclバージョンは最初に実装された
Spreadsheet tool	C version: 1,460 lines Tcl version: 380 lines	4		
Simulator and GUI (Randy Wang)	Javaバージョン：3,400行、3～4週間 Tclバージョン：1,600行、1週間未満	2	3-4	Tclバージョンは10～20パーセント機能が多く、最初に実装された。

*コード比率は2つの実装のコード行数の比率(<1はシステムプログラミング言語がより多く必要であることを ines).
**意味する）努力比率は開発時間の比率。ほとんどの場合、2つのバージョンは別々の人によって実装された。

スクリプト言語は、システムプログラミング言語の代替にはならない。システム・プログラミング言語では、断片をつなげるために大量の定型文や変換コードを必要とするが、スクリプト言語ではこれを直接行うことができる。複雑なアルゴリズムやデータ構造の場合、システム・プログラミング言語の強力な型付けにより、プログラムの管理が容易になる。実行速度が重要な場合、システム・プログラミング言語は実行時のチェックが少ないため、スクリプト言語よりも10倍から20倍速く実行できることが多い。

特定のタスクにスクリプト言語とシステムプログラミング言語のどちらを使うかを決める際には、 以下の質問を考慮してください：

- アプリケーションの主なタスクは既存のコンポーネントを接続することか?
- アプリケーションは様々なものを操作するのでしょうか?
- アプリケーションにGUIが含まれているか?
- そのアプリケーションは多くの文字列操作を行うか?
- アプリケーションの機能は時間とともに急速に進化するか?
- アプリケーションは拡張可能である必要があるか?

これらの質問に対する肯定的な回答は、スクリプト言語がアプリケーションにとってうまく機能することを示唆する。

一方、以下の質問に対する肯定的な回答は、アプリケーションがシステ ムプログラミング言語に適していることを示唆する：

- アプリケーションは複雑なアルゴリズムやデータ構造を実装するのか?
- アプリケーションが大規模なデータセットを操作するのか，例えば画像の全ピクセルを操作するのか，そのような場合は実行速度が重要になるのか?
- アプリケーションの機能はきちんと定義されていて、変更に時間がかかるか？

過去30年間の主要なコンピューティングプラットフォームのほとんどは、 システムプログラミング言語とスクリプト言語の両方を提供してきた。例えば、粗末なものではあるが、最初のスクリプト言語の1つはジョブ制御言語（JCL）で、OS／360のジョブステップをシーケンスするために使われた。個々のジョブステップは、当時のシステムプログラミング言語であったPL／1、Fortran、またはアセンブラ言語で書かれていた。1980年代のユニックス・マシンでは、システム・プログラミングにはC言語が使われ、スクリプティングにはshやcshといったシェル・プログラムが使われた。1990年代のPCの世界では、システム・プログラミングにはCとC++が、スクリプトにはVisual Basicが使われている。現在形作られつつあるインターネットの世界では、システム・プログラミングにはJavaが使われ、スクリプティングにはJavaScript、Perl、Tclといった言語が使われている。

Scripting and system programming are symbiotic. Used together, they produce programming environments of exceptional power.

スクリプトとシステム・プログラミングは共生関係にある。スクリプトとシステム・プログラミングは共生している。システム・プログラミング言語は、スクリプト言語を使って組み立てることができるエキサイティングなコンポーネントを作成するために使われる。例えば、Visual Basicの魅力の多くは、システム・プログラマーがC言語でActiveXコンポーネントを書くことができ、あまり洗練されていないプログラマーがVisual Basicアプリケーションでそのコンポーネントを使うことができる点にある。Unixでは、C言語で書かれたアプリケーションを呼び出すシェルスクリプトを簡単に書くことができる。新しいコマンドを実装するCコードを書くことで言語を拡張できる。

