Important Concepts and Tools in Computer Programming

Duane Gann

Darrion Nowak

First Edition, 2012

ISBN 978-81-323-1278-9

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1 - Introduction to Computer Programming

Chapter 2 - Programming Language

Chapter 3 - Programmer

Chapter 4 - Algorithmic Skeleton and Control Table

Chapter 5 - Nassi–Shneiderman Diagram

Chapter 6 - Profiling (Computer Programming) and Integrated Development Environment

Chapter 7 - Code Coverage and Compiler

Chapter 8 - Programming Style and Text Editor

Chapter 9 - Data Modeling

Chapter 10 - ActiveReports

Chapter 11 - FarPoint Spread and Hildon

Chapter 12 - JBoss Tools and XLeratorDB

Chapter 1

Introduction to Computer Programming

Computer programming (often shortened to **programming** or **coding**) is the process of designing, writing, testing, debugging / troubleshooting, and maintaining the source code of computer programs. This source code is written in a programming language. The purpose of programming is to create a program that exhibits a certain desired behaviour. The process of writing source code often requires expertise in many different subjects, including knowledge of the application domain, specialized algorithms and formal logic.

Definition

Hoc and Nguyen-Xuan define computer programming as "the process of transforming a mental plan in familiar terms into one compatible with the computer." Said another way, programming is the craft of transforming requirements into something that a computer can execute.

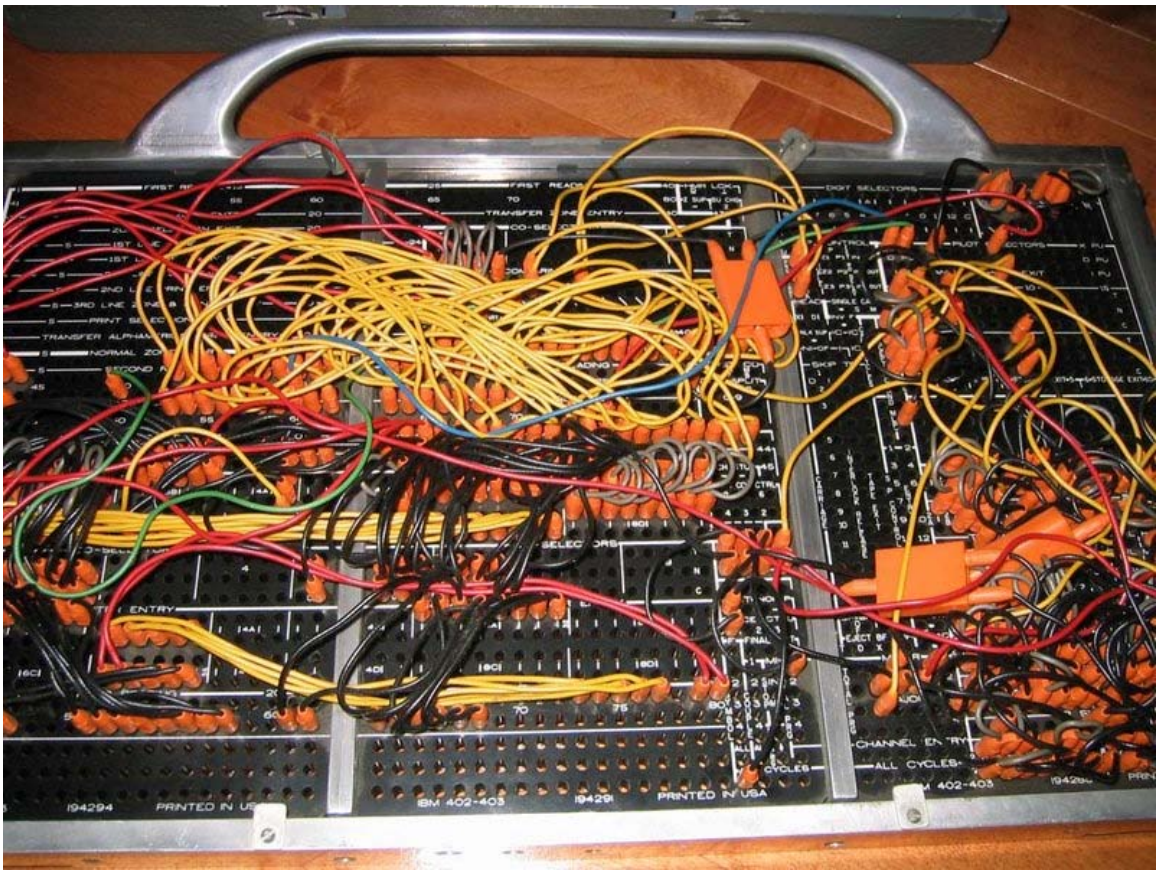
Overview

Within software engineering, programming (the *implementation*) is regarded as one phase in a software development process.

