Computer programming

Computer programming is the process of designing and building an executable computer program to accomplish a specific computing result or to perform a specific task. Programming involves tasks such as: analysis, generating algorithms, profiling algorithms' accuracy and resource consumption, and the implementation of algorithms in a chosen programming language (commonly referred to as **coding**).[1][2] The source code of a program is written in one or more languages that are intelligible to programmers, rather than machine code, which is directly executed by the central processing unit. The purpose of programming is to find a sequence of instructions that will automate the performance of a task (which can be as complex as an operating system) on a computer, often for solving a given problem. Proficient programming thus often requires expertise in several different subjects, including knowledge of the application domain, specialized algorithms, and formal logic.

Tasks accompanying and related to programming include: testing, debugging, source code maintenance, implementation of build systems, and management of derived artifacts, such as the machine code of computer programs. These might be considered part of the programming process, but often the term *software development* is used for this larger process with the term *programming*, *implementation*, or *coding* reserved for the actual writing of code. *Software engineering* combines engineering techniques with software development practices. *Reverse engineering* is a related process used by designers, analysts and programmers to understand and re-create/re-implement.[3]:3

Contents

History

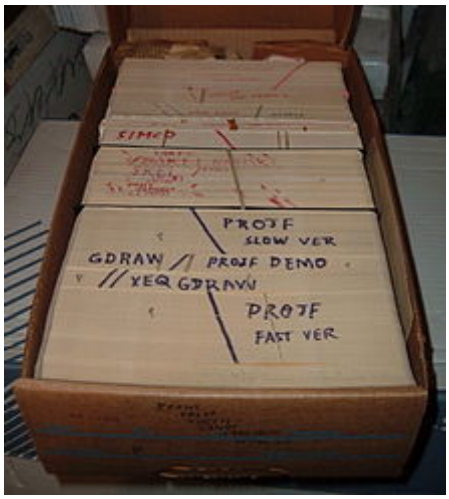
Programmable devices have existed for centuries. As early as the 9th century, a programmable music sequencer was invented by the Persian Banu Musa brothers, who described an automated mechanical flute player in the *Book of Ingenious Devices*.[4][5] In 1206, the Arab engineer Al-Jazari invented a programmable drum machine where a musical mechanical automaton could be made to play different rhythms and drum patterns, via pegs and cams.[6][7] In 1801, the Jacquard loom could produce entirely different weaves by changing the "program" – a series of pasteboard cards with holes punched in them.

Ada Lovelace, whose notes added to the end of Luigi Menabrea's paper included the first algorithm designed for processing by an Analytical Engine. She is often recognized as history's first computer programmer.

Code-breaking algorithms have also existed for centuries. In the 9th century, the Arab mathematician Al-Kindi described a cryptographic algorithm for deciphering encrypted code, in *A Manuscript on Deciphering Cryptographic Messages*. He gave the first description of cryptanalysis by frequency analysis, the earliest code-breaking algorithm.[8]

The first computer program is generally dated to 1843, when mathematician Ada Lovelace published an algorithm to calculate a sequence of Bernoulli numbers, intended to be carried out by Charles Babbage's Analytical Engine.[9]

In the 1880s Herman Hollerith invented the concept of storing *data* in machine-readable form.[10] Later a control panel (plugboard) added to his 1906 Type I Tabulator allowed it to be programmed for different jobs, and by the late 1940s, unit record equipment such as the IBM 602 and IBM 604, were programmed by control panels in a similar way, as were the first electronic computers. However, with the concept of the stored-program computer introduced in 1949, both programs and data were stored and manipulated in the same way in computer memory.



Data and instructions were once stored on external punched cards, which were kept in order and arranged in program decks.

Machine language

Machine code was the language of early programs, written in the instruction set of the particular machine, often in binary notation. Assembly languages were soon developed that let the programmer specify instruction in a text format, (e.g., ADD X, TOTAL), with abbreviations for each operation code and meaningful names for specifying addresses. However, because an assembly language is little more than a different notation for a machine language, any two machines with different instruction sets also have different assembly languages.