スクリプトは増加中

スクリプト言語は古くから存在していたが、近年、いくつかの要因が重なって重要性が増している。最も重要な要因は、アプリケーションの構成が、アプリケーションをグルー プ化する方向にシフトしていることである。このシフトの3つの例が、GUI、インターネット、コンポーネントフレームワークである。

グラフィカル・ユーザー・インターフェース

GUIは1980年代初頭に初めて登場し、10年後までには広く普及した。現在では、多くのプログラミング・プロジェクトにおいて、GUIが総工数の半分以上を占めるようになっている。GUIは基本的にアプリケーションの接着剤である。その目的は、新しい機能を作ることではなく、グラフィカル・コントロールのコレクションとアプリケーションの内部機能とのつながりを作ることである。システム・プログラミング言語をベースとしたGUIの迅速な開発環境については、私は知らない。その環境がWindowsであれ、Macintosh Toolboxであれ、Unix Motifであれ、CやC++のような言語をベースにしたGUIツールキットは、習得が難しく、使いづらく、結果の柔軟性に欠けることが証明されている。これらのシステムの中には、基礎となる言語を隠して画面レイアウトをデザインするための優れたグラフィカル・ツールを備えているものもあるが、デザイナーがインターフェイス要素のビヘイビアを提供するコードなどを書かなければならなくなった途端、事態は難しくなる。最高の迅速開発GUI環境はすべて、スクリプト言語をベースにしている： Visual Basic、HyperCard、Tcl／Tkなどだ。

Internet

インターネットの成長もスクリプト言語を普及させた。インターネットは接着剤にすぎない。新しい計算やデータを生み出すのではなく、膨大な数の既存のものに簡単にアクセスできるようにするだけだ。ほとんどのインターネット・プログラミング・タスクにとって理想的な言語は、接続されたすべてのコンポーネントが一緒に動作することを可能にするもの、つまりスクリプト言語である。たとえばPerlはCGIスクリプトを書くのに人気があり、JavaScriptはウェブページでスクリプトを書くのに人気がある。

コンポーネントフレームワーク

スクリプト指向のアプリケーションの3つ目の例は、ActiveXやJavaBeansのようなコンポーネントフレームワークである。システム・プログラミング言語はコンポーネントを作成するのに適しているが、コンポーネントをアプリケーションに組み立てる作業はスクリプトの方が適している。コンポーネントを操作するための優れたスクリプト言語がなければ、コンポーネント・フレームワークの力の多くは失われてしまう。VisualBasicが便利なスクリプティングツールを提供するPCで、スクリプティングがコンポーネントフレームワークに含まれていないUnix／CORBAなどの他のプラットフォームよりも、コンポーネントフレームワークが成功している理由の一端は、このためかもしれない。

より優れたスクリプト技術

スクリプト言語の人気が高まっているもう1つの理由は、スクリプト技術の向上である。TclやPerlのような最新のスクリプト言語は、JCLのような初期のスクリプト言語とはかけ離れている。例えば、JCLは基本的な反復処理さえ提供しなかったし、初期のUnixシェルはプロシージャをサポートしていなかった。スクリプト技術は、今日でもまだ比較的未熟である。例えば、Visual Basicは本当の意味でのスクリプト言語ではなく、もともとは単純なシステム・プログラミング言語として実装され、その後スクリプトに適したように改良されたものです。将来のスクリプト言語は、より現在よりも優れたものになるだろう。ハードウェアの高速化からも恩恵を受けている。かつては、どんな複雑なアプリケーションでも許容できるパフォーマンスを得るには、システム・プログラミング言語を使うしかなかった。場合によっては、システム・プログラミング言語でさえ十分に効率的でなかったため、アプリケーションはアセンブラで書かなければならなかった。しかし、今日のマシンは1980年のマシンに比べて100倍から500倍も高速であり、性能は1年半ごとに倍増している。今日、多くのアプリケーションはインタプリタ言語で実装しても優れた性能を発揮できる。例えば、Tclスクリプトは数千のオブジェクトを含むコレクションを操作することができ、なおかつ優れた対話応答を提供することができる。コンピュータが高速化するにつれて、スクリプトはより大規模なアプリケーションにとって魅力的なものになるだろう。