There is an ongoing debate on the extent to which the writing of programs is an art, a craft or an engineering discipline. In general, good programming is considered to be the measured application of all three, with the goal of producing an efficient and evolvable software solution (the criteria for "efficient" and "evolvable" vary considerably). The discipline differs from many other technical professions in that programmers, in general, do not need to be licensed or pass any standardized (or governmentally regulated) certification tests in order to call themselves "programmers" or even "software engineers." However, representing oneself as a "Professional Software Engineer" without a license from an accredited institution is illegal in many parts of the world. However, because the discipline covers many areas, which may or may not include critical applications, it is debatable whether licensing is required for the profession as a whole. In most cases, the discipline is self-governed by the entities which require the programming, and sometimes very strict environments are defined (e.g. United States Air Force use of AdaCore and security clearance).

Another ongoing debate is the extent to which the programming language used in writing computer programs affects the form that the final program takes. This debate is analogous to that surrounding the Sapir–Whorf hypothesis in linguistics, which postulates that a particular spoken language's nature influences the habitual thought of its speakers. Different language patterns yield different patterns of thought. This idea challenges the possibility of representing the world perfectly with language, because it acknowledges that the mechanisms of any language condition the thoughts of its speaker community.

History

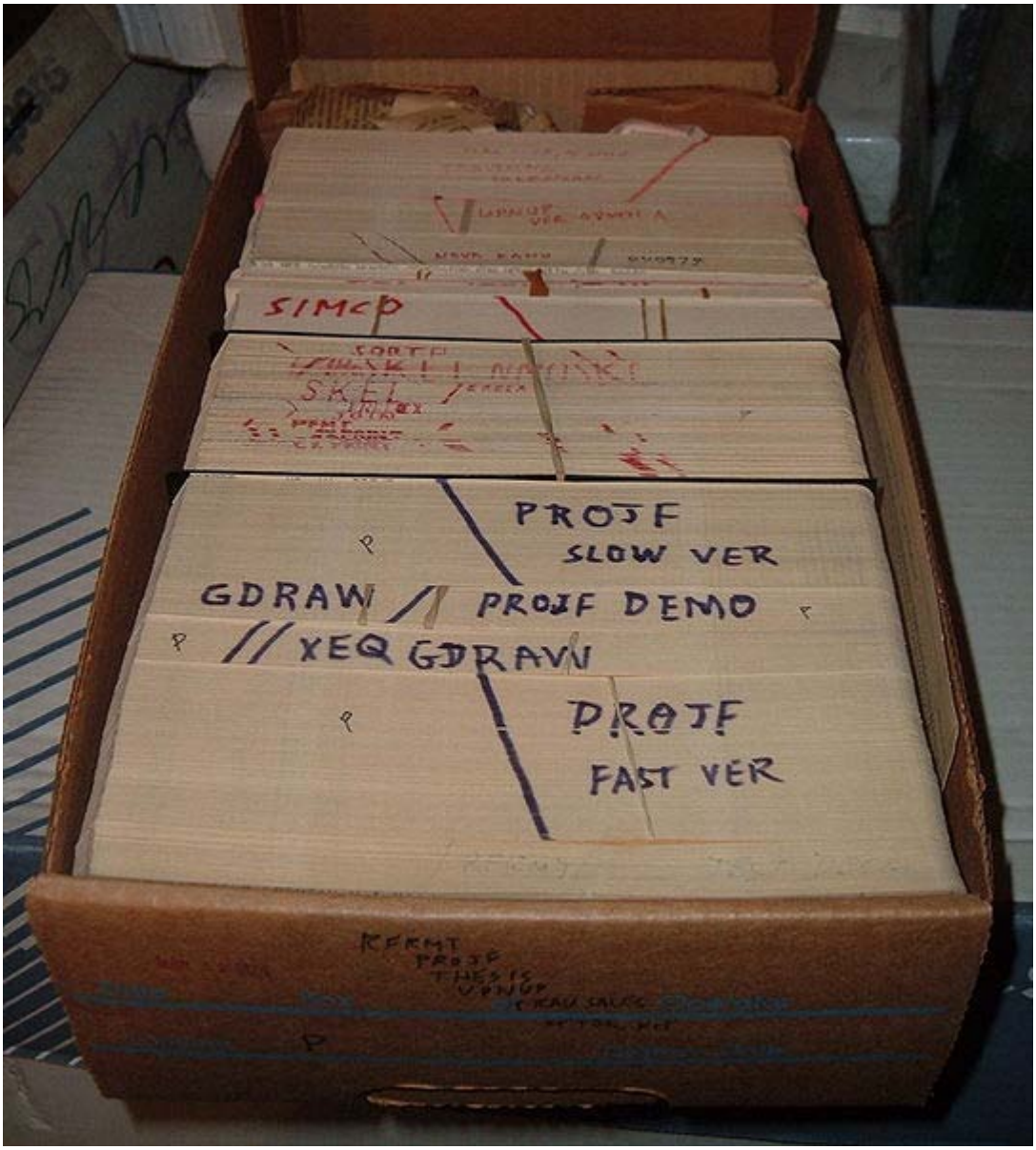


Wired plug board for an IBM 402 Accounting Machine

The Antikythera mechanism from ancient Greece was a calculator utilizing gears of various sizes and configuration to determine its operation, which tracked the metonic cycle still used in lunar-to-solar calendars, and which is consistent for calculating the dates of the Olympiads. Al-Jazari built programmable Automata in 1206. One system employed in these devices was the use of pegs and cams placed into a wooden drum at specific locations. which would sequentially trigger levers that in turn operated percussion instruments. The output of this device was a small drummer playing various rhythms and drum patterns. The Jacquard Loom, which Joseph Marie Jacquard developed in 1801, uses a series of pasteboard cards with holes punched in them. The hole pattern

