2
Speaking in Code

I have traveled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won't last out the year.

—*The editor in charge of business books for Prentice Hall, 1957*[1]

INSIDE THIS CHAPTER

- The birth and evolution of modern computing
- The development of operating systems and software platforms
- The role of APIs in reducing duplication of programming efforts
- The production of commercial and open-source software

Software platforms come in several varieties, depending on what the code does and where among the various computing devices it resides.

Some have a single block of code that does everything from controlling the switches in the microprocessors to helping applications show three-dimensional objects. When you play Doom 3 on your Apple PC, the Mac OS X code is doing a lot of the work.

Others come in pieces. There is code on the device that controls the microprocessor. Then there is another piece that provides services to programmers who are writing applications for the device. The Nokia Series 60 Platform is what mobile phone applications use for many software services. The Nokia platform in turn relies on the Symbian OS to control the phone hardware.

1. Allan Afuah, *Innovation Management: Strategies, Implementation and Profits* (New York: Oxford University Press, 2003).

Still other software platforms are written so that they can work on multiple devices with different microprocessors and operating systems that control those microprocessors. The Java 2 Micro Edition provides technologies that enable developers to write programs that can run on consumer electronics devices with different chips and operating systems.

These alternatives for building and providing software platforms have key consequences for the structure of industries based on computer devices and the dynamics of competition within these industries.

The Historical Foundations of Computers and Programming

A program tells a computer what to do.

A loom designed by Joseph Jacquard in 1801 was the first programmable computing device. The Jacquard loom used punch cards made of stiff pasteboard to control the patterns of threading through the fabric. It revolutionized the textile industry—after a rebellion of weavers who feared it would eliminate their jobs was put down. Punch cards remained the dominant method for transmitting programs to computing devices until the late 1970s: "do not fold, spindle, or mutilate" was a famous programmer admonition.[2]

The origins of modern computing lie in efforts to make performing complex calculations easier. Charles Babbage developed the basic ideas behind mechanical computing and programming in the early nineteenth century out of frustration.[3] As Babbage wrote in his memoirs,

> . . . I was sitting in the rooms of the Analytical Society, at Cambridge, my head leaning forward on the table in a kind of dreamy mood, with a table of logarithms lying open before me. Another member, coming into the room, and seeing me half asleep, called out, "Well, Babbage, what are you dreaming about?" to which I replied "I am thinking that all these tables" (pointing to the logarithms) "might be calculated by machinery."

He invented mechanical methods (using a machine called the Difference Engine) for calculating astronomical and mathematical tables, and

2. http://ccat.sas.upenn.edu/slubar/fsm.html.

3. Charles Babbage, *Passages from the Life of a Philosopher* (London, 1864), http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Babbage.html.

reported them in an article published in 1822.[4] He went on to describe an "Analytical Engine" based on Jacquard's punch cards that contained many of the features of modern computers.

Though it was never actually built, Babbage's Analytical Engine led to a mathematical literature on how to write programs to solve problems with it. Ada King, the countess of Lovelace and Lord Byron's daughter, is often credited, along with Babbage, with whom she collaborated closely, with the first computer program—a set of instructions for calculating a sequence of numbers (+1, $-\frac{1}{2}$, $+\frac{1}{6}$, . . .) known as Bernoulli numbers. Although it is a matter of dispute whether she was an originator, interpreter, or popularizer, many of the ideas of contemporary programming were presented in her 1843 annotated translation of Menabrea's *Notions sur la machine analytique de Charles Babbage*.

The British government withdrew funding for an advanced version of Babbage's Difference Engine, and the Analytical Engine was beyond the technology available in the mid-nineteenth century. Further significant advances were not made until the demands of World War II, combined with technical progress, resulted in several breakthroughs, and until mathematicians Alan Turing, Claude Shannon, and John von Neumann laid the modern foundations of computer programming in the years before and after the war.

In the late 1930s, Turing, who is famous for writing programs that helped crack the Germans' Enigma code during World War II, published a paper that introduced what is now known as the Turing machine. It described the features of a mathematical computational device using a tape the machine can read and write on, and it defined a group of tasks that could be computed using this device.[5]

4. http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Babbage.html. The Difference Engine was "programmed" to evaluate a polynomial such as $y = a \times x^2 + b \times x + c$ for successive values of x given the values of a, b, and c.

5. A Turing Machine is a "state machine": at any time it is in one of a finite number of states. It has a head that can read and write symbols—a 0 or 1, for example—onto an infinite tape. For every combination of a current state and the symbol under the head, a new state and action are defined. The action can be one of the following: change the symbol under the head, move the head one step right, or move the head one step left. The machine halts if there is no action defined for the current combination of a state and symbol under the head.

A program for this machine consists of a series of instructions:

```
X ← Read the symbol under the head;
If there is a transition rule for the combination of X and the current state
N
   Then write Y on the square under head or move the head one step left
or right
   Then change state to M
If not halt
Repeat
```

All current computers and higher programming languages are mathematically identical to a Turing machine, though much easier to use. The concepts of *if* and *then* are important parts of the programming idiom.

