4 Programming languages and software engineering

> The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.
> Edsger Dijkstra[1]

The software crisis

The term *software engineering* originated in the early 1960s, and the North Atlantic Treaty Organization sponsored the first conference on the "software crisis" in 1968 in Garmisch-Partenkirchen, West Germany. It was at this conference that the term *software engineering* first appeared. The conference reflected on the sad fact that many large software projects ran over budget or came in late, if at all. Tony Hoare, a recipient of the Turing Award for his contributions to computing, ruefully remembers his experience of a failed software project:

> There was no escape: The entire Elliott 503 Mark II software project had to be abandoned, and with it, over thirty man-years of programming effort, equivalent to nearly one man's active working life, and I was responsible, both as designer and as manager, for wasting it.[2]

B.4.1. Fred Brooks made a major contribution to the design of IBM/360 computers. His famous book *The Mythical Man-Month* describes his experiences in software development.

In his classic book *The Mythical Man-Month*, Fred Brooks (B.4.1) of IBM draws on his experience developing the operating system for IBM's massive System/360 project. Brooks makes some sobering reflections on software engineering, saying, "It is a very humbling experience to make a multimillion-dollar mistake, but it is also very memorable."[3]

In this chapter we will explore two aspects of the way the software industry has addressed this crisis: the evolution of programming languages (Fig. 4.1) and the emergence of software engineering methodologies. We will see how two major ideas provide the basis for modern software development: (1) *structured programming*, in which the statements are organized in a specific way to minimize error; and (2) *object-oriented software*, which is organized around the

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

Fig. 4.1. Some popular programming languages. Wikipedia lists more than seven hundred programming languages but only a few of these have attained widespread use.

objects the programmer wants to manipulate rather than the logic required to do individual operations. To computer scientists, an *object* is any item that can be individually selected and handled. In object-oriented programming, an *object* consists of not only the data but also the *methods* employed to operate on that type of data. In addition, the introduction of engineering practices in the specification, design, coding, and testing of software has helped the software industry make progress toward taming the software crisis. However, it is important to remember that even with the best software engineering practices of today, software systems contain somewhere between ten and ten thousand errors per million lines of code. It is therefore not surprising that testing and bug fixing play a large role in the life cycle of software. Lastly, we shall take a look at an alternative model of software development. This model is based on *crowdsourcing*, which incorporates contributions from a large group of people, and the free sharing of the *source code*, the program instructions in their original form. Such sharing of source code is called the *open-source* approach. One of its major advantages is that it permits rapid finding of bugs. Eric Raymond has memorably summarized this idea as "Given enough eyeballs, all bugs are shallow."[4] In other words, the more people who see and test a set of code, the more likely any flaws will be caught and fixed quickly. Raymond called this "Linus's Law" after Linus Torvalds, creator of the open-source Linux operating system. However, other experienced software engineers would contest this statement!

Elements of modern programming languages

Before the introduction of FORTRAN in the 1950s, programmers had to work in machine code, made up of binary commands (Fig. 4.2), or in assembly languages, composed of symbolic expressions, both of which were difficult and time-consuming to write. FORTRAN was the first commercially successful *high-level language*. That is, it resembled natural human language, was easy to learn, and required less knowledge of the computer on which it would be run. Many people expected that FORTRAN would make software easy to create and simple to fix. It is amusing to look back at the optimism expressed in the original FORTRAN proposal from John W. Backus in 1954:

> Since FORTRAN should virtually eliminate coding and debugging, it should be possible to solve problems for less than half the cost that would be required without such a system. Furthermore, since it will be possible to devote nearly all usable machine time to problem solution instead of only half the usable machine time, the output of a given machine should be almost doubled.[5]

Fig. 4.2. In the early days of computing, programming was done in binary. It was only with the introduction of assembly language that programming became slightly more intuitive. High-level programming languages like FORTRAN and COBOL only became popular with the development of efficient compilers that produced performance close enough to assembly language programs.

So how did software production reach such a crisis point only a decade later? Part of the answer lies in the features of early languages.

We will begin by looking at some early programming languages and three concepts that led to better programming: (1) *type checking*, which checks and enforces certain rules to help prevent one major category of errors; (2) *recursion*, where a function can call itself; and (3) *dynamic data structures*. A *data structure* is any organized form, such as a list or file, in which connected data items are held in a computer. A dynamic data structure provides a way of organizing data that allows more efficient use of computer memory.

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

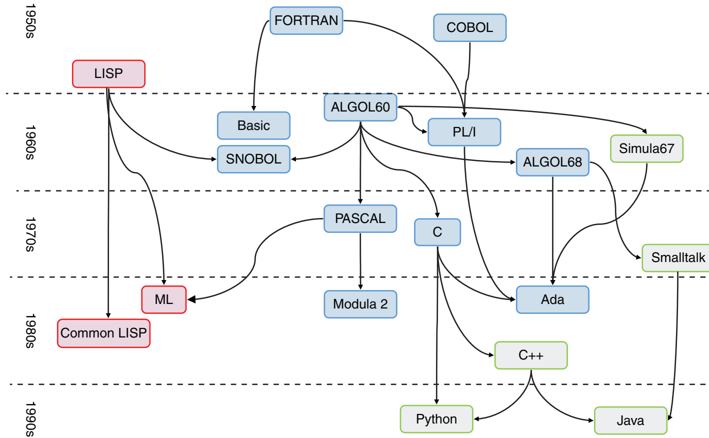


Fig. 4.3. A simplified evolutionary graph of popular programming languages. The language styles are marked by different color frames: red – declarative, blue – imperative (procedural), and green – object oriented.

Programming languages are artificial languages designed to communicate instructions to a computer. FORTRAN and most other programming languages consist of sequences of text, including words, numbers, and punctuation marks. Programming languages may be divided into two broad categories, *imperative* and *declarative* languages (Fig. 4.3). Roughly speaking, imperative languages specify *how* a computation is to be done, while declarative languages focus on *what* the computer is supposed to do. FORTRAN was the first commercial example of an imperative language. A year later, in 1958, John McCarthy and his students at the Massachusetts Institute of Technology (MIT) developed LISP, standing for LISt Processing, a programming language targeted at artificial intelligence (AI) applications. LISP was the first attempt at creating a type of declarative language in which computation proceeds by evaluating functions.

In computer programming, a *variable* is a symbol or name that stands for a location in the computer's memory where data can be stored. Variables are important because they enable programmers to write flexible programs. Rather than putting data directly into the program, the programmer can use variables to represent the data, allowing the same program to process different sets of information, depending on what is stored in the computer's memory. Instructions known as *declaration statements* specify the sort of information each variable can contain, called the *data type*.

Many of the main ideas of the original FORTRAN language are still used in programming languages today. Features of early versions of FORTRAN were:

- Variable names in a FORTRAN program could be up to six characters long and could change their values during execution of the program.
- Variable names beginning with the letters I, J, K, L, M, or N represented *integers*, that is, whole numbers with no fractional parts. All other variables represented *real numbers*, which could be any positive or negative numbers, including integers.
- Boolean variables, variables that have the value of either true or false, could be specified with a *logical* declaration statement.
- Five basic arithmetic operations were supported: + for addition; – for subtraction; * for multiplication; / for division; and ** for exponentiation, raising one quantity to the power of another.

Downloaded from http://www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http://www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

Another important feature of FORTRAN was that it introduced a data structure called an *array* that was especially useful for scientific computations. An array is a group of logically related elements stored in an ordered arrangement in the computer's memory. Individual elements in the array may be located using one or more indexes. The *dimension* of an array is the number of indexes needed to find an element. For example, a list is a one-dimensional array, and a block of data could be a two- or three-dimensional array. An instruction called a *dimension statement* instructs the compiler to assign storage space for each array and gives it a symbolic name and dimension specifications. For example, a one-dimensional array of numbers called a *vector* may be specified by the dimension statement VEC(10) or a three-dimensional field of values by MAGFLD(64, 64, 64).

Modern programming languages have improved on the minimal specification of data types in FORTRAN. Many languages today employ *strong typing* – strict enforcement of type rules with no exceptions. Checking that a large, complex software system correctly uses appropriate data types greatly reduces the number of bugs in the resulting code.

For efficiency reasons, the development of FORTRAN was closely aligned to the hardware architecture of the computer. Thus FORTRAN *assignment statements*, instructions that assign values to the variables, are actually descriptions of how the data moves in the machine. In FORTRAN, assignment statements are written using the "equal" sign, but these statements do not mean mathematical equality. Consider the assignment statement

$$A = 2.0 * (B + C)$$

In FORTRAN, this statement means, "Replace the value at address A by the result of the calculation 2.0*(B+C)." Similarly, the odd-looking statement

$$J = J + 1$$

means, "Read the value from address J, add 1 to this value, and then store the result at the same address."