represented the pattern that the loom had to follow in weaving cloth. The loom could produce entirely different weaves using different sets of cards. Charles Babbage adopted the use of punched cards around 1830 to control his Analytical Engine. The synthesis of numerical calculation, predetermined operation and output, along with a way to organize and input instructions in a manner relatively easy for humans to conceive and produce, led to the modern development of computer programming. Development of computer programming accelerated through the Industrial Revolution.

In the late 1880s, Herman Hollerith invented the recording of data on a medium that could then be read by a machine. Prior uses of machine readable media, above, had been for control, not data. "After some initial trials with paper tape, he settled on punched cards..." To process these punched cards, first known as "Hollerith cards" he invented the tabulator, and the keypunch machines. These three inventions were the foundation of the modern information processing industry. In 1896 he founded the *Tabulating Machine Company* (which later became the core of IBM). The addition of a control panel (plugboard) to his 1906 Type I Tabulator allowed it to do different jobs without having to be physically rebuilt. By the late 1940s, there were a variety of plug-board programmable machines, called unit record equipment, to perform data-processing tasks (card reading). Early computer programmers used plug-boards for the variety of complex calculations requested of the newly invented machines.



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

The invention of the von Neumann architecture allowed computer programs to be stored in computer memory. Early programs had to be painstakingly crafted using the instructions (elementary operations) of the particular machine, often in binary notation. Every model of computer would likely use different instructions (machine language) to do the same task. Later, assembly languages were developed that let the programmer specify each instruction in a text format, entering abbreviations for each operation code instead of a number and specifying addresses in symbolic form (e.g., ADD X, TOTAL). Entering a program in assembly language is usually more convenient, faster, and less prone to human error than using machine language, but 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.

In 1954, FORTRAN was invented; it was the first high level programming language to have a functional implementation, as opposed to just a design on paper. (A high-level language is, in very general terms, any programming language that allows the programmer to write programs in terms that are more abstract than assembly language instructions, i.e. at a level of abstraction "higher" than that of an assembly language.) It allowed programmers to specify calculations by entering a formula directly (e.g. Y = X*2 + 5*X + 9). The program text, or *source*, is converted into machine instructions using a special program called a compiler, which translates the FORTRAN program into machine language. In fact, the name FORTRAN stands for "Formula Translation". Many other languages were developed, including some for commercial programming, such as COBOL. Programs were mostly still entered using punched cards or paper tape. By the late 1960s, data storage devices and computer terminals became inexpensive enough that programs could be created by typing directly into the computers. Text editors were developed that allowed changes and corrections to be made much more easily than with punched cards. (Usually, an error in punching a card meant that the card had to be discarded and a new one punched to replace it.)

As time has progressed, computers have made giant leaps in the area of processing power. This has brought about newer programming languages that are more abstracted from the underlying hardware. Although these high-level languages usually incur greater overhead, the increase in speed of modern computers has made the use of these languages much more practical than in the past. These increasingly abstracted languages typically are easier to learn and allow the programmer to develop applications much more efficiently and with less source code. However, high-level languages are still impractical for a few programs, such as those where low-level hardware control is necessary or where maximum processing speed is vital.

Throughout the second half of the twentieth century, programming was an attractive career in most developed countries. Some forms of programming have been increasingly subject to offshore outsourcing (importing software and services from other countries, usually at a lower wage), making programming career decisions in developed countries more complicated, while increasing economic opportunities in less developed areas. It is unclear how far this trend will continue and how deeply it will impact programmer wages and opportunities.

Modern programming

Quality requirements

Whatever the approach to software development may be, the final program must satisfy some fundamental properties. The following properties are among the most relevant:

- **Efficiency/performance**: the amount 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 correct disposal of some resources, such as cleaning up temporary files and lack of memory leaks.
- **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 not due to programmer error. This includes situations such as incorrect, inappropriate or corrupt data, unavailability of needed resources such as memory, operating system services and network connections, and user error.
- **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 behaviour 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 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.

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, *O(n)*, 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.

Methodologies

The first step in most formal software development projects 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.

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).