Claude Shannon recognized that 0s and 1s could be used to represent whether relay and switching circuits were on or off. In his 1938 MIT master's thesis, he demonstrated that these circuits could be used to perform complex calculations based on a series of 0s and 1s.[6] A decade later he coined the term bit—from *bi*nary digi*t*—in a paper on signal processing that founded modern information theory.

By the time of Shannon and Turing, hardware technology had progressed far enough that their ideas were soon put into action. Mechanical computing machines had been around for many years, ranging from the abacus for simple calculations to gear-shaft and cog systems that could solve differential equations. The demands of World War II saw the development of electronic computers in England, Germany, and the United States. The ENIAC, mentioned in the introduction, is considered the first fully operational electronic general-purpose computer. Unlike its special-purpose mechanical predecessors, it had "conditional jumps" (such as "if *X* go to instruction *N*") and could do many different kinds of calculations. Operators programmed it manually by setting switches and plugging cables. Data for programs were entered and stored on punch cards. This system was tedious.

6. Shannon relied on a branch of mathematics known as Boolean algebra.

John von Neumann helped devise what became the programming architecture of modern computers. His key insights were that instructions could be reduced to binary values and that both instructions and data could be stored efficiently in memory. This led to an architecture consisting of five components: an input unit, a control unit, memory, a calculating unit, and an output unit. The instructions are fetched and executed one at a time—sequentially—by the central processing unit (CPU). The CPU must have almost instantaneous access to memory for this to work. Practically, that resulted in a hierarchy of memory based on access speed, including what is now called random access memory (RAM). The stored program computer became the mainstay of the computer industry.

The Development of Programming Languages

A programming language has a vocabulary and grammatical rules that permit humans to communicate instructions to computers.

From the introduction of the ENIAC, machines have only understood a language consisting of 0s and 1s. That is called *machine language*. The invention of a stored program permitted computer operators to convey instructions through punch cards containing 0s and 1s rather than manipulating cables and toggles. That tedious process was made simpler by the invention of what is known as *assembly language* to convey these instructions. Something like the use of "LOL" for "laughing out loud," "li $t0 8" instructs the computer to load the value 8 into register t0 in the processor; this command replaces writing 001101 00000 01000 00000 00000 001000 in machine language.[7] Short Code, invented in the late 1940s, was the first programming language. The programmer used its symbols to write out a program. When complete, she then had to translate the symbols back into 0s and 1s. A few years later that tedious process was eliminated with the development of a *compiler* that did this translation automatically.

Higher-level languages were then developed that made "writing" complex programs substantially easier because the programming is done

7. 34080008 in hexadecimal notation.

at a level that is intuitive to humans. The programmer can simply instruct the computer to add two numbers, for instance, without keeping track of where in the CPU they and their sum are stored. FORTRAN, developed at IBM for scientific computing and introduced in 1957, was the first of these languages. Its vocabulary provides a sense of what it could do: IF, THEN, GOTO, DO, END, TRUE, FALSE. It was especially popular for scientific applications. Other languages were developed over time, such as COBOL, which was used mainly for writing business applications. FORTRAN and COBOL were used to develop applications for mainframe computers, large computers owned by enterprises. Today, most of the code running on mainframe computers is in COBOL.

Another popular language was BASIC. It was introduced as a teaching tool at Dartmouth College in 1963. It had a simple vocabulary and grammar and was easy for beginners to use. It became the leading language for the PCs introduced in the late 1970s in part because it required little memory. Microsoft's first product was a version of BASIC for the Altair, the first PC. Soon after dropping out of Harvard, Microsoft founder Bill Gates wrote a BASIC *interpreter*—which is similar to a compiler in the sense that it translates a higher-level language into something the machine can understand—that fits into 4 kilobytes (kb) of memory (about 10 percent of the memory on a smart credit card).[8]

The simple program in Figure 2.1 illustrates the role of a high-level language. We start with an algorithm for calculating 2 multiplied by itself n times (that is, 2 to the nth power), where the user can specify any n she would like. We then write a program in BASIC that communicates this algorithm to a computer. That program is then translated into 0s and 1s. We represent the 0s and 1s in hexadecimal notation (each pair of hexadecimal digits corresponds to a unique combination of eight 0s and 1s).

Even in the early days of computing, programs designed for business or technical applications might have had many thousands of lines of high-level code. In order to write such large programs in a timely fashion it was necessary to have many individuals working in parallel. Unfortunately, changes to any one part of a large program may affect how other

8. Smart cards have 24 kb of ROM and 16 kb of programmable ROM. http://electronics.howstuffworks.com/question332.htm.

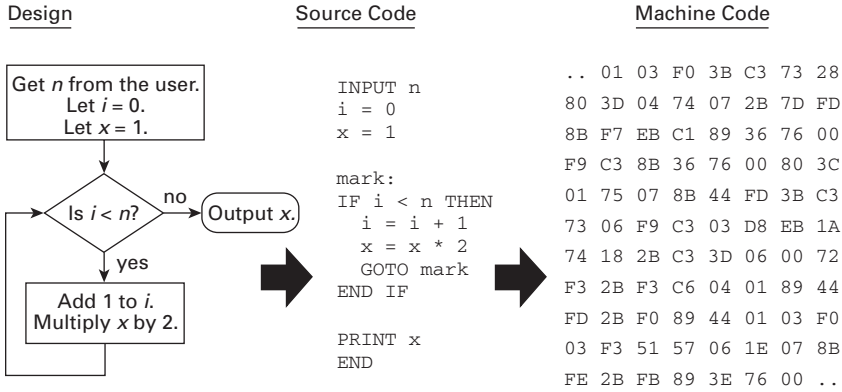


Figure 2.1
Simple program: from design to source code to machine code. The machine code is written in hexadecimal notation (where numbers go up to 16 and A–F represent 11–16).