Compiler languages

High-level languages made the process of developing a program simpler and more understandable, and less bound to the underlying hardware. FORTRAN, the first widely used high-level language to have a functional implementation, came out in 1957[11] and many other languages were soon developed – in particular, COBOL aimed at commercial data processing, and Lisp for computer research.

Wired control panel for an IBM 402 Accounting Machine.

These compiled languages allow the programmer to write programs in terms that are syntactically richer, and more capable of abstracting the code, making it targetable to varying machine instruction sets via compilation declarations and heuristics. The first compiler for a programming language was developed by Grace Hopper.[12] When Hopper went to work on UNIVAC in 1949, she brought the idea of using compilers with her.[13][14] Compilers harness the power of computers to make programming easier[11] by allowing programmers to specify calculations by entering a formula using infix notation (e.g., Y = X*2 + 5*X + 9) for example. FORTRAN, the first widely used high-level language to have a functional implementation which permitted the abstraction of reusable blocks of code, came out in 1957[11] and many other languages were soon developed - in particular, COBOL aimed at commercial data processing, and Lisp for computer research. In 1951 Frances E. Holberton developed the first sort-merge generator, which ran on the UNIVAC I.[15] Another woman working at UNIVAC, Adele Mildred Koss, developed a program that was a precursor to report generators.[15] The idea for the creation of COBOL started in 1959 when Mary K. Hawes, who worked for the Burroughs Corporation, set up a meeting to discuss creating a common business language.[16] She invited six people, including Grace Hopper.[16] Hopper was involved in developing COBOL as a business language and creating "self-documenting" programming.[17][18] Hopper's contribution to COBOL was based on her programming language, called FLOW-MATIC.[14] In 1961, Jean E. Sammet developed FORMAC and also published *Programming Languages: History and Fundamentals*, which went on to be a standard work on programming languages.[16][19]

Source code entry

Programs were mostly still entered using punched cards or paper tape. See Computer programming in the punch card era. By the late 1960s, data storage devices and computer terminals became inexpensive enough that programs could be created by typing directly into the computers. Frances Holberton created a code to allow keyboard inputs while she worked at UNIVAC.[20]

Text editors were developed that allowed changes and corrections to be made much more easily than with punched cards. Sister Mary Kenneth Keller worked on developing the programming language BASIC while she was a graduate student at Dartmouth in the 1960s.[21] One of the first object-oriented programming languages, Smalltalk, was developed by seven programmers, including Adele Goldberg, in the 1970s.[22]

Modern programming

Quality requirements

Whatever the approach to development may be, the final program must satisfy some fundamental properties. The following properties are among the most important:[23] [24]

- Reliability: how often the results of a program are correct. This depends on conceptual correctness of algorithms, and minimization of programming mistakes, such as mistakes in

resource management (e.g., buffer overflows and race conditions) and logic errors (such as division by zero or off-by-one errors).
- Robustness: how well a program anticipates problems due to errors (not bugs). This includes situations such as incorrect, inappropriate or corrupt data, unavailability of needed resources such as memory, operating system services, and network connections, user error, and unexpected power outages.
- Usability: the ergonomics of a program: the ease with which a person can use the program for its intended purpose or in some cases even unanticipated purposes. Such issues can make or break its success even regardless of other issues. This involves a wide range of textual, graphical, and sometimes hardware elements that improve the clarity, intuitiveness, cohesiveness and completeness of a program's user interface.
- Portability: the range of computer hardware and operating system platforms on which the source code of a program can be compiled/interpreted and run. This depends on differences in the programming facilities provided by the different platforms, including hardware and operating system resources, expected behavior of the hardware and operating system, and availability of platform-specific compilers (and sometimes libraries) for the language of the source code.
- Maintainability: the ease with which a program can be modified by its present or future developers in order to make improvements or customizations, fix bugs and security holes, or adapt it to new environments. Good practices[25] during initial development make the difference in this regard. This quality may not be directly apparent to the end user but it can significantly affect the fate of a program over the long term.
- Efficiency/performance: Measure of system resources a program consumes (processor time, memory space, slow devices such as disks, network bandwidth and to some extent even user interaction): the less, the better. This also includes careful management of resources, for example cleaning up temporary files and eliminating memory leaks. This is often discussed under the shadow of a chosen programming language. Although the language certainly affects performance, even slower languages, such as Python, can execute programs instantly from a human perspective. Speed, resource usage, and performance are important for programs that bottleneck the system, but efficient use of programmer time is also important and is related to cost: more hardware may be cheaper.