The original FORTRAN specification provided three kinds of *control statements*, instructions that switch the computer from automatically reading the next line of code to reading a different one:

- The DO statement was used to allow the program to perform a *loop*, a sequence of instructions that repeats either a specified number of times or until a particular condition is met.
- The IF statement was employed for *branches*, instructions that tell the computer to go to some other part of the program, depending on the result of some test.
- The GO TO statement was used to direct the computer to jump to a specific numbered statement.

Forms of the basic *do* loop and *if* statement constructs are available in most modern programming languages. We can write the general form of these control statements in *pseudocode*, an informal, high-level description of a program that is intended to be read by humans. A computer could not directly execute

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

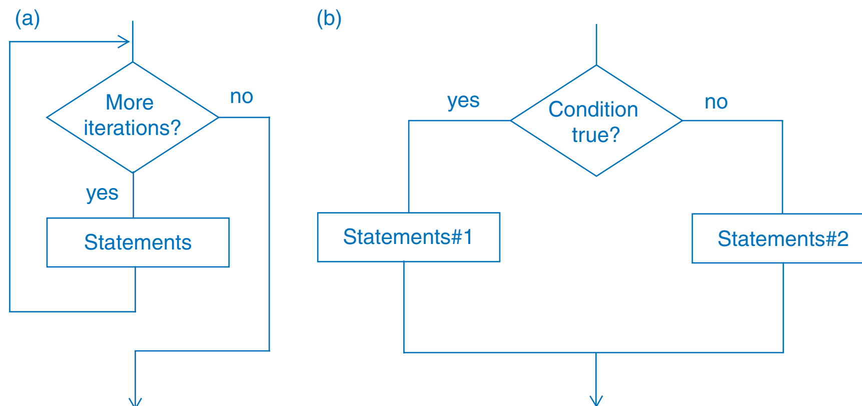


Fig. 4.4. Flowcharts for the "for" loop (a) and "if-then-else" (b). They illustrate the decision points in the program where a branch can be made depending on a check of some condition. The "for" loop performs a set number of iterations through the same piece of code that use different data. The "if-then-else" branch executes a different piece of code depending on the result of a test condition.

it, however, because the pseudocode version of the program omits details not essential for understanding by people but necessary for a machine to run the program. Thus we can write a *do* loop using the *for* keyword

```
for <var> in <sequence>
    <statements>
```

The variable <var> after the keyword *for* is called the *loop index*, and the statements in the body of the *for* loop – <statements> – are executed according to the number of times specified in the <sequence> portion of the loop heading. After completing the loop the required number of times, the program goes to the program statement after the last statement in the body of the loop.

Similarly, we can write an *if* statement as:

```
if <condition>:
    <statements#1>
else:
    <statements#2>
```

If the Boolean condition <condition> is true, the program executes the first set of statements – <statements#1> – and then jumps to the statement after the second block of statements – <statements#2>. If the condition is false, the program jumps over the first block of statements and executes the second block of code <statements#2> under the *else* keyword, before carrying on with the next statement of the program. It is sometimes helpful to visualize these control structures as diagrams called *flowcharts*, first introduced into computing by Herman Goldstine and John von Neumann in 1947. Flowcharts use boxes to represent the different parts of the program with arrows between the boxes showing the sequence of events. Flowcharts representing the *for* loop and the *if-then-else* flowcharts are shown in Fig. 4.4.

Modern programming languages generally do not encourage use of the *go to* statement. Undisciplined use of *go to* statements often led to very complex "spaghetti" code that was difficult to understand and debug. In 1968, Edsger Dijkstra (B.4.2) wrote a famous article titled "Go To Statement Considered Harmful":

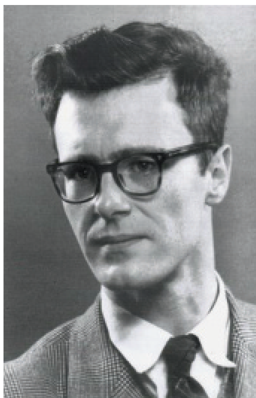
Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

> For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code).[6]

Criticism about the undisciplined use of *go to* statements was one of the factors that led to the *structured programming* movement of the 1970s. Structured programming aims to improve the clarity, quality, and development time of software by identifying coherent blocks of code and using *subroutines*, standard sets of instructions to perform frequently used operations within a program. Structured programming uses *for* loops and *if-then-else* constructs as the only control and decision structures.

Recursion and dynamic data structures

Since the beginning of computer programming, most programming languages have been imperative languages that tell the machine how to do something. It is remarkable, however, that one of the earliest high-level programming languages to be developed was a declarative language, which told the computer what the programmer would like to have happen and let the machine determine how to do it. This declarative language was LISP, the brainchild of John McCarthy, who had spent the summer of 1958 as a visiting researcher in the IBM Information Research Department. In addition to becoming frustrated with batch processing and wanting to pursue time-sharing systems, he investigated the requirements for a programming language to support symbolic rather than purely numeric computations. These requirements included *recursion*, the ability



B.4.2. Edsger Dijkstra (1930–2002) was one of the pioneers of computer science. From the early days he was championing a mathematically rigorous approach to programming. In 1972 he received the Turing Award for his fundamental contributions to the development of programming languages. He was well known for his forthright opinions on programming and programming languages.

On FORTRAN:

> FORTRAN, "the infantile disorder", by now nearly 20 years old, is hopelessly inadequate for whatever computer application you have in mind today: it is now too clumsy, too risky, and too expensive to use.[B1]

On COBOL:

> The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense.[B2]

On BASIC:

> It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.[B3]

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

for a program or subroutine to *call* itself, and *dynamic data structures*, for which the memory space required is not set in advance but can change size as the program runs. The newly developed FORTRAN language did not support either of these requirements.

When he returned to MIT, McCarthy set about devising a language suitable for the types of symbolic and AI applications he had in mind. His LISP language focused on the manipulation of *dynamic lists* that could grow or shrink as the program was running. The style of programming was very different from that of an imperative language like FORTRAN. There were no assignment statements in LISP and no implicit model of state in the background, tied to the physical implementation of a "pigeonhole" model of computer memory. Instead, in LISP everything is a mathematical function. For example, the expression x^2 is a *function*: applying this function to the variable x returns the value of x^2. In LISP, computation is achieved by just applying functions to arguments. This feature makes it easier to reason mathematically about the behavior and correctness of LISP programs. It removes the possibility of dangerous "side effects" – situations in an imperative program where, unsuspected by the programmer, some part of the program has altered the value of a variable in memory from the value that the programmer had intended. Memory allocation for McCarthy's dynamic lists was not performed in advance of running the program. It was therefore necessary to regularly clean up the memory space by removing from the list of assigned storage locations any locations that were no longer being used. McCarthy's team called this process of reclaiming memory space *garbage collection*. These ideas have been influential in the implementation of modern languages like Java and C# (C Sharp).

McCarthy's ideas about recursion and dynamic data structures are not limited to declarative languages like LISP. They are now essential components of almost all imperative languages, such as FORTRAN, BASIC, and C. We can illustrate the idea of recursion through the problem of calculating the factorial of the number *n* – that is, the product of the whole number *n* and all the whole numbers below it. The factorial of *n* is written as *n!* and is defined to be the product n × (n – 1) × (n – 2) . . . × 2 × 1. We could calculate this using a *for* loop, but we can also perform the calculation using recursion, a process in which the function repeatedly calls itself with smaller and smaller arguments until it reaches a *termination condition*, a state in which it will stop calling itself. The pseudocode for the recursive calculation of factorial *n* is:

```
factorial (n)
    if n <= 1:
        return 1
    else:
        return n * factorial (n – 1)
```

The expression "n <= 1" is an integer check to see if the value of n is less than (<) or equal (=) to 1. The calculation of the factorial starts with *n* and multiplies this by factorial (n – 1). This repetition continues until (n – 1) equals 1.