parts operate, so the more individuals there are working in parallel on a single program, the more likely it is that they will create problems for one another and thus slow the effort down.[9]

The solution to this problem has been to exploit the power of *modularity* in software design.[10] The basic idea is to move from the specification of what a new program as a whole should do to the specification of an *architecture* that describes the overall program as a set of modules, specifies the functions that each module is to perform, and specifies the interfaces that link them. In a program to handle payroll, for instance, one module might be assigned the task of calculating each employee's Social Security contribution as a function of current law, this period's earnings, and past contributions—all supplied by other modules. If the program's architecture is sound, individuals or small

9. This is a general property of complex systems: changing the engine in a car, for instance, may require making changes in its brakes, fuel tank, frame, and many other components. The classic discussion in the context of software is Frederick P. Brooks, *The Mythical Man-Month: Essays in Software Engineering* (New York: Addison-Wesley, 1975).

10. This powerful idea is used in a wide variety of design contexts. See Carliss Y. Baldwin and Kim B. Clark, *Design Rules* (Cambridge, Mass.: MIT Press, 2000).

teams can work in parallel in the various modules, and if the modules meet their specifications, the program built by linking them together will operate as intended.

The modular approach has numerous advantages. If a new program (or other complex system) can be specified as *N* modules, *N* teams can work in parallel. Moreover, individual modules can subsequently be improved without touching other parts of the overall program, and they can be used in other programs. On the other hand, specifying an architecture in detail is complex, and unforeseen interdependencies between modules often occur in development and have to be resolved. Because of the complexity involved, innovation at the architectural (as opposed to modular) level is difficult. Finally, an architecture that facilitates program development (by having a large number of modules and simple interfaces, say) may fail to optimize performance (by, in effect, requiring excessive communication between modules, for instance). There are trade-offs between and among modularity, design costs, and efficiency.

Object-oriented programming is a recent innovation in high-level languages based on modularity. A program is written as a set of *objects*, each of which corresponds to particular functions and data. The other parts of the program can then use these objects to get access to the data and functions they include. These objects make it easier to reuse code for multiple purposes. The programmer can just take one of these components off the shelf, so to speak, and deploy it when needed. The programmer can also decide to make any object off-limits to certain other parts of the program in the interests of reliability.

Two high-level languages, C and Java, are widely used today for writing much of the software discussed in the following pages.

C, along with its variants (including C+, C++, and C#), is the most widely used programming language. It was developed at Bell Laboratories in the 1970s. Though difficult to learn, it is one of the most powerful and flexible languages for writing efficient programs. It is especially popular for writing system software such as the operating system. C++ added object-oriented programming to C.

Sun's Java programming language is part of a set of programming technologies that are designed so that applications can run on many differ-

ent operating systems and hardware configurations: "write once, run everywhere" is its aspiration and mantra. It was originally developed for handheld devices and has become widely used for writing small Web-based applications. Similar to C++, Java is an object-oriented language. Java programs go through a series of translations that enable them to be run on different machines. The programs are first compiled into *byte code* files, which can then be translated into machine-level instructions by a *Java Virtual Machine* that is written specifically for an individual operating system or hardware platform.

Evolution of Operating Systems

Modern operating systems are software programs. If you think about the computer platform from the viewpoint of the microprocessor, the operating system is usually the only program from which the CPU receives instructions. All other programs sit on top of the operating system and interact with the CPU only through it. Today, most operating systems are written in a high-level language such as C++ and are then translated into machine language before being installed on the computer hardware.

There were no operating systems for the early computers. From the late 1940s to the mid-1950s, a person ran the machine from a console that had toggle switches and display lights. Programs in machine language were submitted to the machine through a punch card reader. Debugging programs required looking at lights for the processor registers and main memory to figure out the source of the errors. Moreover, it was hard to schedule time with the expensive computing hardware. Only one program could run at a time, and users had to schedule blocks of time. They might finish early and leave the computer idle for a time, or not get any results at all before their time ran out.

Batch operating systems were developed to maximize the utilization of these expensive machines. General Motors created the first in the mid-1950s for use on its IBM 701 computer.[11] Other customers followed

11. Frank Hayes, "The Story So Far: Bell Labs, GM and MIT Played Major Roles in the Development of Operating Systems," Computerworld, 30, March 17, 2003.

suit. By the early 1960s, many computer manufacturers had developed batch operating systems for their machines. These early systems had a *monitor.* Although now synonymous with the screens many of us look at for hours every day, the original monitor was a portion of the code that acted as a sentry. That sentry controlled the sequencing of instructions to the processor, prevented user programs from altering memory where the monitor program itself resided, reserved for itself certain privileged instructions such as input and output, and timed programs to prevent them from using system resources for too long. Users could communicate with the monitor using *job control language* (JCL). In these early days of computing, programmers put canned JCL instructions at the beginning and end of their programs—all on punch cards.