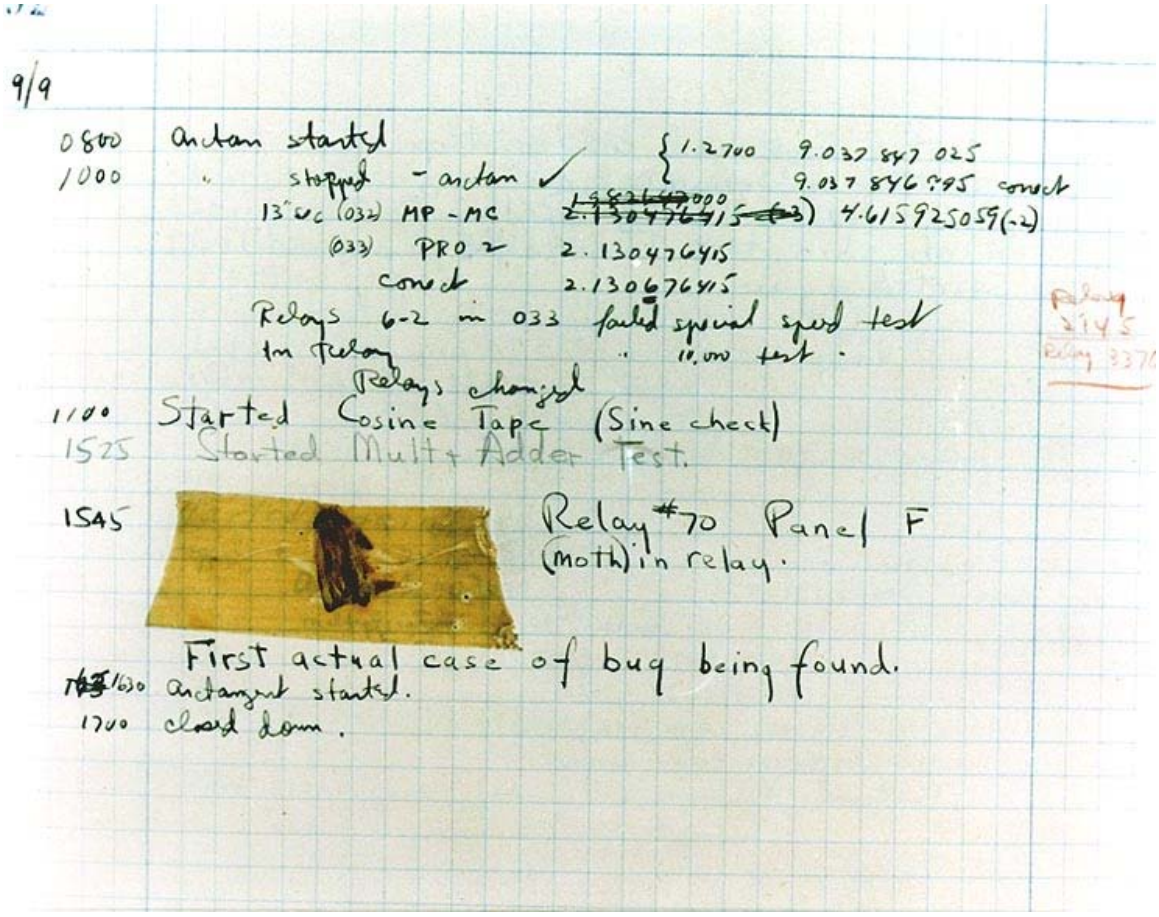
Implementation techniques include imperative languages (object-oriented or procedural), functional languages, and logic languages.

Measuring language usage

It is very difficult to determine what are the most popular of modern programming languages. Some languages are very popular for particular kinds of applications (e.g., COBOL is still strong in the corporate data center, often on large mainframes, FORTRAN in engineering applications, scripting languages in web development, and C in embedded applications), while some languages are regularly used to write many different kinds of applications.

Methods of measuring programming language popularity include: counting the number of job advertisements that mention the language, the number of books teaching the language that are sold (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).

Debugging



A bug, which was debugged in 1947

Debugging is a very important task in the software development process, because an incorrect 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 analysis tool can help detect some possible problems.

Debugging is often done with IDEs like Eclipse, Kdevelop, NetBeans, and Visual Studio. Standalone debuggers like gdb are also used, and these often provide less of a visual environment, usually using a command line.

Chapter 2

Programming Language

A **programming language** is an artificial language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine, to express algorithms precisely, or as a mode of human communication.

Many programming languages have some form of written specification of their syntax (form) and semantics (meaning). Some languages are defined by a specification document. For example, the C programming language is specified by an ISO Standard. Other languages, such as Perl, have a dominant implementation that is used as a reference.

The earliest programming languages predate the invention of the computer, and were used to direct the behavior of machines such as Jacquard looms and player pianos. Thousands of different programming languages have been created, mainly in the computer field, with many more being created every year. Most programming languages describe computation in an imperative style, i.e., as a sequence of commands, although some languages, such as those that support functional programming or logic programming, use alternative forms of description.

Definitions

A programming language is a notation for writing programs, which are specifications of a computation or algorithm. Some, but not all, authors restrict the term "programming language" to those languages that can express *all* possible algorithms. Traits often considered important for what constitutes a programming language include:

- *Function and target:* A *computer programming language* is a language used to write computer programs, which involve a computer performing some kind of computation or algorithm and possibly control external devices such as printers, disk drives, robots, and so on. For example PostScript programs are frequently created by another program to control a computer printer or display. More generally, a programming language may describe computation on some, possibly abstract, machine. It is generally accepted that a complete specification for a programming language includes a description, possibly idealized, of a machine or

processor for that language. In most practical contexts, a programming language involves a computer; consequently programming languages are usually defined and studied this way. Programming languages differ from natural languages in that natural languages are only used for interaction between people, while programming languages also allow humans to communicate instructions to machines.