As an example of dynamic data structures we consider *linked lists*. In scientific calculations, the size of the array structures that store related groups of

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

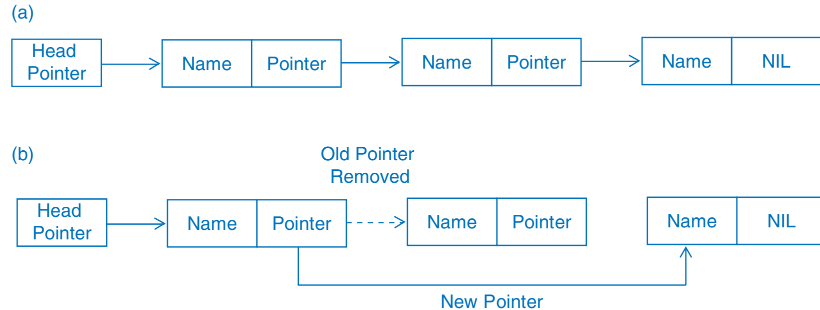


Fig. 4.5. A linked list is an example of a dynamic data structure that makes it easy to remove or add items to a list. (a) This shows the structure of a linked list with each entry in the list having a *pointer* to the memory location of the next element. (b) This illustrates how easy it is to delete a node from the linked list by just changing the pointer.

items can usually be specified in advance. The same is not true for most types of lists. The membership list for a sports club, for example, will grow and shrink as new members join and old members leave. If we store a list of names in an array of fixed length using sequential memory locations in the computer, removing or adding names becomes very laborious because the data in the list must be frequently reshuffled to keep them in the right order in memory. These problems can be avoided if we do not store items in the list in a sequential memory block but just in any convenient area of memory. To store the contents of the list, each name in the list is stored in some location along with a *pointer* – a memory address – to the location of the next name on the list (Fig. 4.5a). Deleting or adding names is now straightforward because it just requires changing a single pointer (Fig. 4.5b). With pointers, the address of the data storage location can be kept separately from the actual data. Retrieving the data is thus a two-step process – getting the address of the storage location and then going to that location to get the data. Other common dynamic data structures are LIFO (Last-In-First-Out) stacks, collections of items in which only the most recently added item may be removed, and FIFO (First-In-First-Out) queues, collections of items in which only the earliest added item may be removed.

Programming with objects: from SIMULA to C++

An important idea in object-oriented programming is *data abstraction*, which focuses on classes, objects, and types of data in terms of how they function and how they can be manipulated, while hiding details of how the work is carried out. The idea of data abstraction can be traced back to two Norwegian computer scientists, Kristen Nygaard and Ole-Johan Dahl (B.4.3). They first presented their programming language SIMULA 67 in March 1967. They were interested in using computers to run simulations and needed the language to support subprograms that could stop and later restart at the place they had stopped. To do this, Nygaard and Dahl introduced the idea of a *class*. The key property of a class is that a data structure and the routines that manipulate that data structure are packaged together. This led to the important idea of *abstract data types*, sets of objects that share a common structure and behavior.

B.4.3. Ole-Johan Dahl (1931–2002) (left) and Kristen Nygaard (1926–2002) were first to introduce classes and objects in their Simula programming language.

Abstract data types were actually present in the original FORTRAN language. There was a built-in *floating-point* data type, which represented

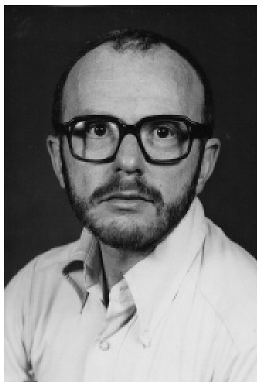
Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

numeric values with fractional parts, and a set of arithmetic operations that were allowed to act on floating-point variables. How FORTRAN included these data types is an example of one of the key ideas of data abstraction, namely, *information hiding*. The details of the actual way a floating-point number is represented in the computer are hidden from the user and cannot be accessed by the programmer. In addition, the programmer can only create new operations on this type of data by using the built-in operations already supported on floating-point variables. Because of such information hiding, FORTRAN programs could run on many different machines, even though floating-point variables were frequently implemented very differently on different machines.

Modern object-oriented (O-O) programming languages allow programmers to create their own abstract data types. Consider again the problem of writing a program to manipulate a list containing the names of members in a sports club. In an imperative programming language, the list is just a collection of data and we need to write separate software procedures to add, delete, and sort items in the list. In an O-O language, the list is constructed as an *object* consisting of both the list data and the collection of procedures – called *methods* in O-O speak – for manipulating this data. Thus an O-O program to sort the list would not contain a separate sorting procedure but make use of the methods already built in for the list object. What is the relationship between a class and an object? A *class* is a template for all the objects with the same data type and methods. The list class applies to all list objects with data in the form of a list and the methods to operate on the list. As a slightly more complicated example, let us define a bank account class. The abstract data type for a bank account consists of the name of the client, the number of the account, and the balance of money in the account. The class consists of bank account data of this type plus methods that define the different operations that can be carried out on the account – withdrawals, deposits, transfers, and so on. Accounts belonging to different customers obviously contain different data and are called *objects* or *instances* of this class. The methods that act on the data within an object are usually small imperative programs.

Two other important properties of O-O languages are *inheritance* and *encapsulation*. The idea of inheritance is that a class can be extended to create another class that inherits the properties of the original class. Thus the class “bank account” could be extended to create a new class “savings bank account” that inherits the same data structure and methods as the original class but with additional properties and methods (Fig. 4.6). Encapsulation means that there are certain properties of an object that are not accessible to other parts of the program. Only the object is able to access these properties.

Canadian computer scientist David Parnas (B.4.4) was one of the pioneers of information hiding. Turing Award recipient Alan Kay and his research team at the Xerox PARC (Palo Alto Research Center) in Silicon Valley in the 1970s first introduced the term *object-oriented programming*. They developed the Smalltalk language, which was based on the idea of building programs with objects that communicated by sending messages.



B.4.4. David Parnas is a Canadian computer scientist who pioneered ideas of “information hiding.” These ideas are now an integral part of data abstraction in object-oriented programming.

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

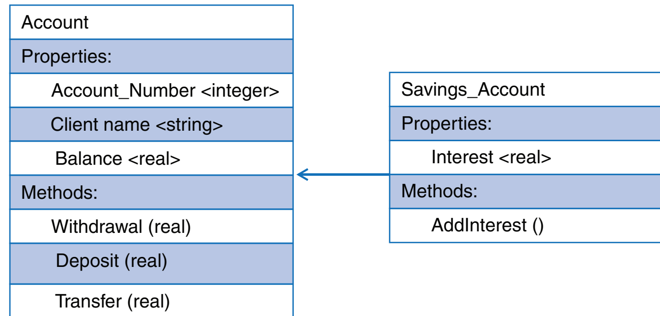
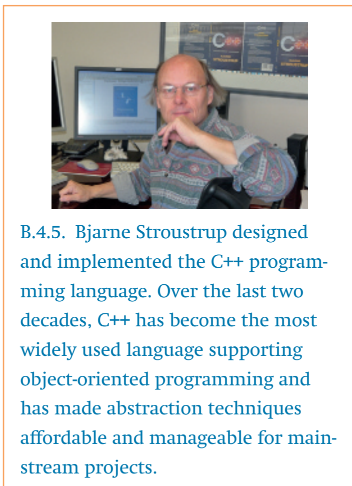


Fig. 4.6. This figure illustrates the concept of class inheritance. The properties of the Account class are the data items. The methods represent the actions that we can carry out on the data. For the Savings_Account class, in addition to the properties and methods inherited from the Account class, there is also a new data item called interest and a new AddInterest method. In this way we can construct more complex classes from simpler ones.

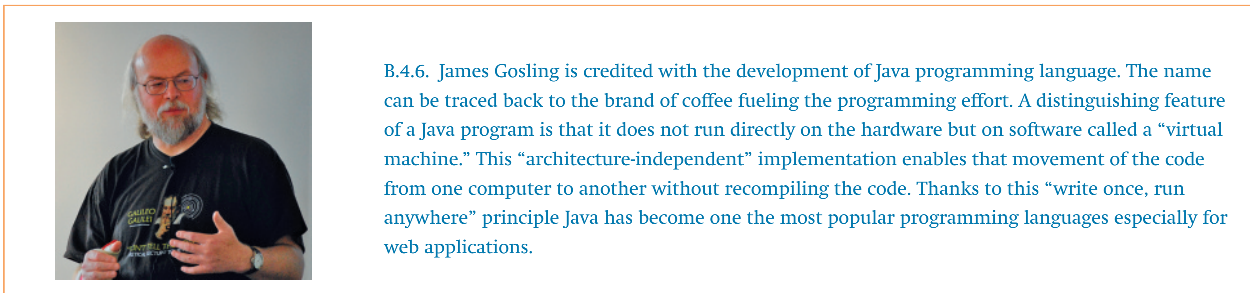
One of the most widely used O-O languages today is the C++ programming language. Bell Labs researcher Bjarne Stroustrup (B.4.5) was familiar with SIMULA and had found the class feature to be useful in large software development projects. When he started working at Bell Labs, he explored ways of enhancing Dennis Ritchie's C language, which was both fast and portable (see section on C and Unix at the end of this chapter). In 1979, Stroustrup started by adding classes to C to create what he called "C with Classes." Over the next few years, Stroustrup added several other features and renamed the language "C++." There is now a C++ software library called the Standard Template Library (STL) that contains predefined, useful classes that are provided as part of the C++ programming environment. By incorporating the STL library of classes into a program, the programmer does not have to explicitly specify these data structures. Two other examples of widely used object-oriented programming languages are Java (B.4.6) and C#.