In those days, inputting instructions and outputting results accounted for most of the time it took to run a program. The CPU was mostly idle while this was going on. Multiprogramming (or multitasking) was developed to make better use of the CPU. It required hardware that enabled the processor to be interrupted when input and output operations were completed and there was computing to be done. And it required memory management that enabled several programs to be kept in the main memory and that juggled the execution of these programs around while input and output operations were being conducted. One of the first multiprogrammed, batch operating systems was the IBM OS/360 for the System/360 in 1964.[12]

"As more and more features have been added to operating systems, and as the underlying hardware has become more capable and versatile," William Stallings has observed, "the size and complexity of operating systems has grown."[13] UNIVAC's operating system for the 1107,

12. A related development during the late 1960s and 1970s was time-sharing. Programmers interacted directly with the computers during the very early years, and, although it was inconvenient, they could see and debug errors in real time. Batch processing cut the connection between the programmer and computer. The programmer had to submit a job and get the results back from the computer operator. Multiprogramming helped make it possible for multiple users to interact with a machine. Users submitted commands at a terminal and got responses back from the computer. Time-sharing systems traded increased processing use for decreased response time for the user.

13. William Stallings, *Operating Systems: Internals and Design Principles*, 4th ed. (Upper Saddle River, N.J.: Prentice Hall, 2001), p. 77.

announced in 1960, had 25,000 lines of code.[14] IBM's OS/360 had a million lines when introduced in 1964. The Multics operating system, developed by MIT and Bell Labs, had 20 million in 1975. Windows XP has about 40 million.[15] These are not apple-to-apple comparisons because they are for different programs written for different systems. Nonetheless, they highlight an important trend throughout the history of operating systems.

Operating system designers have added features that make improved use of the hardware or that control new hardware features. More important for the evolution of industries based on software platforms, operating system designers have also added many features that save programmers from having to write their own code. Just as object-oriented programming helps programmers avoid reinventing the proverbial wheel—such as code for displaying data in three dimensions—developing operating systems as rich sets of modules and making public the interfaces that link them saves programmers of diverse applications from having to write code for a wide variety of common tasks. Indeed, this modularity has transformed operating systems into software platforms. We explain how next.

Application Programming Interfaces

Operating systems provide services to applications through Application Programming Interfaces (APIs). These services range from rudimentary hardware services, such as moving a cursor on a monitor, to sophisticated software services, such as drawing and rotating three-dimensional objects. The APIs serve as interfaces between these services and applications. (As we discuss later, they may also serve as interfaces between modules of the operating system itself.)

Applications obtain services by passing specific information to the APIs and obtaining other information back. The API, which the programmer sees, calls on a black box (a system module), which the programmer does not see, to perform a specific task. The method is similar to the mathematical functions included in high-level programs. Suppose

14. http://www.cc.gatech.edu/gvu/people/randy.carpenter/folklore/v1n3.html.
15. http://en.wikipedia.org/wiki/Source_lines_of_code.

you wanted a computer to multiply two numbers, *X* and *Y*. You could write that program in machine language to get the microprocessor to do that. But since multiplication is a common problem that people face, high-level languages all have functions that do this for you. All you have to do is give the value of the two numbers to a function in the program and the function will return the result. In particular, you might write *X* = 6, *Y* = 7, and *Z* = *X* ∗ *Y*; "∗" is often the symbol that tells the high-level language that it should insert *X* and *Y* as arguments in its multiplication algorithm. You will get the answer 42, although you will never see the machine code that does the calculation.

The APIs for operating systems take "arguments" (like the 6 and 7 in the example above) from the application program and call on "system services" (like the machine code that calculates 6 times 7) to perform the work desired by the application program. The Linux kernel has an API, shown in Figure 2.2, that allocates memory for an object that the application intends to use. The application has to specify the size of the object in bytes and some parameter flags for the API. The Linux source code then allocates the memory necessary for this object and returns a pointer to enable the application to find the memory needed for the object. This API is simple. It just enters information into the function (_cache_alloc) that calls on the system services to do the work. There are at least 116 lines of source code in Linux that carry out the work required by this seven-line API.[16] As a result, the programmer can avoid writing more than 100 lines of code by inserting the necessary data into this seven-line API.

It is easy to see why application developers find the ability to access system services through APIs appealing. Rather than every application developer writing hundreds of lines of code to allocate memory to an object, to take the example above, the operating system developer writes 116 lines of code and makes the system services this code provides available to all application developers through the API.