- *Abstractions:* Programming languages usually contain abstractions for defining and manipulating data structures or controlling the flow of execution. The practical necessity that a programming language support adequate abstractions is expressed by the abstraction principle; this principle is sometimes formulated as recommendation to the programmer to make proper use of such abstractions.
- *Expressive power:* The theory of computation classifies languages by the computations they are capable of expressing. All Turing complete languages can implement the same set of algorithms. ANSI/ISO SQL and Charity are examples of languages that are not Turing complete, yet often called programming languages.

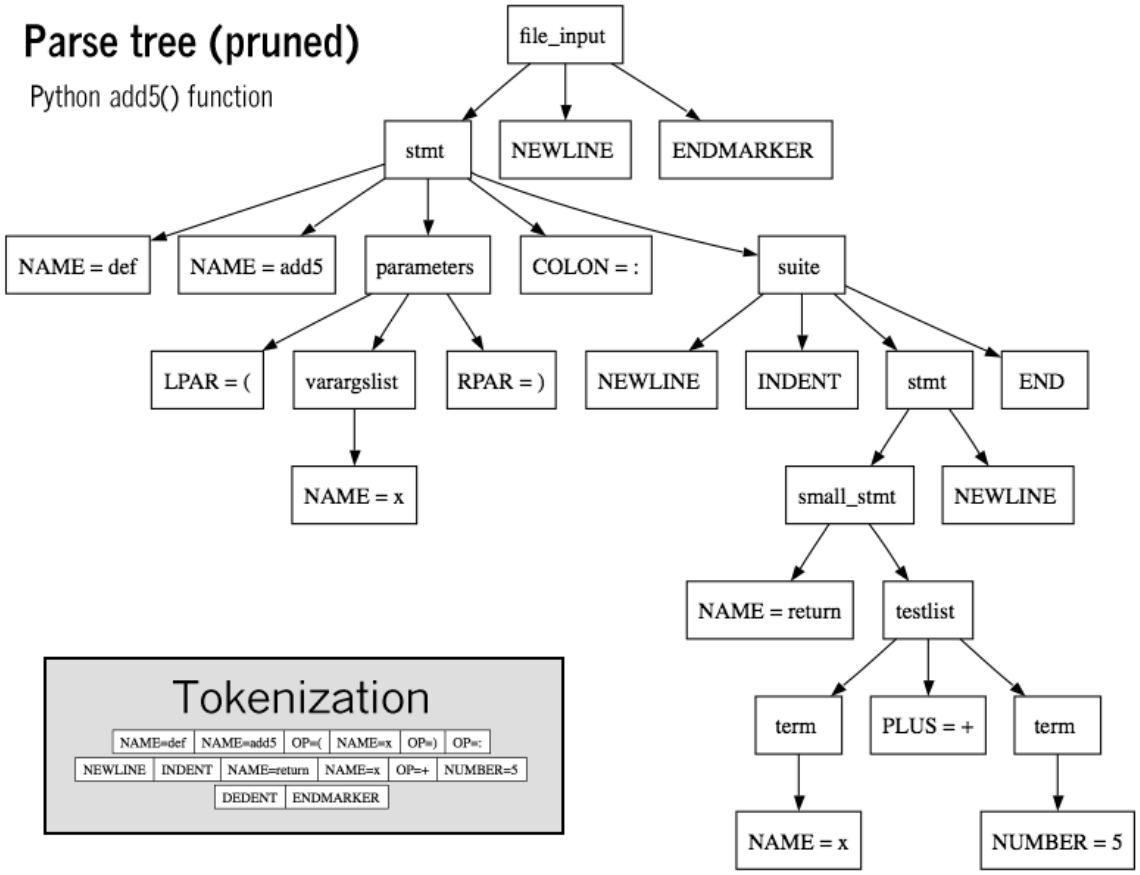
Markup languages like XML, HTML or troff, which define structured data, are not generally considered programming languages. Programming languages may, however, share the syntax with markup languages if a computational semantics is defined. XSLT, for example, is a Turing complete XML dialect. Moreover, LaTeX, which is mostly used for structuring documents, also contains a Turing complete subset.

The term *computer language* is sometimes used interchangeably with programming language. However, the usage of both terms varies among authors, including the exact scope of each. One usage describes programming languages as a subset of computer languages. In this vein, languages used in computing that have a different goal than expressing computer programs are generically designated computer languages. For instance, markup languages are sometimes referred to as computer languages to emphasize that they are not meant to be used for programming. Another usage regards programming languages as theoretical constructs for programming abstract machines, and computer languages as the subset thereof that runs on physical computers, which have finite hardware resources. John C. Reynolds emphasizes that formal specification languages are just as much programming languages as are the languages intended for execution. He also argues that textual and even graphical input formats that affect the behavior of a computer are programming languages, despite the fact they are commonly not Turing-complete, and remarks that ignorance of programming language concepts is the reason for many flaws in input formats.