B.4.5. Bjarne Stroustrup designed and implemented the C++ programming language. Over the last two decades, C++ has become the most widely used language supporting object-oriented programming and has made abstraction techniques affordable and manageable for mainstream projects.

Why do we need software engineering?

As we have seen, computer scientists originally hoped that programming in a high-level language would, as Backus said, "virtually eliminate coding and debugging."[7] For small scientific programs written by one or two researchers, programming certainly became much easier, with the hard work of converting a FORTRAN program into efficient assembly code delegated to a computer program, the compiler. However, many scientific programs nowadays are complex simulation codes incorporating many different aspects of the problem under investigation. Writing and debugging such programs has become much more



B.4.6. James Gosling is credited with the development of Java programming language. The name can be traced back to the brand of coffee fueling the programming effort. A distinguishing feature of a Java program is that it does not run directly on the hardware but on software called a "virtual machine." This "architecture-independent" implementation enables that movement of the code from one computer to another without recompiling the code. Thanks to this "write once, run anywhere" principle Java has become one the most popular programming languages especially for web applications.

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

difficult. An additional challenge is that new researchers, who were not the authors of the original program, may need to extend and modify the code.

Similarly, as the software for business applications became larger and more complicated, it was no longer possible for a small team of talented programmers to write all the code. Teams of hundreds or even thousands of programmers now work on software systems consisting of hundreds of thousands or millions of lines of code, and programmers now have to coordinate their work. Accurate estimates of the time and cost of writing a complex software system have become vitally important for software companies. Brooks discusses these issues in his book on the "mythical" man-month unit of programming effort. As a result of his experience, he formulated "Brooks Law," which says, "Adding manpower to a late software project makes it later."[8]

Software companies have an urgent need for reliable answers to the questions: How many lines of code will it take to provide the desired functionality? How many programmers will be needed? How long will it take? If a software project is behind schedule, what should you do? Software engineering attempts to define methodologies and frameworks to answer these questions. The Institute of Electrical and Electronics Engineers (IEEE), a professional organization devoted to promoting technological innovation, defines software engineering in its Standard 610.12 as "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."[9]

One of the earliest attempts to apply engineering methodologies to software development was the "waterfall" model (Fig. 4.7). This identified four distinct phases of software development – requirements analysis, design, implementation, and testing. The waterfall method calls for completing each phase before proceeding to the next, which requires the systematic description and documentation of both the requirements and the design of the software to be completed before any actual coding begins. In practice, the phases are rarely completely separate. Software developers often find in later phases that they must go back and change things in earlier phases. Backtracking and multiple versions of each phase are common. David Parnas says about the design process:

> Even if we knew the requirements, there are many other facts that we need to know to design the software. Many of the details only become known to us as we progress in the implementation. Some of the things we learn invalidate our design and we must backtrack.[10]

The recognition that software development is not a linear process has led to a philosophy called *agile software development*. Agile methods break the task of writing the whole system into smaller segments or "sprints," each of which involves all the four phases of software development – analyzing requirements, designing, implementing, and testing. Based on the results of testing the latest version of a design, the developers make changes and improvements. The sprints typically last around four weeks, and the goal is to have a working prototype with some of the functionality required of the final product by the end of each sprint. One of the main motivations for these more flexible software engineering methodologies is that customers often do not know all their

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

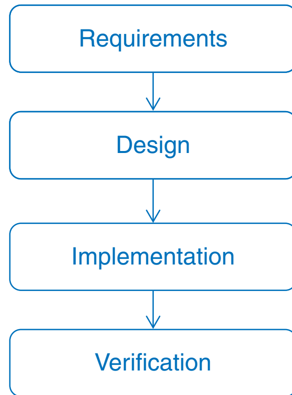
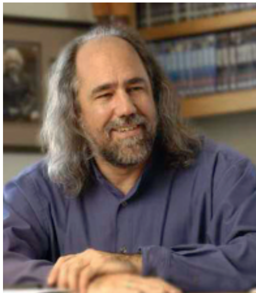


Fig. 4.7. The waterfall model is one of the earliest methods used to systematize software development. In principle, the model consists of several independent stages with each stage feeding to a subsequent stage. This approach is sometimes referred to as "Big Design Up Front" – because a new stage can only start if the preceding one has been fully completed. This is the strength of the method – but also its weakness. In reality, software development is not a linear process and many issues cannot be foreseen until later stages in the project. So, in practice, the individual stages are not fully isolated from each other: often we need to backtrack to make revisions and changes in the previous stages.

requirements at the beginning of a project. To incorporate changes in requirements makes a less rigid approach than the formal approach of the waterfall model essential.

The first stage of the software life cycle is requirements analysis and specification. One of the earliest tools for documenting computer programs was the *flowchart*, a diagram representing the sequence of operations in a program. We have already seen examples of flowcharts for *for* loops and *if-then-else* control statements (see Fig. 4.4). However, in the production of large, complex software systems, flowcharts have proved to be of limited value. In the 1980s, computer scientist David Harel was working with avionics engineers trying to specify the behavior of a software system to control a modern jet aircraft. An avionics system is *reactive* – a term coined by Harel and his colleague the late Amir Pnueli – in the sense that it has to respond predictably to a wide variety of different types of events. Harel eventually converged on a diagrammatic way to specify the responses and transitions of the avionics system, which he called *statecharts* – "the only unused combination of 'state' or 'flow' with 'chart' or 'diagram.'"[11] By 1986, Harel and his colleagues had built the *Statemate* tool, which not only allowed users to construct statecharts but was also able to automatically generate code to fully execute them. In the 1990s, they developed an O-O version of statecharts, which later became the heart of the Unified Modeling Language, or UML. UML was devised in 1996 by Grady Booch (B.4.7), James Rumbaugh, and Ivar Jacobson. It is a collection of visual languages for specifying, constructing, and documenting complex software designs. Booch comments, "If you look across the whole history of software engineering, it's one of trying to mitigate complexity by increasing levels of abstraction."[12] The UML approach (Fig. 4.8) is yet one more attempt to reduce the complexity of software production.



B.4.7. Grady Booch is an evangelist of the systematic approach to software design. In one of the interviews he referred to his mission with the following words: "if I had not discovered software I would have been a musician or a priest." He is one of the authors of the UML, which represents a framework for constructing and reasoning about the software. UML is a collection of diagrams and tools that allows programmers to cope with complex systems by raising the level of abstraction.

The final phase of the software life cycle is testing and maintenance. For complex software systems, it is impossible to test all branches of the code under all possible combinations of input data and initial states. According to Dijkstra, "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."[13] A 2002 report from the National Institute of Standards and Technology, a U.S. government agency that works to promote innovation and industrial competitiveness, estimated that inadequate software testing cost the U.S. economy nearly $60 billion per year. The report also stated, "In fact, the process of identifying and correcting defects during the software development process represents approximately 80 percent of development costs."[14]

Testing a modern software system involves the application of a variety of different tools. *Dynamic software testing* involves running the code using a set of

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

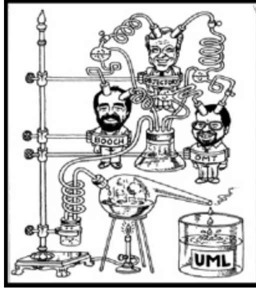


Fig. 4.8. UML is a general methodology that allows a systematic "step-by-step" approach to software design. UML is a unification of three software techniques developed by Grady Booch (Booch), Ivar Jacobson (Objectory), and James Rumbaugh (OMT); although UML also has roots in David Harel's statecharts. In software engineering circles Booch, Jacobson, and Rumbaugh are often referred to as the "three amigos." UML diagrams allow evaluation of various implementation options prior to actual program coding.

test cases. *White-box* testing is designed to test the internal structures of a program. The tester attempts to choose sets of inputs that exercise all the different possible paths through the code. *Black-box* testing takes the view of the user rather than the software developer. The tester checks the software's functionality with no knowledge of the system's internal structure. Another important type of evaluation is *fuzz testing*, in which valid input data sets are modified with random mutations and then fed into the program. Providing such invalid and unexpected inputs to the system allows the tester to determine how a program handles *exceptions* – unpredictable conditions or situations that can cause a program to crash or possibly create a security risk in the software (Fig. 4.9).