Some operating systems provide fairly minimal services to applications. Others devote large amounts of code to features that applications

16. The 116 lines are part of methods one or two levels below the original API. Going deeper into the kernel would raise this number significantly.

```
void * __kmalloc (size_t size, int flags)
{
     struct cache_sizes *csizep = malloc_sizes;

     for (; csizep->cs_size; csizep++) {
         if (size > csizep->cs_size)
                  continue;

          return __cache_alloc(flags & GFP_DMA ?
               csizep->cs_dmacachep : csizep->cs_cachep,
flags);
     }
     return NULL;
}
```

Figure 2.2
An example of a Linux API. (Source: Linux Kernel 2.6.10; File: slab.c.)

can use. The computer device itself influences this choice. Because a mobile phone has limited memory, for instance, an operating system for a mobile phone cannot do as much as an operating system for a PC. (Of course, today's mobile phone can do more than PCs could in the 1980s.) An operating system for a single-purpose device such as the temperature control system in a car need not cater to applications because at least for now, there is no demand for applications to run on such a device. The business model adopted by the operating system manufacturer also determines the services it provides to applications. Some manufacturers may decide to focus mainly on hardware and let others provide system services. Sony provides significantly fewer services to game developers on its PlayStation than Microsoft does on its Xbox, for instance.

Middleware also provides application services. These software programs leave most of the hardware interactions to the operating system that they work with. They specialize in providing more advanced software features on which application writers can rely.

Some middleware complements the operating system in the sense that there are few overlaps in services. That is true, for example, of the Symbian OS and the Nokia Series 60 and 80 Developer Platform.

Other middleware competes with the operating system by providing services that are also in the operating system. That is true of Java. It provides software services that programmers can use in place of the services available in the operating systems with which Java works—Linux, Mac OS, Windows, and a host of others. Then, on each hardware platform on which Java runs, Java leaves the job of controlling the hardware to the operating system running on that hardware.

Software Platforms

The software platform is the set of programs that stand between the hardware and an application. A particular computing device may have several possible software platforms for an application based on different combinations of operating systems and middleware programs.

Mobile telephones provide an interesting example. Consider a mobile telephone based on the Texas Instruments OMAP line of microprocessors; these are used in Nokia phones, for example. Possible software platforms are shown in Figure 2.3. In the first three stacks the operating system (Linux or Windows Mobile) provides APIs for mobile phone applications. In the fourth stack, Symbian, which is partly owned by Nokia, provides mainly hardware services and Nokia's middleware provides application services.

Consider also a PC based on an Intel chip. Figure 2.4 shows some of the possible software platform configurations. In this example there are two alternative operating systems, Linux and Windows. Each provides

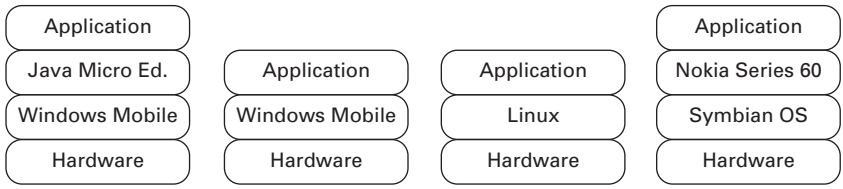


Figure 2.3
Alternative mobile telephone platforms.

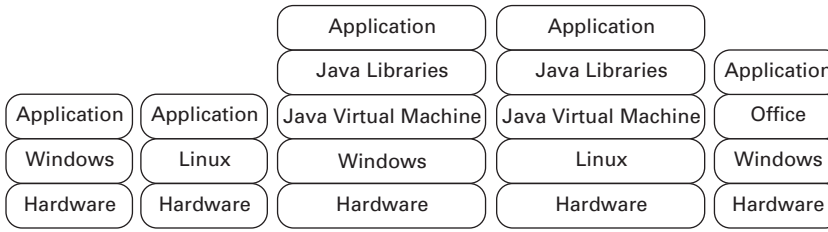


Figure 2.4
Alternative personal computer software platforms. Here, "Linux" includes both Linux kernel and utilities.

a rich set of application services through a series of APIs. However, either can work with Java middleware as long as there is a Java Virtual Machine running alongside the operating system. In addition, other applications may use specialized services made available in other software such as Office.

Three popular software platforms illustrate some of the diverse ways of building platforms.

Windows XP

Windows XP descends from Windows NT, which Microsoft released in 1993. Windows NT was written for processors that could handle instructions and numbers represented with 32 bits (that is, 32 sequential 0s or 1s), compared with 16 bits for earlier versions of Windows. Microsoft has upgraded (and renamed) Windows NT over time while retaining the same basic architecture. With Windows XP, Microsoft unified its lines of desktop operating systems for home (Windows 95, 98, and Me) and business (Windows NT and 2000). In addition, Windows XP added new technology for keeping track of items across computers on a network and other features that were appealing for interconnected computing.

Written mainly in C++, versions of Windows XP have been available for client computers (the computers that sit on our desks or laps) and server computers (the networked computers that focus on such things as Web applications and corporate databases) using Intel-compatible microprocessors. The operating systems for clients and servers have much of their code and APIs in common, but the

server operating systems include various services for servers on networks of computers.