カジュアルなプログラマー

スクリプト言語の使用が増加している最後の理由の1つは、プログラマーコミュニティの変化である。20年前、ほとんどのプログラマーは大規模なプロジェクトに携わる洗練されたプログラマーだった。その時代のプログラマーは、言語とそのプログラミング環境をマスターするのに数ヶ月を費やすことを想定しており、システム・プログラミング言語はそのようなプログラマーのために設計されていた。

しかし、PCの登場以来、より多くのカジュアル・プログラマーがプログラマー・コミュニティに加わるようになった。これらの人々にとって、プログラミングは本業ではなく、本業を助けるために時々使うツールなのだ。カジュアル・プログラミングの例としては、簡単なデータベース・クエリーやスプレッドシートのマクロなどがある。

一般的なプログラマーは、システム・プログラミング言語の習得に何ヶ月も費やしたがらないが、スクリプト言語であれば、数時間で有用なプログラムを書くのに十分な知識を身につけられることが多い。スクリプト言語は、システム・プログラミング言語よりも構文が単純で、オブジェクトやスレッドなどの複雑な機能が省略されているため、習得が容易である。例えば、Visual BasicとVisual C++を比べてみよう。カジュアルなプログラマーでVisual C++を使おうとする人はほとんどいないだろうが、Visual Basicを使って便利なアプリケーションを作った人は多い。

今日でも、スクリプト言語で書かれたアプリケーションの数は、 システムプログラミング言語で書かれたアプリケーションの数を はるかに上回っている。ユニックスシステムでは、Cプログラムよりもシェルスクリプトの方が多く、ウィンドウズでは、CやC++よりもVisual Basicのプログラムやアプリケーションの方が多い。もちろん、最大かつ最も広く使われているアプリケーションのほとんどは、システム・プログラミング言語で書かれているので、コードの総行数やインストールされているコピーの数で比較すると、やはりシステム・プログラミング言語が有利になるかもしれない。とはいえ、スクリプト言語はすでにアプリケーション開発において大きな力を持っており、その市場シェアは今後さらに拡大するだろう。

オブジェクトの役割

スクリプト言語は、プログラミング言語やソフトウェア工学の専門家からはほとんど見落とされてきた。その代わりに、専門家たちはC++やJavaのようなオブジェクト指向システムプログラミング言語に注目してきた。オブジェクト指向(OO)プログラミングは、プログラミング言語の進化における次の大きなステップになると広く信じられている。強力な型付けや継承といったOOの特徴は、開発時間を短縮し、ソフトウェアの再利用を増やし、スクリプト言語が扱う問題を含む他の多くの問題を解決するとしばしば主張される。

OOプログラミングは、実際にどれだけの利益をもたらしたのだろうか？残念ながら、私はこの質問に明確に答えられるだけの定量的なデータを見たことがない。私の意見では、オブジェクトがもたらす恩恵はわずかである。おそらく生産性は20～30%向上するだろうが、10倍はおろか2倍にもならないだろう。オブジェクトは、スクリプティングのような劇的な生産性の向上はなく、OOプログラミングの利点はスクリプティング言語で達成できる。

OOプログラミングは、プログラミングのレベルを上げるものでも、再利用を促すものでもないため、生産性を大きく向上させるものではない。C++のようなOO言語では、プログラマーは依然として、詳細に記述し操作しなければならない小さな基本ユニットを扱う。原理的には、強力なライブラリ・パッケージを開発することができ、これらのライブラリが広く使用されれば、プログラミングのレベルを上げることができる。しかし、そのようなライブラリはほとんど存在しない。ほとんどのOO言語の強力な型付けは、再利用が困難な狭義のパッケージを奨励している。各パッケージは特定の型のオブジェクトを必要とする。2つのパッケージを一緒に使うには、パッケージが要求する型間を変換するコードを書く必要がある。