Empirical software engineering

The increasing complexity of modern software development is indicated by the numbers of programmers and lines of code in three releases of the Microsoft Windows operating system (Fig. 4.10). Here the programmers are divided into "developers," who write the code, and "testers," who systematically check the code for bugs. To allow such large numbers of programmers to work on different parts of the software system simultaneously, Microsoft developed a *synchronize-and-stabilize* approach to writing software. Breaking up the software into several different "branches" that can be worked on at the same time allows "large teams to work like small teams." Much of the complexity now lies in the process of correctly joining the branches back together. Microsoft solved the problem using "daily synchronizations through product builds, periodic milestone stabilizations, and continual testing."[15] Microsoft also developed an error-reporting tool so that users could inform the company of any software problems. Analysis of the data led to some interesting conclusions, as summarized by former Microsoft CEO Steve Ballmer:

> One really exciting thing we learned is how, among all the software bugs involved in reports, a relatively small proportion causes most of the errors. About 20 percent of the bugs cause 80 percent of all errors, and – this is stunning to me – one percent of bugs cause half of all errors.[16]

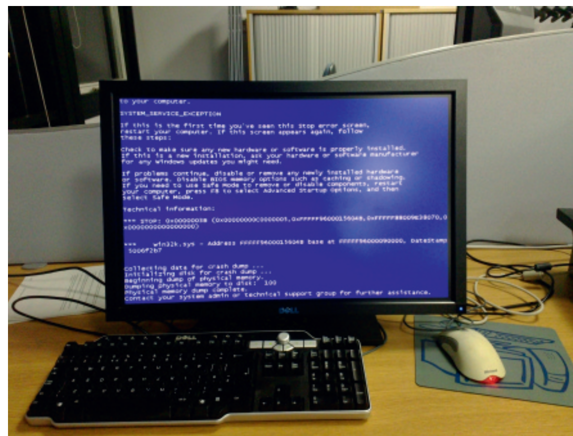


Fig. 4.9. A screenshot of the dreaded moment when a computer crashes. In programmer circles such an event is known as "the blue screen of death."

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

Fig. 4.10. Size and scale of the programming teams and code base for versions of the Microsoft Windows operating system.

Ship Date	Product	Development Team Size	Test Team Size	Lines of Code (LOC)
July 1993	Windows NT	200	140	5 million
December 1999	Windows 2000	1,400	1,700	30 million
October 2001	Windows XP	1,800	2,200	40 million

In his book *Code Complete*, Steve McConnell estimates the extent of the bug problem:

> Industry average experience is about 1–25 errors per 1000 lines of code for delivered software. The Applications Division at Microsoft experiences about 10– 20 defects per 1000 lines of code during in-house testing and 0.5 defects per 1000 lines of code in released product.[17]

The problem of deciding which bugs to fix and which are likely to generate new errors is complex. This is an area where the new field of empirical software engineering aims to help. The *Journal of Empirical Software Engineering* says:

> Over the last decade, it has become clear that empirical studies are a fundamental component of software engineering research and practice: Software development practices and technologies must be investigated by empirical means in order to be understood, evaluated, and deployed in proper contexts. This stems from the observation that higher software quality and productivity have more chances to be achieved if well-understood, tested practices and technologies are introduced in software development. Empirical studies usually involve the collection and analysis of data and experience that can be used to characterize, evaluate and reveal relationships between software development deliverables, practices, and technologies.[18]

This statement has now been adopted as part of the manifesto of the International Software Engineering Research Network.

One example of this empirical approach to software engineering is the CRANE tool developed by researchers at Microsoft – where CRANE is an acronym formed from Change Risk ANalysis and impact Estimation. The CRANE project looked at the challenges of providing support for multiple versions of Windows, running on a wide variety of computers, with a user base of more than a billion. One immediate challenge is that software maintenance for a released product is done by different teams of software engineers than those who developed the software. The goal of the CRANE project was to use historical information about the software being serviced to build risk-prediction models using advanced statistical techniques that could guide bug fixing and testing. For every bug in any software component, the tool provides the following information: what has happened to the component so far in servicing; what exactly is being changed with the proposed fix; which fixes carry more than average risk of causing more bugs; which tests to run after the change; which other components to test in addition to the changed component; and

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

which applications are potentially impacted by the change. Such empirical software engineering tools are enabling maintenance software engineers to make informed, data-driven decisions about their priorities.

Open-source software

A very different model of software development is the philosophy promoted by the *open-source software* movement. One of the origins of this movement was the decision of AT&T to allow the distribution of the Unix source code under a "free" license (see section on Unix and C). Bell Labs researchers Ken Thompson and Dennis Ritchie wrote the Unix operating system in the early 1970s. It was the first operating system to be written in a high-level language, the C programming language developed by Ritchie. The source code for the C compiler developed by Bell Labs researcher Stephen Johnson was also freely distributed with the Unix code. For only a few-hundred-dollar licensing fee, the university research community could obtain not only a functional operating system but also a platform for teaching and research. In 1956, AT&T had settled an antitrust monopoly suit with the U.S. Department of Justice, and AT&T's lawyers interpreted the agreement as forbidding the company to enter new markets not related to telephones. The AT&T license agreement for Unix was intended to make it crystal clear that the company was not creating a new business with computers:

> The terms of the early Unix licenses were minimal: The software came "as is" with no royalties to AT&T, but also no support and no bug fixes.[19]

One immediate result of this license agreement was to encourage the research community to set up self-help networks and share information on bug fixes. This set the style for the development of a global Unix support and development community with developers freely sharing their suggested code changes. The most significant research collaboration focused on Unix was between the original Bell Labs team of Richie and Thompson and the Computer Systems Research Group (CSRG) at the University of California, Berkeley. In 1983, the CSRG team released the latest version of their "Berkeley Unix" software, known as 4.2 BSD. This software incorporated the new Internet protocols and allowed Unix systems to be easily connected to the rapidly growing Internet. The initials BSD stand for Berkeley Software Distribution, which included an open-source software license

B.4.8. Richard Stallman is the originator of the free software movement. In 1979 he was working in the AI lab at MIT when the lab installed a new laser printer from Xerox. The printer suffered from paper jams and Stallman wanted access to the source code of the printer driver so he could modify it and fix the problem. Xerox would not give him the source code and he ended up being very frustrated. In 1984 Stallman resigned from MIT to set up the Free Software Foundation. Stallman was very explicit in his explanation of "free": "Since free refers to freedom, not to price, there is no contradiction between selling copies and free software."[B4] He called his project to build a free operating system by the recursive acronym GNU – standing for GNU's Not Unix. He also devised the GPL source license that was designed to ensure that any modifications to the source code were covered by the same license, including combinations of GPL software with commercial software.

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

that allowed free use of the source code. Importantly, the license allowed the possibility of incorporating all or part of the Berkeley software in a closed-source commercial product. The license only required that any copyright notices in the code were maintained along with the disclaimer of any warranty.

The 1980s were a confusing time for the Unix community. By this time, AT&T had realized that Unix was a very valuable software product and, under the terms of a new antitrust settlement in 1984, the company began charging for the Unix software. By 1992, friction between AT&T's new, commercially focused Unix Systems Laboratories division and the freewheeling Berkeley open-source community had come to a head. AT&T began a court case against the University of California. In addition to these legal problems, many different and incompatible variants of Unix had been spun-off – "forked" – from the original open-source Unix code, leading to a very fragmented Unix development community. Meanwhile, at MIT, a software developer in the AI lab named Richard Stallman (B.4.8) had become concerned about the loss of community that happened when software could not be freely shared. In 1984, Stallman founded the Free Software Foundation (B.4.9) with the goal of developing "an entirely free operating system that anyone could download, use, modify, and distribute freely."[20] He named his project GNU, standing for "GNU's Not Unix." To ensure that the source code remained open and freely shareable, Stallman devised the GNU Public License (GPL) that is very different from the permissive Berkeley BSD open-source license. The GPL license requires that any modifications of the software must be released under the same GPL open-source license. More important for commercial software companies was the "viral" requirement that any software formed by combining free, GPL-licensed software with commercial software must all be released under a free GPL license. Under the GNU umbrella, Stallman created some very popular tools for writing software that are still widely used by the computer science community – the *GNU Emacs* text editor, the *GCC* compiler, and the *GDB* debugger. However, it was left to a young Finnish graduate student named Linus Torvalds (B.4.10) to reunite the Unix community around his version of the Unix kernel, the core component of the Unix operating system.