Elements

All programming languages have some primitive building blocks for the description of data and the processes or transformations applied to them (like the addition of two numbers or the selection of an item from a collection). These primitives are defined by syntactic and semantic rules which describe their structure and meaning respectively.

Syntax



Parse tree of Python code with inset tokenization

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s' % (nodename, label),
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
    else:
      print '"];'
      children = []
      for  in n, childenumerate(ast[1:]):
         children.append(dotwrite(child))
      print ,'    %s -> {' % nodename
      for  in :namechildren
         print '%s' % name,
```

Syntax highlighting is often used to aid programmers in recognizing elements of source code. The language above is Python.

A programming language's surface form is known as its syntax. Most programming languages are purely textual; they use sequences of text including words, numbers, and punctuation, much like written natural languages. On the other hand, there are some programming languages which are more graphical in nature, using visual relationships between symbols to specify a program.

The syntax of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation). Since most languages are textual, here we discusses textual syntax.

Programming language syntax is usually defined using a combination of regular expressions (for lexical structure) and Backus–Naur Form (for grammatical structure). Below is a simple grammar, based on Lisp:

```
expression ::= atom | list
atom ::= number | symbol
number ::= [+-]?['0'-'9']+
symbol ::= ['A'-'Z''a'-'z'].*
list ::= '(' expression* ')'
```

This grammar specifies the following:

- an *expression* is either an *atom* or a *list*;
- an *atom* is either a *number* or a *symbol*;
- a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- a *symbol* is a letter followed by zero or more of any characters (excluding whitespace); and
- a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

The following are examples of well-formed token sequences in this grammar: '`12345`', '`()`','`(a b c232 (1))`'

Not all syntactically correct programs are semantically correct. Many syntactically correct programs are nonetheless ill-formed, per the language's rules; and may (depending on the language specification and the soundness of the implementation) result in an error on translation or execution. In some cases, such programs may exhibit undefined behavior. Even when a program is well-defined within a language, it may still have a meaning that is not intended by the person who wrote it.

Using natural language as an example, it may not be possible to assign a meaning to a grammatically correct sentence or the sentence may be false:

- "Colorless green ideas sleep furiously." is grammatically well-formed but has no generally accepted meaning.
- "John is a married bachelor." is grammatically well-formed but expresses a meaning that cannot be true.

The following C language fragment is syntactically correct, but performs an operation that is not semantically defined (because `p` is a null pointer, the operations `p->real` and `p->im` have no meaning):