Windows NT and its successors relied on the modular and object-oriented design concepts mentioned earlier. One can think of this as using the concept of APIs within the operating system code. A lower-level object provides services to higher-level objects. For example, in Microsoft Windows XP, Internet Explorer relies on a module called the Microsoft XML Core Services, which is used to interpret XML documents. The same Microsoft XML Core Services module is used by other parts of the operating system, such as the part that handles playing media.[17]

Windows XP also has an extensive set of application services that are known as the Win32 API Set. Bill Gates put the number of APIs in the initial version of Windows 2000 at more than 6,000.[18] The next version of the Windows operating system, Vista, will feature sophisticated support for digital rights management (see Chapter 8), a new user interface technology, and many new application services made available through the WinFX API Set. It will have a total of more than 1,000 APIs available to developers. (The number of APIs reported here is based on the ones that Microsoft documents and manages.)

Unix and Linux

Linux is an operating system that is widely used on server computers and is making headway in everything from embedded devices to mobile telephones to client computers. It has descended from Unix. Unix in turn is an operating system that was widely used for minicomputers and workstations during the 1970s and 1980s and remains popular for servers and other heavy-duty uses.

Computer scientists at Bell Labs developed the first versions of Unix. It was first used on a minicomputer made by Digital Equipment Corpo-

17. http://support.microsoft.com/?scid=kb;en-us;q272633; http://download.microsoft.com/download/9/6/5/9657c01e-107f-409c-baac-7d249561629c/MSXML4SP_RelNote.htm; http://www.microsoft.com/downloads/details.aspx?FamilyID=3144b72b-b4f2-46da-b4b6-c5d7485f2b42&DisplayLang=en.

18. U.S. v. Microsoft, Civil Action No. 98–1233, Direct Testimony of Bill Gates, ¶56.

ration. Like most operating systems at the time, it was written in assembly language; at the time, this was considered the best approach for writing complex programs that were also efficient. In a major innovation a few years after its introduction, it was rewritten in C, thereby demonstrating the advantages of using a high-level language for writing most operating system code. Bell Labs licensed the Unix source code to universities and businesses. Several developers started modifying the code. Since they did not do this in concert, the Unix code "forked," leading over time to multiple versions of the operating system, and programs written for one version could not necessarily run on another version without modification. The most popular of these versions was BSD Unix, created at the University of California, Berkeley.

What is normally considered the Unix operating system is a collection of several components. These include the kernel, an interface that permitted applications to call on the services of the kernel and therefore the hardware, and a series of canned commands and libraries of programs that users can rely on for calling on system services directly or can use in applications written for Unix. Over time, the various versions of Unix have improved the kernel. For example, Sun's Solaris operating system is based on a version of Unix (System V Release 4) that AT&T and Sun Microsystems developed jointly. These partners completely rewrote the kernel that was the basis for the standard version of Unix at the time.

Linux is based on a variant of BSD Unix that can run on IBM-compatible personal computers. Linus Torvalds wrote the original Linux kernel in C and posted an early version on the Web in 1991. He used software tools that were developed by the Free Software Foundation. Working over the Internet, programmers from around the world suggested improvements and additions to the kernel as well as other utilities. Torvalds incorporated these periodically into releases of Linux. Over time, Torvalds secured the help of a number of individuals to help manage the process of building and releasing successive versions of Linux. (We discuss the Free Software Foundation and the production of open-source programs in more detail in the next chapter.)

The Linux kernel was organized as a collection of modules. Each module could be modified independently of the others, and new modules

could be added to the system. This facilitated improvements in Linux and also made it possible for independent programmers to make contributions.

Working versions of Linux consist of the Linux kernel, various applications, and specialized configuration and installation tools that together form a Linux distribution such as Red Hat Linux. The Linux kernel is the core of the operating system, managing things like memory and task scheduling. The system is often modified to fit specialized tasks such as deployment in specialized devices (mobile phones, set-top boxes, robots) or in unique circumstances (security).

Java

Sun initially envisioned Java, first called Oak, as a programming language for digital devices for television and other media. The idea was to create a language that was useful for devices that lacked the processing power and memory of larger computers. It was also aimed for use on devices based on different microprocessors. Java's programming team first displayed Java's potential prowess by creating a digital entertainment device with an animated touch screen and Duke, Java's mascot, doing cartwheels on the screen.[19] Over time, this new language has evolved into a platform that can be used on many computing devices. As of 2006, however, Java is most commonly used for small Web applications, in enterprises as a component of applications on servers, in set-top boxes, and on portable devices.

The platform consists of three major components. First, Java is a programming language. Second, the Java Virtual Machine is the middleware layer that is at the core of Java's operating system independence. Unlike other programming languages, Java programs are not written for a particular operating system or device. Instead, they run on a Java Virtual Machine that serves as an intermediary between the program and the underlying operating system.

The Java Virtual Machine specific to a given operating system and hardware environment then converts the byte codes into machine code.

19. http://www.engin.umd.umich.edu/CIS/course.des/cis400/java/java.html; http://newsletter.paragon-systems.com/articles/58/1/ja/8427.

Thus the same program, without modification, runs on every system for which a Java Virtual Machine is available. However, the portability comes with a price: performance. The Java Virtual Machine is essentially another layer of instructions that sits between the application and the hardware. Consequently, Java applications run slower and require more memory than comparable applications written specifically for any given operating system. As a result, Java is rarely used for complex or demanding applications, and the applications are heavily optimized for specific operating systems and processors when it is used. Such optimization hurts portability.