> 実装継承は
>
> **intertwining and brittleness that have been observed when**
>
> goto文が多用される。その結果、OOシステムはしばしば複雑さと再利用の欠如に悩まされる。

問題は、継承を重視することである。あるクラスが別のクラスのために書かれたコードを拝借する実装継承は、ソフトウェアの管理や再利用を難しくする悪い考えだ。クラスの実装を束縛してしまうので、どちらのクラスも他方のクラスなしでは理解できない。サブクラスは、継承されたメソッドがスーパークラスでどのように実装されているかを知らなければ理解できず、スーパークラスは、そのメソッドがサブクラスでどのように継承されているかを知らなければ理解できない。複雑なクラス階層では、階層内の他のすべてのクラスを理解しなければ、個々のクラスを理解することはできません。さらに悪いことに、再利用するためにクラスを階層から切り離すことはできません。多重継承はこれらの問題をさらに悪化させます。実装継承は、goto文が多用されたときに観察されたのと同じように、絡み合いやもろさを引き起こす。その結果、OOシステムはしばしば複雑さと再利用の欠如に悩まされる。

一方、スクリプト言語は、実際にソフトウェアの再利用を大きく促進してきた。スクリプト言語が従うモデルは、興味深いコンポーネントをシステム・プログラミング言語で構築し、スクリプト言語でアプリケーションにまとめるというものだ。この分業は、再利用性のための自然なフレームワークを提供する。コンポーネントは再利用できるように設計されており、コンポーネントとスクリプトの間には、コンポーネントを使いやすくするための明確なインターフェイスがある。例えば、Tclでは、コンポーネントはCで実装されたカスタム・コマンドであり、組み込みコマンドと同じように見えるので、Tclスクリプトで簡単に呼び出すことができる。Visual Basicでは、コンポーネントはActiveXの拡張機能であり、パレットからオブジェクトをドラッグすることで使用できる。

それにもかかわらず、OOプログラミングは2つの有用な機能を提供する。1つ目はカプセル化である。オブジェクトは、実装の詳細を隠す方法でデータとコードを結合する。これにより、大規模なシステムの管理が容易になる。2つ目の便利な機能はインターフェースの継承で、クラスが同じメソッドやアプリケーション を提供する。

プログラミングのレベルを上げることは、言語設計者にとって最も重要な目標である、as it has the greatest effect on programmer productivity. It is not clear that strong typing contributes to this goal.

実装が異なっていても、API（プログラミング・インターフェース）を使用することができます。これによってクラスは互換性を持ち、再利用が促進される。

幸いなことに、オブジェクトの利点はシステム・プログラミング言語だけでなくスクリプト言語でも実現可能であり、事実上すべてのスクリプト言語がOOプログラミングをある程度サポートしている。例えば、PythonはOOスクリプト言語であり、Perl v5はオブジェクトをサポートしている。Object RexxはRexxのOOバージョンであり、Incr TclはTclのOO拡張である。オブジェクトRexxはRexxのOOバージョンであり、Incr TclはTclのOO拡張である。1つの違いは、スクリプト言語のオブジェクトは型付けされない傾向があり、システム・プログラミング言語のオブジェクトは強く型付けされる傾向があることである。

他の言語

この記事は、すべてのプログラミング言語の完全な特徴付けを意図したものではない。プログラミング言語には、型付けの強さやプログラミングのレベル以外にも多くの属性があり、システム・プログラミング言語やスクリプト言語としてきれいに特徴付けることができない興味深い言語がたくさんある。例えば、Lispファミリーは、スクリプト言語とシステム・プログラミング言語の中間に位置し、それぞれの属性を併せ持っている。Lispは、解釈や動的型付けといった、現在スクリプト言語で一般的な概念や、自動ストレージ管理や統合開発環境といった、現在スクリプト言語とシステム・プログラミング言語の両方で使用されている概念の先駆者である。