```
complex *p = NULL;
complex abs_p = sqrt (p->real * p->real + p->im * p->im);
```

If the type declaration on the first line were omitted, the program would trigger an error on compilation, as the variable "p" would not be defined. But the program would still be syntactically correct, since type declarations provide only semantic information.

The grammar needed to specify a programming language can be classified by its position in the Chomsky hierarchy. The syntax of most programming languages can be specified using a Type-2 grammar, i.e., they are context-free grammars. Some languages, including Perl and Lisp, contain constructs that allow execution during the parsing phase. Languages that have constructs that allow the programmer to alter the behavior of the parser make syntax analysis an undecidable problem, and generally blur the distinction between parsing and execution. In contrast to Lisp's macro system and Perl's `BEGIN`

blocks, which may contain general computations, C macros are merely string replacements, and do not require code execution.

Static semantics

The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms. For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time. Examples include checking that every identifier is declared before it is used (in languages that require such declarations) or that the labels on the arms of a case statement are distinct. Many important restrictions of this type, like checking that identifiers are used in the appropriate context (e.g. not adding a integer to a function name), or that subroutine calls have the appropriate number and type of arguments can be enforced by defining them as rules in a logic called a type system. Other forms of static analyses like data flow analysis may also be part of static semantics. Newer programming languages like Java and C# have definite assignment analysis, a form of data flow analysis, as part of their static semantics.

Type system

A type system defines how a programming language classifies values and expressions into *types*, how it can manipulate those types and how they interact. The goal of a type system is to verify and usually enforce a certain level of correctness in programs written in that language by detecting certain incorrect operations. Any decidable type system involves a trade-off: while it rejects many incorrect programs, it can also prohibit some correct, albeit unusual programs. In order to bypass this downside, a number of languages have *type loopholes*, usually unchecked casts that may be used by the programmer to explicitly allow a normally disallowed operation between different types. In most typed languages, the type system is used only to type check programs, but a number of languages, usually functional ones, perform type inference, which relieves the programmer from writing type annotations. The formal design and study of type systems is known as *type theory*.

Typed versus untyped languages

A language is *typed* if the specification of every operation defines types of data to which the operation is applicable, with the implication that it is not applicable to other types. For example, "`this text between the quotes`" is a string. In most programming languages, dividing a number by a string has no meaning. Most modern programming languages will therefore reject any program attempting to perform such an operation. In some languages, the meaningless operation will be detected when the program is compiled ("static" type checking), and rejected by the compiler, while in others, it will be detected when the program is run ("dynamic" type checking), resulting in a runtime exception.

A special case of typed languages are the *single-type* languages. These are often scripting or markup languages, such as REXX or SGML, and have only one data type—most commonly character strings which are used for both symbolic and numeric data.

In contrast, an *untyped language*, such as most assembly languages, allows any operation to be performed on any data, which are generally considered to be sequences of bits of various lengths. High-level languages which are untyped include BCPL and some varieties of Forth.

In practice, while few languages are considered typed from the point of view of type theory (verifying or rejecting *all* operations), most modern languages offer a degree of typing. Many production languages provide means to bypass or subvert the type system.

Static versus dynamic typing

In *static typing* all expressions have their types determined prior to the program being run (typically at compile-time). For example, 1 and (2+2) are integer expressions; they cannot be passed to a function that expects a string, or stored in a variable that is defined to hold dates.

Statically typed languages can be either *manifestly typed* or *type-inferred*. In the first case, the programmer must explicitly write types at certain textual positions (for example, at variable declarations). In the second case, the compiler *infers* the types of expressions and declarations based on context. Most mainstream statically typed languages, such as C++, C# and Java, are manifestly typed. Complete type inference has traditionally been associated with less mainstream languages, such as Haskell and ML. However, many manifestly typed languages support partial type inference; for example, Java and C# both infer types in certain limited cases.

Dynamic typing, also called *latent typing*, determines the type-safety of operations at runtime; in other words, types are associated with *runtime values* rather than *textual expressions*. As with type-inferred languages, dynamically typed languages do not require the programmer to write explicit type annotations on expressions. Among other things, this may permit a single variable to refer to values of different types at different points in the program execution. However, type errors cannot be automatically detected until a piece of code is actually executed, making debugging more difficult. Ruby, Lisp, JavaScript, and Python are dynamically typed.

Weak and strong typing

Weak typing allows a value of one type to be treated as another, for example treating a string as a number. This can occasionally be useful, but it can also allow some kinds of program faults to go undetected at compile time and even at runtime.

Strong typing prevents the above. An attempt to perform an operation on the wrong type of value raises an error. Strongly typed languages are often termed *type-safe* or *safe*.

An alternative definition for "weakly typed" refers to languages, such as Perl and JavaScript, which permit a large number of implicit type conversions. In JavaScript, for example, the expression `2 * x` implicitly converts `x` to a number, and this conversion succeeds even if `x` is `null`, `undefined`, an `Array`, or a string of letters. Such implicit conversions are often useful, but they can mask programming errors.

Strong and *static* are now generally considered orthogonal concepts, but usage in the literature differs. Some use the term *strongly typed* to mean *strongly, statically typed*, or, even more confusingly, to mean simply *statically typed*. Thus C has been called both strongly typed and weakly, statically typed.

Execution semantics

Once data has been specified, the machine must be instructed to perform operations on the data. For example, the semantics may define the strategy by which expressions are evaluated to values, or the manner in which control structures conditionally execute statements. The *execution semantics* (also known as *dynamic semantics*) of a language defines how and when the various constructs of a language should produce a program behavior. There are many ways of defining execution semantics. Natural language is often used to specify the execution semantics of languages commonly used in practice. A significant amount of academic research went into formal semantics of programming languages, which allow execution semantics to be specified in a formal manner. Results from this field of research have seen limited application to programming language design and implementation outside academia.

Core library

Most programming languages have an associated core library (sometimes known as the 'standard library', especially if it is included as part of the published language standard), which is conventionally made available by all implementations of the language. Core libraries typically include definitions for commonly used algorithms, data structures, and mechanisms for input and output.