Readability of source code

In computer programming, readability refers to the ease with which a human reader can comprehend the purpose, control flow, and operation of source code. It affects the aspects of quality above, including portability, usability and most importantly maintainability.

Readability is important because programmers spend the majority of their time reading, trying to understand and modifying existing source code, rather than writing new source code. Unreadable code often leads to bugs, inefficiencies, and duplicated code. A study[26] found that a few simple readability transformations made code shorter and drastically reduced the time to understand it.

Following a consistent programming style often helps readability. However, readability is more than just programming style. Many factors, having little or nothing to do with the ability of the computer to efficiently compile and execute the code, contribute to readability.[27] Some of these factors include:

- Different indent styles (whitespace)
- Comments
- Decomposition
- Naming conventions for objects (such as variables, classes, procedures, etc.)

The presentation aspects of this (such as indents, line breaks, color highlighting, and so on) are often handled by the source code editor, but the content aspects reflect the programmer's talent and skills.

Various visual programming languages have also been developed with the intent to resolve readability concerns by adopting non-traditional approaches to code structure and display. Integrated development environments (IDEs) aim to integrate all such help. Techniques like Code refactoring can enhance readability.

Algorithmic complexity

The academic field and the engineering practice of computer programming are both largely concerned with discovering and implementing the most efficient algorithms for a given class of problem. For this purpose, algorithms are classified into *orders* using so-called Big O notation, which expresses resource use, such as execution time or memory consumption, in terms of the size of an input. Expert programmers are familiar with a variety of well-established algorithms and their respective complexities and use this knowledge to choose algorithms that are best suited to the circumstances.

Chess algorithms as an example

"Programming a Computer for Playing Chess" was a 1950 paper that evaluated a "minimax" algorithm that is part of the history of algorithmic complexity; a course on IBM's Deep Blue (chess computer) is part of the computer science curriculum at Stanford University.[28]

Methodologies

The first step in most formal software development processes is requirements analysis, followed by testing to determine value modeling, implementation, and failure elimination (debugging). There exist a lot of differing approaches for each of those tasks. One approach popular for requirements analysis is Use Case analysis. Many programmers use forms of Agile software development where the various stages of formal software development are more integrated together into short cycles that take a few weeks rather than years. There are many approaches to the Software development process.

Popular modeling techniques include Object-Oriented Analysis and Design (OOAD) and Model-Driven Architecture (MDA). The Unified Modeling Language (UML) is a notation used for both the OOAD and MDA.

A similar technique used for database design is Entity-Relationship Modeling (ER Modeling).

Implementation techniques include imperative languages (object-oriented or procedural), functional languages, and logic languages.

Measuring language usage

It is very difficult to determine what are the most popular modern programming languages. Methods of measuring programming language popularity include: counting the number of job advertisements that mention the language,[29] the number of books sold and courses teaching the language (this overestimates the importance of newer languages), and estimates of the number of existing lines of code written in the language (this underestimates the number of users of business languages such as COBOL).

Some languages are very popular for particular kinds of applications, while some languages are regularly used to write many different kinds of applications. For example, COBOL is still strong in corporate data centers[30] often on large mainframe computers, Fortran in engineering applications, scripting languages in Web development, and C in embedded software. Many applications use a mix of several languages in their construction and use. New languages are generally designed around the syntax of a prior language with new functionality added, (for example C++ adds object-orientation to C, and Java adds memory management and bytecode to C++, but as a result, loses efficiency and the ability for low-level manipulation).