Finally, Java also includes class libraries for each operating system. The class libraries are the services provided by the APIs and form the backbone of Java. They are also a limitation on Java's platform independence. Even if a particular system has a Java Virtual Machine, a Java program will run only if all the class libraries it relies on are available on that system. That is seldom an issue for PCs, but smaller, more limited devices such as mobile phones may not have room for the full class libraries. The Java Virtual Machine and the class libraries are sometimes referred to as the Java Runtime Environment.

The architecture of the Java technologies for an Audiovox SMT5600 mobile phone was shown in Figure 2.3.[20] This device has a microprocessor from Texas Instruments and uses the Windows Mobile 2003 operating system. The phone also supports the Java 2 platform. An application developer can write his program in Java and rely on the Java class libraries for the equivalent of APIs and underlying application services. He can then compile this program into Java byte codes.

Operating System and Software Platform Production

Modern operating systems have tens of millions of lines of code. Ten million lines of code have roughly the same information as the *Encyclopædia Britannica*. The many sections of this code—objects or modules—are highly interdependent, much like the parts of a modern jet aircraft. There is a further challenge. Users want their applications to

20. http://www.mobiletechreview.com/audiovox_SMT5600_smartphone.htm.

work with successive generations of the platform. Changes in the platform must maintain "backward compatibility" as much as possible.

Anyone who has programmed knows that it is rarely easy to make a complex program work properly with all possible inputs. One must typically write code, run it, watch it fail, find and remove the error or bug that caused the failure, try again, and keep debugging until the program does what it was intended to do. With long, complex programs written by many people, this process becomes distinctly harder: not only does each programmer have to make sure her module works correctly, but those in charge of the program need to make sure that all of the modules work properly when fit together. Debugging is easiest with single-purpose programs. Operating systems and software platforms are harder because they are designed to support many other programs that cause the many elements of the operating system to interact in an enormously large number of ways. They need to go through a testing process even after they seem to run well in normal use. This process often continues to what is known as alpha and beta testing by users.

There are important differences between the development process for operating systems that are manufactured by companies—what is known as proprietary software—and those that are produced through the open-source process.

Proprietary Operating Systems

Sophisticated operating systems are developed over many months by large teams of designers, programmers, and testers. New versions of an operating system can take several years from conception to release. It took Apple five years to develop the Mac OS X, and this involved a complete redesign and rewriting of its earlier operating system. Microsoft Windows NT, mentioned earlier, was developed over five years by a team that grew to more than 200 designers, programmers and testers working for Microsoft—in addition to the over 15,000 alpha and beta testers who volunteered to try early versions of the program.[21]

The process of developing a new operating system begins with its architecture that, like a building, will determine the constraints

21. Internal Microsoft information, obtained by interview.

on remodeling efforts for years to come. Programming teams then begin writing the code for each individual module of the operating system. In Microsoft's case, this involves working in parallel but linking the pieces daily and debugging the resulting system.[22] It also entails constant testing. Microsoft employs roughly one tester for every programmer.

This description makes it seem as if writing software was like writing a newspaper—and in some ways it is. There are many people working in loosely formed teams whose output is combined every day into a single document. But the outputs of the software teams must work together. And while journalism has its own complexities, software developers have many hard mathematical problems to conquer in the process of writing an operating system. For instance, software manufacturers face complex mathematical problems in designing and implementing systems for maintaining and updating directories in networks. Between 1995 and 2004, IBM, Microsoft, Novell, Sun Microsystems, and other companies were granted more than 290 patents for their programming methods. There is also a premium on writing code that is efficient—that maximizes the performance of the computer's hardware.

Open-Source Software

Open-source communities face similar problems in dealing with large, complex software programs, but solve them quite differently.

A program begins when one or more individuals conceive its architecture. Often these individuals will write a first draft of the program, or a significant segment, and then post the code and key elements of the overall architecture on the Internet. They will form a self-perpetuating committee that will guide further development by accepting changes contributed by others to current versions. Debugging and testing takes place mainly through people—usually the information technology savvy, but in later stages also interested members of the general public—identifying problems and reporting them to a site that is dedicated to the program. The

22. Michael A. Cusumano and Richard W. Selby, *Microsoft Secrets* (London: HarperCollins, 1995), pp. 269–270.

open-source community prides itself on having "a thousand eyeballs" look at and use program code and thereby steadily improve its quality.

Linux was developed more or less in this way. Torvalds and his lieutenants have been coordinating its development ever since he released the first kernel. For example, since mid-2001, approximately 4,000 programmers have made contributions to Linux.[23] The code for the kernel has expanded from about 2.5 million lines to more than 4 million lines in the same period, and many utilities, libraries, and other software platform-related modules have been written for it.

The Linux development process is much more organized than it might seem. Many of the contributors are employed and paid by large corporations, such as IBM and Hewlett-Packard (HP). Patches and new ideas for individual modules are conceived and started by individual developers or contributing enterprises. After that, the project begins a life of its own, depending on the idea's popularity within the Linux community. These projects are posted on the Internet and are usually run under some direction. The one below, taken from the Linux home page in February 2005, shows a project for creating a new sound API. The home page for the project advertises for programmers to work on further development.