スクリプト言語は、システム・プログラミング言語とは異なるトレードオフを意味する。スクリプト言語は、システム・プログラミング言語と比較して、実行速度と型付けの強さを放棄しているが、プログラマーの生産性とソフトウェアの再利用性は大幅に向上している。このトレードオフは、コンピュータがプログラマーに比べて高速で安価になるにつれて、ますます理にかなってくる。システム・プログラミング言語は、複雑さがデータ構造とアルゴリズムにあるコンポーネントを構築するのに適しており、スクリプト言語は、複雑さが接続にあるアプリケーションをグルーイングするのに適している。そのため、スクリプトは次の世紀においてますます重要なプログラミング・パラダイムとなるだろう。

私は、この記事が3つの点でコンピューティング・コミュニティーに影響を与えることを望んでいる。第一に、プログラマーが新しいプロジェクトを始めるときに、スクリプトとシステム・プログラミングの違いを考慮し、それぞれのタスクに最も強力なツールを選択することを望む。第二に、コンポーネント・フレームワークの設計者がスクリプティングの重要性を認識し、コンポーネントを作成する機能だけでなく、コンポーネントを接着する機能もフレームワークに含めるようにしてほしい。

最後に、プログラミング言語の研究コミュニティがスクリプト言語に関心を移し、さらに強力なスクリプト言語の開発に貢献することを望む。プログラミングのレベルを上げることは、言語設計者にとって最も重要な目標である。強力な型付けがこの目標に貢献するかどうかは定かではない。❖

謝辞

ジョエル・バートレット（Joel Bartlett）、ビル・エルドリッジ（Bill Eldridge）、ジェフリー・ヘーマー（Jeffrey Haemer）、マーク・ハリソン（Mark Harrison）、ポール・マクジョーンズ（Paul McJones）、デビッド・パターソン（David Patterson）、スティーブン・ウーラー（Stephen Uhler）、ハンク・ウォーカー（Hank Walker）、クリス・ライト（Chris Wright）、コンピューター・レフェリー（Computer referees）、そして初期の草稿のネットニュースでの白熱した議論に参加した何十人もの人々などである。Colin StevensがMFC版のボタン例を書き、Stephen UhlerがJava版を書いた。

References

1. J. Ousterhout, "Additional Information for Scripting White Paper," http://www.scriptics.com/people/john.ousterhout/scriptextra.html.
2. C. Jones, "Programming Languages Table, Release 8.2," Mar. 1996, http://www.spr.com/library/0langtbl.htm.
3. B. Boehm, *Software Engineering Economics,* Prentice Hall, Englewood Cliffs, N.J., 1981.
4. L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl,* 2nd ed., O'Reilly & Associates, Sebastopol, Calif., 1996.
5. M. Lutz, *Programming Python,* O'Reilly & Associates, Sebastopol, Calif., 1996.
6. R. O'Hara and D. Gomberg, *Modern Programming Using REXX,* Prentice Hall, Englewood Cliffs, N.J., 1988.
7. J. Ousterhout, *Tcl and the Tk Toolkit,* Addison-Wesley, Reading, Mass., 1994.
8. S. Johnson, "Objecting To Objects," Invited Talk, Usenix Technical Conf., San Francisco, Jan. 1994.

John K. OusterhoutはScriptics Corp.のCEOである。以前はサン・マイクロシステムズ社のディスティングイッシュト・エンジニアで、本稿の執筆に携わった。専門はスクリプト言語、インターネット・プログラミング、ユーザー・インターフェース、オペレーティング・システム。Tclスクリプト言語とTkツールキットを開発。エール大学で物理学の理学士号を、カーネギーメロン大学でコンピューターサイエンスの博士号を取得。また、ACMグレース・マレー・ホッパー賞、米国国立科学財団大統領若手研究者賞、カリフォルニア大学バークレー校特別教授賞など多くの賞を受賞している。

連絡先：Ousterhout、ouster@scriptics.com。