In 1991, Torvalds was a graduate student at the University of Helsinki and had bought himself a new personal computer (PC) based on Intel's 386 microprocessor. Because he wanted to run Unix on his PC, he bought and installed Minix, a version of Unix suitable for teaching that had been created by Andy Tanenbaum at the Vrije Universiteit in Amsterdam. Inspired by the Minix software, Torvalds started creating his own version of the Unix kernel for the PC.

B.4.9. Hal Abelson is a professor of electrical engineering and computer science at MIT. He is passionate about both open-source software and open courseware, and has been a champion for the right to open access for publicly funded research publications. Abelson was one of the founders of the Free Software Foundation and the Creative Commons movements. In addition, Abelson has long believed in the potential for using computation as a conceptual framework in teaching. He is the author of several influential textbooks and implemented the Logo programming language on Apple II computers. Logo is widely regarded as one of the best programming languages for introducing computing to children. His pioneering work in education was recognized in 2012 by his receiving the ACM SIGCSE Award for Outstanding Contributions to Computer Science Education.

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

Fig. 4.11. Linux celebrates twenty years with release 3.0. Tux is the official mascot of the Linux community. According to legend, Torvalds was looking for something fun and sympathetic to associate with Linux, and a slightly fat penguin sitting down after having had a great meal perfectly fit the bill.

In 1991, he made the source code of his new operating system, called Linux (Fig. 4.11), available on the Internet with the following announcement:

> I'm working on a free version of a Minix look-alike for AT-386 computers. It has finally reached the stage where it's even usable (though it may not be, depending on what you want), and I am willing to put out the sources for wider distribution.… This is a program for hackers by a hacker. I've enjoyed doing it, and somebody might enjoy looking at it and even modifying it for their own needs. It is still small enough to understand, use and modify, and I'm looking forward to any comments you might have. I'm also interested in hearing from anybody who has written any of the utilities/library functions for Minix. If your efforts are freely distributable (under copyright or even public domain) I'd like to hear from you so I can add them to the system.[21]

Torvalds was surprised by the response to his invitation from the worldwide Unix community. Within a couple of years, hundreds of developers had joined his Internet newsgroup and were contributing bug fixes, improvements, and new features to Linux. By 1994, Torvalds was able to release the first complete version of his operating system, Linux version 1.0. This listed nearly eighty developers as contributors, from a dozen different countries. From these modest beginnings, Linux has become much more than a hobbyist's PC operating system. By 1999, Red Hat and VA Linux were established as public companies offering "Linux support" – although the basic code was still freely available. By 2000, Linux had received official recognition from IBM, which announced it would offer enterprise support for Red Hat Linux on their mainframe computers. Major software companies such as Oracle Corporation and SAP soon followed, and by 2013 Linux had become established as a major component of both university and business software environments.

Who are the developers who contribute to Linux? One recent study found that there were more software developers from industry than from universities and research organizations. It is also probably true that, over the last decade or so, several hundred professional software engineers from companies like IBM and Intel have participated in major open-source projects. Another survey found that 10 percent of the developers are credited on more than 70 percent of the code. In his book *The Success of Open Source*, Steven Weber concludes:

B.4.10. Linus Torvalds is credited with the development and maintenance of the Linux kernel, which has become the basis for most popular open-source operating systems. In the programmer community he is considered as a "benevolent dictator" who makes sure that the released code is always in perfect shape. Despite the fact that it took him eight years to get his master's degree at the University of Helsinki, he turned out to be a very successful programmer. He described the development of the Linux kernel in a book *Just for Fun*. Most of his concern is not the technical side of Linux but the software patents that are notoriously difficult to deal with.

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

> These numbers count only the major contributors to the Linux kernel. Other active developers report and patch bugs, write small utilities and other applications, and contribute in less elaborate but still important ways to the project. The credit for these kinds of contributions is given in change logs and source code comments, far too many to read and count in a serious way. It is a reasonable guess that there are at least several thousand, and probably in the tens of thousands, of developers who make these smaller contributions to Linux.[22]

How is the work of these volunteer contributors organized? Unlike the formal software engineering frameworks described earlier, with open-source software development there is no authority other than consensus. In the case of Linux, Torvalds still acts as a sort of benevolent dictator supported by a small number of key lieutenants. Other open-source efforts have a small core team who make the decisions about what code to accept. This informal model of software development has produced a complex modern operating system consisting of millions of lines of code with a quality and stability that can rival that of commercial software.

There are now thousands of open-source software projects addressing a large number of different application areas. For many university computer science departments, the use of open-source software for research is the standard way of working. In 2013, SourceForge, a popular site for open-source software projects, stated, "3.4 million developers create powerful software in over 324,000 projects,"[23] which works out as an average of about ten developers per project. In addition, the SourceForge directory "connects more than 46 million consumers with these open source projects and serves more than 4,000,000 downloads a day."[24] Even though only a very small number of these projects attract a critical mass of developers and attain widespread use, the open-source software development model has clearly proved to be a viable alternative to traditional software development methodologies.

Scripting languages

Another type of programming language that is increasing in popularity is a group of languages known as *scripting languages*, high-level programming languages that are interpreted by another program at runtime rather than needing a compiler to transform the source code into an executable program. A *shell script* in Unix was a sequence of commands that could be read from a file and executed in sequence, as if they had been typed in using a keyboard. By extension, the term *script* has become used to describe a set of instructions executed directly by the computer rather than needing a compiler like a traditional programming language. Today, scripting languages have become much more powerful than these early examples because of the addition of standard programming language concepts, such as loops and branches. There are two main uses for scripting languages. The first is as a "glue" language that allows applications to connect off-the-shelf software components that are written in a conventional programming language. The second exploits the functionality and ease of use of scripting languages to employ them as an alternative to conventional languages for a range of general programming tasks.

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

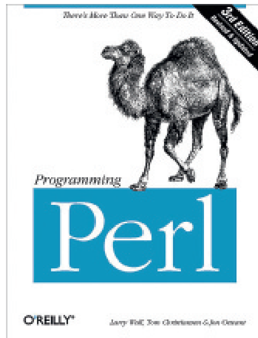


Fig. 4.12. A guide to programming in Perl by its creator, Larry Wall, with Tom Christiansen, and Jon Orwant, widely known as the "Camel" book.

A major characteristic of modern scripting languages is their interactivity, sometimes referred to as a *REPL* programming environment. REPL stands for "Read-Eval-Print Loop" and has its origins in the early work on LISP at MIT. When a user enters an expression, it is immediately evaluated and displayed. In this sense, scripting languages behave as if they were "interpreted," meaning that they operate on an immediate line-by-line execution basis. This is in contrast to traditional "compiled" languages, which generate a binary object file that needs to be explicitly linked to the required set of program libraries – the traditional "edit-compile-link-run" cycle of programming. Because of the increasing power of today's computer chips, the ease of use of scripting languages is often more important than the increase in program efficiency that can be achieved with a compiled language. For example, in scripting languages, to minimize the complexity of programs, declaration of variable types to designate the sort of information each variable can contain is often optional. The variable types are declared implicitly by their usage context and initialized to be something sensible when first used. However, as scripting language programs have become longer and more complex, the benefits of type declarations have been recognized, and most scripting languages now provide an option to make explicit type declarations.

The development of Perl (Practical Extraction and Report Language) in the late 1980s (B.4.11) was one of the defining events in the evolution of scripting languages (Fig. 4.12). David Barron in his book *The World of Scripting Languages* remarks that:

> Perl rapidly developed from being a fairly simple text-processing language to a fully-featured language with extensive capabilities for interacting with the system to manipulate files and processes, establish network connections and other similar system-programming tasks.[25]

From its origins in the Unix world, Perl scripts are now able to run unchanged on all the popular operating system platforms. Other popular scripting languages are VBScript (Visual Basic Scripting Edition) for the Microsoft platform and JavaScript for web applications. The characteristics of ease of use and immediate execution with a REPL environment are sometimes taken as the definition of a scripting language. Under this definition, the Python programming language, which is growing rapidly in popularity, would be regarded as a scripting language.