Name:	Advanced Linux Sound Architecture (ALSA) Project
Website:	http://www.alsa-project.org/
Contact:	perex@suse.cz
Description:	Primary goals are create modern sound driver for Linux with new sound API which solves all OSS/Lite trouble and create good libraries for sound applications.

The Website says: We need users to use, test and provide feedback, programmers to work on low level drivers, writers to extend and improve our documentation, and application developers who choose to use ALSA as the basis for their programs. If you are interested, please subscribe to a mailing list. We welcome all constructive ideas, opinions and feedback!

23. Josh Lerner, Parag Pathak, and Jean Tirole, "The Dynamics of Open Source Contributions," *American Economic Review Papers and Proceedings* 96 (May 2006): forthcoming.

The contact for the project was located at SuSE, a company that distributes open-source software, in the Czech Republic (SuSE is a German company that is owned by Novell). We return to the open-source movement and its origins in the next chapter.

Operating System, Software Platforms, and Computing Devices

Several computing devices are the focus of discussion in the remaining chapters. Table 2.1 shows the main operating systems and software platforms that are used for these devices. It also shows the different manufacturers that provide the hardware, the operating system, and the middleware. There is great variety. The remainder of this book documents this heterogeneity and explores the business, economic, and historical reasons behind it.

Two features of software platforms have wide ramifications. The provision of services through APIs makes them inherently two-sided platforms that serve users and developers simultaneously. The fact that software platforms are no more than written symbols means that, like books, movies, patents, and other intellectual property, these platforms often cost much to create and little to reproduce. The next chapter examines the economic implications of these and other characteristics of software platforms.

INSIGHTS

- Software platforms were made possible by the development and improvement of programming languages that enable humans to tell computers what to do.
- The design of software platforms and the business models they serve have important consequences for the structure of industries based on computing devices.
- Most software platforms are composed of modules that provide software services to other software programs. APIs provide application programmers access to these services. In effect, the programmer submits

Table 2.1
Summary of Platforms

Platform, Examples	Microprocessor	Operating System	Middleware
PC			
Microsoft Windows	Intel, AMD	Windows	Windows, Java, more
Apple Macintosh	POWERPC, Intel	Mac OS	Mac OS, Java, more
Linux	Intel, AMD, POWERPC, more	Linux kernel	Linux Utilities, Java, more
Game Consoles			
Sony PlayStation 2	300 MHz Emotion Engine	Sony proprietary	Sony PS2 SDK (limited), sound and graphics APIs
Microsoft Xbox	733 MHz Intel Pentium 3*	Windows 2000*	XDK, sound and graphics APIs
PDA			
Palm OS	Intel, TI, ARM, Motorola, more	Palm OS	Palm OS, Java
Microsoft Windows Mobile	Intel, TI, ARM, Motorola, more	Windows Mobile	Windows Mobile, Java 2 Micro Edition, Mophun
BlackBerry	Intel, Motorola, more	RIM proprietary	Java
Smart Phones			
Symbian	Intel, TI, ARM, Motorola, more	Symbian	Nokia Series 60/80, Java 2 Micro Edition, Mophun
Microsoft Windows Mobile	Intel, TI, ARM, Motorola, more	Windows Mobile	Windows Mobile, Java 2 Micro Edition, Mophun
Linux	Intel, TI, ARM, Motorola, more	Linux	Linux Utilities, Vendor SDKs, Java 2 Micro Edition
Digital Media			
Microsoft Windows Media	NA	NA	Windows Media Player
RealNetworks	NA	NA	RealPlayer
Apple QuickTime	NA	NA	QuickTime
Apple iPod	NA	iPod OS	

* Modified for the needs of the game console.

information to the API, and the software platform performs the service requested.

• The use of APIs enables software platform developers to write code that can be used by many applications vendors, thus reducing duplication of effort.

• Open-source software is built through a decentralized process whereby an initial version of a program is posted on the worldwide Web and anyone who is interested may propose additions or corrections.

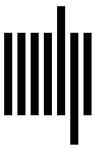
Invisible Engines

How Software Platforms Drive Innovation and Transform Industries

By: David S. Evans, Andrei Hagiu, Richard Schmalensee

Citation:

Invisible Engines: How Software Platforms Drive Innovation and Transform Industries
By: David S. Evans, Andrei Hagiu, Richard Schmalensee
DOI: 10.7551/mitpress/3959.001.0001
ISBN (electronic): 9780262272421
Publisher: The MIT Press
Published: 2008



The MIT Press

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please email special_sales@mitpress.mit.edu or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, MA 02142.

This book was set in Sabon by SNP Best-set Typesetter Ltd., Hong Kong.
Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Evans, David S. (David Sparks)
Invisible engines : how software platforms drive innovation and transform industries / David S. Evans, Andrei Hagiu, and Richard Schmalensee.
p. cm.
Includes bibliographical references and index.
ISBN 0-262-05085-4 (alk. paper)
1. Application program interfaces (Computer software). 2. Industries—Data processing. I. Hagiu, Andrei. II. Schmalensee, Richard. III. Title.
QA76.76.A63 E93 2006
005.3—dc22

2006046629

10 9 8 7 6 5 4 3 2 1