Debugging

Debugging is a very important task in the software development process since having defects in a program can have significant consequences for its users. Some languages are more prone to some kinds of faults because their specification does not require compilers to perform as much checking as other languages. Use of a static code analysis tool can help detect some possible problems. Normally the first step in debugging is to attempt to reproduce the problem. This can be a non-trivial task, for example as with parallel processes or some unusual software bugs. Also, specific user environment and usage history can make it difficult to reproduce the problem.

The first known actual bug causing a problem in a computer was a moth, trapped inside a Harvard mainframe, recorded in a log book entry dated September 9, 1947.[31] "Bug" was already a common term for a software defect when this bug was found.

After the bug is reproduced, the input of the program may need to be simplified to make it easier to debug. For example, when a bug in a compiler can make it crash when parsing some large source file, a simplification of the test case that results in only few lines from the original source file can be sufficient to reproduce the same crash. Trial-and-error/divide-and-conquer is needed: the programmer will try to remove some parts of the original test case and check if the problem still exists. When debugging the problem in a GUI, the programmer can try to skip some user interaction from the original problem description and check if remaining actions are sufficient for bugs to appear. Scripting and breakpointing is also part of this process.

Debugging is often done with IDEs like Eclipse, Visual Studio, Xcode, Kdevelop, NetBeans and Code::Blocks. Standalone debuggers like GDB are also used, and these often provide less of a visual environment, usually using a command line. Some text editors such as Emacs allow GDB to be invoked through them, to provide a visual environment.

Programming languages

Different programming languages support different styles of programming (called *programming paradigms*). The choice of language used is subject to many considerations, such as company policy, suitability to task, availability of third-party packages, or individual preference. Ideally, the programming language best suited for the task at hand will be selected. Trade-offs from this ideal involve finding enough programmers who know the language to build a team, the availability of compilers for that language, and the efficiency with which programs written in a given language execute. Languages form an approximate spectrum from "low-level" to "high-level"; "low-level" languages are typically more machine-oriented and faster to execute, whereas "high-level" languages are more abstract and easier to use but execute less quickly. It is usually easier to code in "high-level" languages than in "low-level" ones.

Allen Downey, in his book *How To Think Like A Computer Scientist*, writes:

The details look different in different languages, but a few basic instructions appear in just about every language:

- Input: Gather data from the keyboard, a file, or some other device.
- Output: Display data on the screen or send data to a file or other device.
- Arithmetic: Perform basic arithmetical operations like addition and multiplication.
- Conditional Execution: Check for certain conditions and execute the appropriate sequence of statements.
- Repetition: Perform some action repeatedly, usually with some variation.

Many computer languages provide a mechanism to call functions provided by shared libraries. Provided the functions in a library follow the appropriate run-time conventions (e.g., method of passing arguments), then these functions may be written in any other language.

Programmers

Computer programmers are those who write computer software. Their jobs usually involve:

- Coding
- Debugging
- Documentation
- Integration
- Maintenance
- Requirements analysis
- Software architecture
- Software testing
- Specification

See also

- ACCU
- Association for Computing Machinery
- Computer networking
- Hello world program
- Institution of Analysts and Programmers
- National Coding Week
- System programming
- Computer programming in the punched card era
- *The Art of Computer Programming*
- Women in computing
- Timeline of women in computing

References

1. Bebbington, Shaun (2014). "What is coding" (https://yearofcodes.tumblr.com/what-is-coding). *Tumblr*. Archived (https://web.archive.org/web/20200429195646/https://yearofcodes.tumblr.com/what-is-coding) from the original on April 29, 2020. Retrieved March 3, 2014.
2. Bebbington, Shaun (2014). "What is programming" (https://yearofcodes.tumblr.com/what-is-programming). *Tumblr*. Archived (https://web.archive.org/web/20200429195958/https://yearofcodes.tumblr.com/what-is-programming) from the original on April 29, 2020. Retrieved March 3, 2014.
3. Eilam, Eldad (2005). *Reversing: secrets of reverseengineering*. John Wiley & Sons. ISBN 978-0-7645-7481-8.