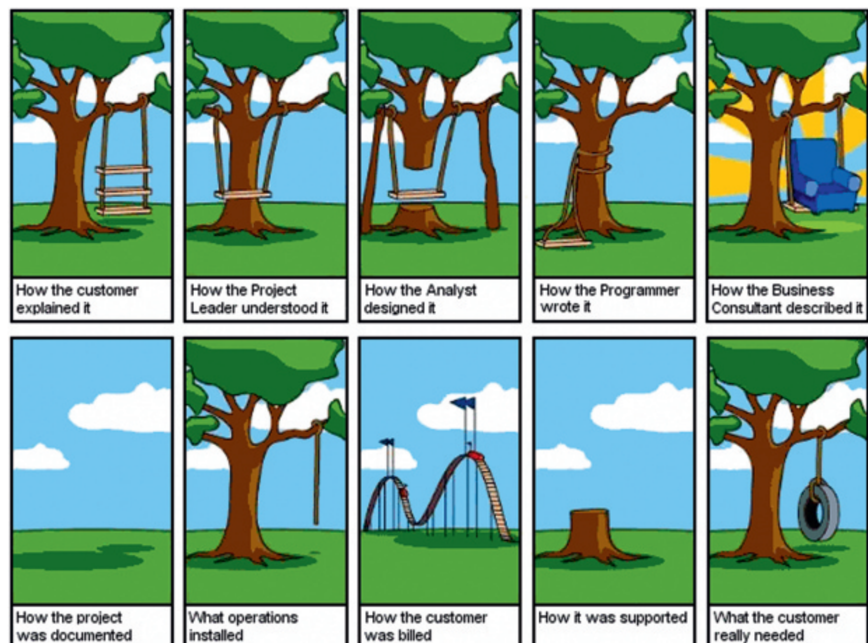
B.4.11. Larry Wall developed Perl in 1987 as a general-purpose Unix scripting language. Since then both the portability and the features in Perl have been expanded greatly and now include support for O-O programming. It is one of the world's most popular programming languages. Wall continues to oversee the evolution of Perl and his role is summarized by the so-called *2 Rules*, taken from the official Perl documentation:

1. Larry is always by definition right about how Perl should behave. This means he has final veto power on the core functionality.
2. Larry is allowed to change his mind about any matter at a later date, regardless of whether he previously invoked Rule 1.

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

Key concepts

- Strong typing
- Control structures – for loops and if-then-else
- Recursion
- Dynamic data structures – linked lists, stacks, and queues
- Data abstraction and information hiding
- Object-oriented programming
  - Classes and objects
  - Inheritance and encapsulation
- Software life cycle
  - Requirements analysis
  - Design
  - Implementation
  - Testing
- Waterfall method and agile methods for software engineering
- Empirical software engineering
- Formal methods
- Open-source software development
- Scripting languages



Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

Some more background on software topics

Unix and C

The origins of time-sharing operating systems can be traced back to MIT, with John McCarthy's early prototype and Fernando Corbató's Compatible Time-Sharing System in 1961. These beginnings led to Licklider's very ambitious project MAC and the Multiplexed Information and Computing Service – Multics – time-sharing operating system. Bell Labs were partners in the project but became frustrated by its size, complexity, and slow progress. When Bell Labs withdrew from the project in 1969, Thompson and Ritchie (B.4.12) and some colleagues from Bell Labs decided to produce their own stripped down version of a time-sharing operating system and to try to create a community around the new code base:

> What we wanted to preserve was not just a good environment in which to do programming, but a system around which a fellowship could form. We knew from experience that the essence of communal computing, as supplied by remote-access, time-shared machines, is not just to type programs into a terminal instead of a key-punch, but to encourage close communication.[26]

They were unable to get funding from Bell Labs management to buy a new computer for their project so they found an old and little-used PDP-7 minicomputer to begin their entirely unfunded *skunkworks* project. In the course of this work they developed a hierarchical file system, the concept of treating devices as files and the notion of processes. They also created a set of utilities giving users the ability to print, copy, delete, and edit files plus a simple command interpreter or *shell*. The Unix operating system then consisted of a set of utilities under the control of a small and efficient operating system *kernel*. The kernel provided services to start and stop programs, handle the file system, and schedule access to resources and devices avoiding conflicts. By promising to create a system specifically designed for editing and formatting text, in 1970 they finally managed to get funding to buy a modern PDP-11 computer. It was also in 1970 that their colleague Brian Kernighan suggested the name Unix, as a play on the name Multics. In 1972 *Unix pipes* were introduced that enabled small utility programs to be combined to create more powerful programs. Using such pipes to create a powerful system utility rather than developing a single monolithic program with the same combined functionality became known as the Unix philosophy – "the idea that the power of a system comes more from the relationships among programs than from the programs themselves."[27] Every program in Unix had originally been written in assembly language but Thompson had developed a definition and compiler for a new language for the PDP-7 that he called B. The language was a stripped down and modified version of the BCPL language developed in Cambridge, U.K., by Martin Richards. With the arrival of the more powerful PDP-11 in 1970, Ritchie took Thompson's B language and developed the C programming language to take advantage of the new machine's byte addressability and other features. By 1973 the new language was powerful enough for much of the Unix kernel to be rewritten in C. In 1977, with further changes to the language, Ritchie and Steven Johnson were able to produce a portable version of the Unix operating system. Johnson's Portable C Compiler then allowed both C and Unix to spread to other platforms. In 1978 Kernighan and Ritchie published the first edition of *The C Programming Language*, which served for many years as the informal specification of the standard (Fig. 4.13).

B.4.12. Ken Thompson and Dennis Ritchie (1941–2011), the developers of C programming language and the Unix operating system.

Thompson and Ritchie's Unix operating system has been enormously influential. Because AT&T was a regulated telephone monopoly it was barred from doing significant commercial developments in

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

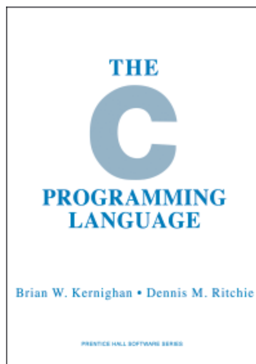


Fig. 4.13. The first edition of Kernighan and Ritchie's C programming language book.

the computing arena. Universities were therefore able to order the data tapes from Bell Labs for a nominal charge of $150 for materials. For that they received the entire source code for the first general-purpose operating system for minicomputers. This along with the portable C compiler was all that university researchers needed to produce their own versions. After July 1974, when a written version of their work appeared in *Communications of the ACM*, orders came flooding in and first hundreds, and then thousands of minicomputer users started porting Unix to their machines. Thompson spent a sabbatical year in Berkeley in 1975 and a graduate student named Bill Joy became an enthusiastic promoter of Unix. By the early 1980s, the Berkeley System Distribution 4.2 Unix was the de facto standard in the university research community. Joy and his team were then commissioned by ARPA, the Advanced Research Projects Agency, to integrate the newly defined networking protocol TCP/IP into Unix. This was a very significant development for the birth of the Internet as we shall see in Chapter 10.

Formal methods

Software engineering involves many disciplines, including mathematics. In the context of software development the field of *formal methods* uses a variety of mathematical techniques to specify and verify software (B.4.13). A formal specification of the system can be used to prove that the program has the desired properties. Automated *theorem proving* systems attempt to prove that the software does what it was intended to do by using its formal specification, a set of logical axioms and a set of inference rules to produce a formal mathematical proof. An alternative approach uses *model checking*, which verifies properties of the system by an exhaustive search of all the possible states that the system could enter during its execution. It is probably fair to say that formal methods have not so far delivered major benefits for assisting the creation of bug-free code in large software systems. However, there are now some examples of such methods being used to solve real software problems. In 2002 Bill Gates said:

> Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability.[28]

B.4.13. Three pioneers of structured programming and formal methods in software development: Tony Hoare, Edsger Dijkstra, and Niklaus Wirth seen here at an Alpine resort.

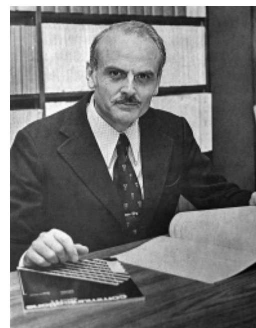
Gates was referring to the SLAM verification engine that checks that software correctly satisfies the behavioral properties of the interfaces that it uses. The SLAM tool is now applied regularly to all Microsoft device drivers and has helped find more than three hundred bugs in the sample drivers that were supplied to developers.

Databases

The main purpose of databases is to store and manage large volumes of data. Database software plays a vital role in our modern society. No bank transactions, online shopping, airline reservations, or even a checking out at the local supermarket would be possible without databases. Database software is now a multibillion dollar business.

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

The relational data model that forms the basis of modern databases is undoubtedly one of the great abstractions of the twentieth century. Historically, there were approaches for handling large volumes of data based on hierarchical, treelike structures or more general network structures. An early example of the hierarchical approach was the IBM's Information Management System. One difficulty with this approach is that not all data relationships fit well into a tree structure. A more general network structure can provide a more flexible solution, but now the user has to know the exact path leading to the data item in order to access or update it. This approach also did not scale well – with the growth of data, programmers found it difficult to navigate through a complicated web of data relationships.



B.4.14. Edgar "Ted" Codd (1923–2003) graduated from Oxford with a degree in mathematics and chemistry and was an RAF fighter pilot during the war. After the war he joined IBM and moved to the United States. In 1981 he received the Turing Award for his contribution to the development of relational databases.

The real breakthrough for database software came with the idea of the relational data model, suggested by a British mathematician Edgar "Ted" Codd (B.4.14). In 1970 he published his groundbreaking paper "A Relational Model of Data for Large Shared Data Banks." The ideas described in this paper became the foundation of modern databases. Ironically, his own company, IBM, was initially not very supportive of his ideas. There were many skeptics and a strong resistance toward relational databases even in professional circles. In the dedication of his book *The Relational Model for Database Management* he refers to this struggle:

> To fellow pilots and aircrew in the Royal Air Force during World War II and the dons at Oxford. These people were the source of my determination to fight for what I believed was right during the ten or more years in which government, industry and commerce were strongly opposed to the relational approach to database management.[29]

The idea of a relational database is simple, yet very powerful. All the data, including the relations between data, is stored in tables that are linked together. The link is established when the same column of data is shared between two or more tables. This column is called a key. The main advantage of the relational data model is that it provides a systematic way to create the interconnections between tables. It is much easier to access data and there is no need to know the path leading to the data (B.4.15). This model is also supported by a powerful mathematical set theory and a declarative programming language called SQL – *structured query language*.

B.4.15. Jim Gray (1944–2007) on board his boat *Tenacious*. In 1988 Gray received the Turing Award for his contributions to database design and transaction processing. After gaining a PhD in computer science from Berkeley, he worked for IBM, Tandem Computers, and DEC. From 1995 Gray was a Technical Fellow at Microsoft Research. He was first to develop a website that displayed geographic data – the Terraserver – and that could deliver data to users using a web service. Gray spent the last decade of his life working with scientists with "Big Data" problems. With astronomer Alex Szalay, he pioneered the hosting of the Sloan Digital Survey astronomical data by creating the SkyServer website. Gray also coined the term *Fourth Paradigm* to reflect the increase in importance of data-intensive science. He was lost at sea, west of San Francisco Bay, in January 2007. Despite a massive collaborative effort by the emergency services and the computer science community, in searching for signs of *Tenacious*, no trace was ever found.

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

Design patterns

B.4.16. The "Gang of Four": Ralph Johnson, Erich Gamma, Richard Helm, and John Vlissides.

The advances of structured and O-O programming still underpin the way in which systems are written today. However, as the software industry grew, the task of teaching each new wave of programmers how to program efficiently has led to a new level of abstraction. In 1995, four software engineers – Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides – got together and identified what they called "design patterns" (B.4.16). These are standard patterns in software that everybody uses to perform a number of simple tasks.

One example is the "Observer pattern." This ensures that when one object changes its state, all of its dependents are notified and updated automatically. They identified twenty-two patterns they called by easy-to-remember names. Apart from *Observer*, other example patterns are *Factory*, *Decorator*, *Interpreter*, and *Visitor*. Their book on design patterns has become one of the best-selling and most-cited books in computer science. It established a fixed vocabulary for talking about O-O software at a level above program code, so that programs – and programmers – became more transferable, understandable, and accurate.

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

Three expensive software errors

NASA's Mariner 1 Space Probe (1962)

A bug in the flight software for the Mariner 1 (Fig. 4.14) mission caused the rocket to divert from its intended path on launch. Mission control destroyed the rocket over the Atlantic Ocean 293 seconds after launch. NASA's website says the problem was caused by a combination of two factors. Improper operation of the Atlas airborne beacon equipment resulted in a loss of the rate signal from the vehicle. The airborne beacon used for obtaining rate data was inoperative for four periods ranging from 1.5 to 61 seconds in duration. Additionally, the Mariner 1 Post Flight Review Board determined that the omission of a hyphen in the data-editing program allowed transmission of incorrect guidance signals to the spacecraft. During the periods the airborne beacon was inoperative, the missing hyphen in the data-editing program caused the computer to incorrectly accept the sweep frequency of the ground receiver as it sought the vehicle beacon signal and combined this data with the tracking data sent to the guidance computation. This caused the computer to automatically generate a series of unnecessary course corrections using the erroneous steering commands and these finally threw the spacecraft off course. The science fiction author Arthur C. Clarke wrote several years later that Mariner 1 was "wrecked by the most expensive hyphen in history."[30]

Fig. 4.14. Mariner 1 Space probe to Venus was the first interplanetary mission aiming to put a satellite around Venus. There are various stories about the reason why this mission had to be aborted. Most of them firmly point at a bug in the FORTRAN code of the guidance system that unexpectedly changed the trajectory of the rocket. A hyphen (overbar) missed in a mathematical expression led to the $80 million failure. Five months later the Mariner 2 was successfully launched and completed the mission.

Ariane 5 Flight 501 Launch (1996)

In his Turing Award lecture, Tony Hoare warned of the dangers of the complexities of the ADA programming language:

> And so, the best of my advice to the originators and designers of ADA has been ignored. In this last resort, I appeal to you, representatives of the programming profession in the United States, and citizens concerned with the welfare and safety of your own country and of mankind: Do not allow this language in its present state to be used in applications where reliability is critical, i.e., nuclear power stations, cruise missiles, early warning systems, anti-ballistic missile defense systems. The next rocket to go astray as a result of a programming language error may not be an exploratory space rocket on a harmless trip to Venus: It may be a nuclear warhead exploding over one of our own cities. An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations. Be vigilant to reduce that risk, not to increase it.[31]

Some of the ADA code for the Ariane 4 rocket was reused in the Ariane 5's control software. The error was in the code that converts a 64-bit floating-point number to a 16-bit signed integer. The faster engines caused the 64-bit numbers to be larger in the Ariane 5 than in the Ariane 4. This triggered an overflow condition that resulted in the flight computer crashing. The backup computer then also crashed, followed 0.05 seconds later by a crash of the primary computer. As a result of these software crashes, the mission was terminated thirty-seven seconds after launch (Fig. 4.15).

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007

Fig. 4.15. Photo of the destruction of the first launch of the Ariane 5 Flight 501 rocket. Just thirty-seven seconds into the launch, the trajectory suddenly tilted by almost 90 degrees and the rocket self-destructed. The software error occurred during data conversion from a 64-bit floating-point number to a 16-bit signed integer. This led to a sequence of events that resulted in a complete loss of the guidance system.

NASA's Mars Climate Orbiter (1999)

The root cause for the loss of the Mars Climate Orbiter (Fig. 4.16) spacecraft was the failure to use metric units in the coding of the software file, "Small Forces," used in trajectory models. Instead of reporting the thruster data in metric units of Newtonseconds (N-s), the data was reported in English units of pound-seconds (lbf-s). Subsequent processing of this thruster data by the navigation software algorithm underestimated the effect on the spacecraft trajectory by a factor of 4.45, which is the required conversion factor from force in pounds to Newtons. An erroneous trajectory was then computed using this incorrect data.

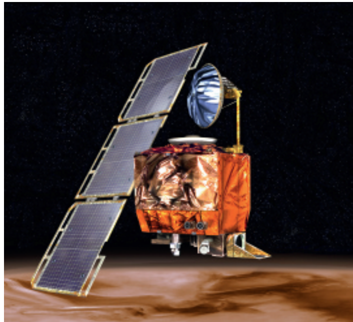


Fig. 4.16. Artist's impression of the Mars Climate Orbiter. The space probe was lost at the first attempt to enter the orbit around Mars on 3 September 1999. Putting a probe into a final planetary orbit is a long process during which the initial orbit is gradually reduced until the probe reaches its permanent orbit. Because of a software error the orbiter entered the Martian atmosphere at too high a velocity and consequently burnt up. The cost of this failure was $125 million.

Downloaded from http:/www.cambridge.org/core. National Library of the Philippines, on 03 Nov 2016 at 06:13:42, subject to the Cambridge Core terms of use, available at http:/www.cambridge.org/core/terms. http://dx.doi.org/10.1017/CBO9781139032643.007