4. Koetsier, Teun (2001), "On the prehistory of programmable machines: musical automata, looms, calculators", *Mechanism and Machine Theory*, Elsevier, **36** (5): 589–603, doi:10.1016/S0094-114X(01)00005-2 (https://doi.org/10.1016%2FS0094-114X%2801%2900005-2).
5. Kapur, Ajay; Carnegie, Dale; Murphy, Jim; Long, Jason (2017). "Loudspeakers Optional: A history of non-loudspeaker-based electroacoustic music" (https://doi.org/10.1017%2FS1355771817000103). *Organised Sound*. Cambridge University Press. **22** (2): 195–205. doi:10.1017/S1355771817000103 (https://doi.org/10.1017%2FS1355771817000103). ISSN 1355-7718 (https://www.worldcat.org/issn/1355-7718).
6. Fowler, Charles B. (October 1967). "The Museum of Music: A History of Mechanical Instruments". *Music Educators Journal*. **54** (2): 45–49. doi:10.2307/3391092 (https://doi.org/10.2307%2F3391092). JSTOR 3391092 (https://www.jstor.org/stable/3391092). S2CID 190524140 (https://api.semanticscholar.org/CorpusID:190524140).
7. Noel Sharkey (2007), A 13th Century Programmable Robot (https://web.archive.org/web/20070629182810/http://www.shef.ac.uk/marcoms/eview/articles58/robot.html), University of Sheffield
8. Dooley, John F. (2013). *A Brief History of Cryptology and Cryptographic Algorithms*. Springer Science & Business Media. pp. 12–3. ISBN 9783319016283.
9. Fuegi, J.; Francis, J. (2003). "Lovelace & Babbage and the Creation of the 1843 'notes' ". *IEEE Annals of the History of Computing*. **25** (4): 16. doi:10.1109/MAHC.2003.1253887 (https://doi.org/10.1109%2FMAHC.2003.1253887).
10. da Cruz, Frank (March 10, 2020). "Columbia University Computing History – Herman Hollerith" (http://www.columbia.edu/acis/history/hollerith.html). *Columbia University*. Columbia.edu. Archived (https://web.archive.org/web/20200429210742/http://www.columbia.edu/cu/computinghistory/hollerith.html) from the original on April 29, 2020. Retrieved April 25, 2010.
11. Bergstein, Brian (March 20, 2007). "Fortran creator John Backus dies" (http://www.nbcnews.com/id/17704662). *NBC News*. Archived (https://web.archive.org/web/20200429211030/http://www.nbcnews.com/id/17704662) from the original on April 29, 2020. Retrieved April 25, 2010.
12. Smith 2013, p. 6.
13. Ceruzzi 1998, p. 84-85.
14. Gürer 1995, p. 176.
15. Gürer 1995, p. 177.
16. Gürer 1995, p. 179.
17. Smith 2013, p. 7.
18. Ceruzzi 1998, p. 92.
19. "Computer Authority to Speak Here" (https://www.newspapers.com/clip/24510360/). *The Times*. April 9, 1972. Retrieved October 13, 2018 – via Newspapers.com.
20. "Frances Holberton, Pioneer in Computer Languages, Dies" (https://www.newspapers.com/clip/24508073). *The Courier-Journal*. December 12, 2001. Retrieved October 13, 2018 – via Newspapers.com.
21. Gürer 1995, p. 180-181.
22. "Adele Goldberg" (https://www.cs.umd.edu/hcil/muiseum/goldberg/goldberg_page.htm). University of Maryland, College Park. Retrieved October 14, 2018.
23. "NIST To Develop Cloud Roadmap" (https://www.informationweek.com/cloud/nist-to-develop-cloud-roadmap/d/d-id/1093958?). *InformationWeek*. November 5, 2010. "Computing initiative seeks to remove barriers to cloud adoption in security, interoperability, portability and reliability."
24. "What is it based on". *ComputerWorld*. April 9, 1984. p. 13. "Is it based on ... Reliability Portability. Compatibility"
25. "Programming 101: Tips to become a good programmer - Wisdom Geek" (https://wisdomgeek.com/programming/tips-become-good-programmer). *Wisdom Geek*. May 19, 2016. Retrieved May 23, 2016.

26. Elshoff, James L.; Marcotty, Michael (1982). "Improving computer program readability to aid modification". *Communications of the ACM*. **25** (8): 512–521. doi:10.1145/358589.358596 (https://doi.org/10.1145%2F358589.358596). S2CID 30026641 (https://api.semanticscholar.org/CorpusID:30026641).
27. Multiple (wiki). "Readability" (http://docforge.com/wiki/Readability). *Docforge*. Archived (https://web.archive.org/web/20200429211203/http://www.docforge.com/wiki/Readability) from the original on April 29, 2020. Retrieved January 30, 2010.
28. Piech, Chris. "Deep Blue" (https://stanford.edu/~cpiech/cs221/apps/deepBlue.html). "In 1950, Claude Shannon published ... "Programming a Computer for Playing Chess", ... "minimax" algorithm"
29. Enticknap, Nicholas (September 11, 2007). "SSL/Computer Weekly IT salary survey: finance boom drives IT job growth" (http://www.computerweekly.com/Articles/2007/09/11/226631/sslcomputer-weekly-it-salary-survey-finance-boom-drives-it-job.htm).
30. Mitchell, Robert (May 21, 2012). "The Cobol Brain Drain" (http://www.computerworld.com/article/2504568/data-center/the-cobol-brain-drain.html). Computer World. Retrieved May 9, 2015.
31. Photograph courtesy Naval Surface Warfare Center, Dahlgren, Virginia, from National Geographic Sept. 1947 (https://www.nationalgeographic.org/thisday/sep9/worlds-first-computer-bug)

Sources

- Ceruzzi, Paul E. (1998). *History of Computing* (https://archive.org/details/historyofmodernc00ceru). Cambridge, Massachusetts: MIT Press. ISBN 9780262032551 – via EBSCOhost.
- Evans, Claire L. (2018). *Broad Band: The Untold Story of the Women Who Made the Internet* (https://books.google.com/books?id=C8ouDwAAQBAJ&q=9780735211759&pg=PP1). New York: Portfolio/Penguin. ISBN 9780735211759.
- Gürer, Denise (1995). "Pioneering Women in Computer Science" (https://courses.cs.washington.edu/courses/csep590/06au/readings/p175-gurer.pdf) (PDF). *Communications of the ACM*. **38** (1): 45–54. doi:10.1145/204865.204875 (https://doi.org/10.1145%2F204865.204875). S2CID 6626310 (https://api.semanticscholar.org/CorpusID:6626310).
- Smith, Erika E. (2013). "Recognizing a Collective Inheritance through the History of Women in Computing" (http://search.ebscohost.com/login.aspx?direct=true&db=lfh&AN=90670848&site=ehost-live). *CLCWeb: Comparative Literature & Culture: A WWWeb Journal*. **15** (1): 1–9 – via EBSCOhost.

Further reading

- A.K. Hartmann, *Practical Guide to Computer Simulations (https://web.archive.org/web/20090211113048/http://worldscibooks.com/physics/6988.html)*, Singapore: World Scientific (2009)
- A. Hunt, D. Thomas, and W. Cunningham, *The Pragmatic Programmer. From Journeyman to Master*, Amsterdam: Addison-Wesley Longman (1999)
- Brian W. Kernighan, *The Practice of Programming*, Pearson (1999)
- Weinberg, Gerald M., *The Psychology of Computer Programming*, New York: Van Nostrand Reinhold (1971)
- Edsger W. Dijkstra, *A Discipline of Programming*, Prentice-Hall (1976)
- O.-J. Dahl, E.W.Dijkstra, C.A.R. Hoare, *Structured Programming*, Academic Press (1972)
- David Gries, *The Science of Programming*, Springer-Verlag (1981)

External links

- Media related to Computer programming at Wikimedia Commons
- Quotations related to Programming at Wikiquote
- Software engineering (https://curlie.org/Computers/Software/Software_Engineering/) at Curlie

---