A language's core library is often treated as part of the language by its users, although the designers may have treated it as a separate entity. Many language specifications define a core that must be made available in all implementations, and in the case of standardized languages this core library may be required. The line between a language and its core library therefore differs from language to language. Indeed, some languages are designed so that the meanings of certain syntactic constructs cannot even be described without referring to the core library. For example, in Java, a string literal is defined as an instance of the `java.lang.String` class; similarly, in Smalltalk, an anonymous function expression (a "block") constructs an instance of the library's `BlockContext` class. Conversely, Scheme contains multiple coherent subsets that suffice to construct the rest of the language as library macros, and so the language designers do not even bother to say which portions of the language must be implemented as language constructs, and which must be implemented as parts of a library.

Design and implementation

Programming languages share properties with natural languages related to their purpose as vehicles for communication, having a syntactic form separate from its semantics, and showing *language families* of related languages branching one from another. But as artificial constructs, they also differ in fundamental ways from languages that have evolved through usage. A significant difference is that a programming language can be fully described and studied in its entirety, since it has a precise and finite definition. By contrast, natural languages have changing meanings given by their users in different communities. While constructed languages are also artificial languages designed from the ground up with a specific purpose, they lack the precise and complete semantic definition that a programming language has.

Many languages have been designed from scratch, altered to meet new needs, combined with other languages, and eventually fallen into disuse. Although there have been attempts to design one "universal" programming language that serves all purposes, all of them have failed to be generally accepted as filling this role. The need for diverse programming languages arises from the diversity of contexts in which languages are used:

- Programs range from tiny scripts written by individual hobbyists to huge systems written by hundreds of programmers.
- Programmers range in expertise from novices who need simplicity above all else, to experts who may be comfortable with considerable complexity.
- Programs must balance speed, size, and simplicity on systems ranging from microcontrollers to supercomputers.
- Programs may be written once and not change for generations, or they may undergo continual modification.
- Finally, programmers may simply differ in their tastes: they may be accustomed to discussing problems and expressing them in a particular language.

One common trend in the development of programming languages has been to add more ability to solve problems using a higher level of abstraction. The earliest programming languages were tied very closely to the underlying hardware of the computer. As new programming languages have developed, features have been added that let programmers express ideas that are more remote from simple translation into underlying hardware instructions. Because programmers are less tied to the complexity of the computer, their programs can do more computing with less effort from the programmer. This lets them write more functionality per time unit.

Natural language processors have been proposed as a way to eliminate the need for a specialized language for programming. However, this goal remains distant and its benefits are open to debate. Edsger W. Dijkstra took the position that the use of a formal language is essential to prevent the introduction of meaningless constructs, and dismissed natural language programming as "foolish". Alan Perlis was similarly dismissive of the idea. Hybrid approaches have been taken in Structured English and SQL.

A language's designers and users must construct a number of artifacts that govern and enable the practice of programming. The most important of these artifacts are the language *specification* and *implementation*.

Specification

The **specification** of a programming language is intended to provide a definition that the language users and the implementors can use to determine whether the behavior of a program is correct, given its source code.

A programming language specification can take several forms, including the following:

- An explicit definition of the syntax, static semantics, and execution semantics of the language. While syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language (e.g., as in the C language), or a formal semantics (e.g., as in Standard ML and Scheme specifications).
- A description of the behavior of a translator for the language (e.g., the C++ and Fortran specifications). The syntax and semantics of the language have to be inferred from this description, which may be written in natural or a formal language.
- A *reference* or *model* implementation, sometimes written in the language being specified (e.g., Prolog or ANSI REXX). The syntax and semantics of the language are explicit in the behavior of the reference implementation.

Implementation

An **implementation** of a programming language provides a way to execute that program on one or more configurations of hardware and software. There are, broadly, two approaches to programming language implementation: *compilation* and *interpretation*. It is generally possible to implement a language using either technique.

The output of a compiler may be executed by hardware or a program called an interpreter. In some implementations that make use of the interpreter approach there is no distinct boundary between compiling and interpreting. For instance, some implementations of BASIC compile and then execute the source a line at a time.

Programs that are executed directly on the hardware usually run several orders of magnitude faster than those that are interpreted in software.

One technique for improving the performance of interpreted programs is just-in-time compilation. Here the virtual machine, just before execution, translates the blocks of bytecode which are going to be used to machine code, for direct execution on the hardware.

Usage

Thousands of different programming languages have been created, mainly in the computing field. Programming languages differ from most other forms of human expression in that they require a greater degree of precision and completeness. When using a natural language to communicate with other people, human authors and speakers can be ambiguous and make small errors, and still expect their intent to be understood. However, figuratively speaking, computers "do exactly what they are told to do", and cannot "understand" what code the programmer intended to write. The combination of the language definition, a program, and the program's inputs must fully specify the external behavior that occurs when the program is executed, within the domain of control of that program.

A programming language provides a structured mechanism for defining pieces of data, and the operations or transformations that may be carried out automatically on that data. A programmer uses the abstractions present in the language to represent the concepts involved in a computation. These concepts are represented as a collection of the simplest elements available (called primitives). *Programming* is the process by which programmers combine these primitives to compose new programs, or adapt existing ones to new uses or a changing environment.

Programs for a computer might be executed in a batch process without human interaction, or a user might type commands in an interactive session of an interpreter. In this case the "commands" are simply programs, whose execution is chained together. When a language is used to give commands to a software application (such as a shell) it is called a scripting language.

Measuring language usage

It is difficult to determine which programming languages are most widely used, and what usage means varies by context. One language may occupy the greater number of programmer hours, a different one have more lines of code, and a third utilize the most CPU time. Some languages are very popular for particular kinds of applications. For example, COBOL is still strong in the corporate data center, often on large mainframes; FORTRAN in engineering applications; C in embedded applications and operating systems; and other languages are regularly used to write many different kinds of applications.

Various methods of measuring language popularity, each subject to a different bias over what is measured, have been proposed:

- counting the number of job advertisements that mention the language
- the number of books sold that teach or describe the language
- estimates of the number of existing lines of code written in the language—which may underestimate languages not often found in public searches

- counts of language references (i.e., to the name of the language) found using a web search engine.

Combining and averaging information from various internet sites, langpop.com claims that in 2008 the 10 most cited programming languages are (in alphabetical order): C, C++, C#, Java, JavaScript, Perl, PHP, Python, Ruby, and SQL.

Taxonomies

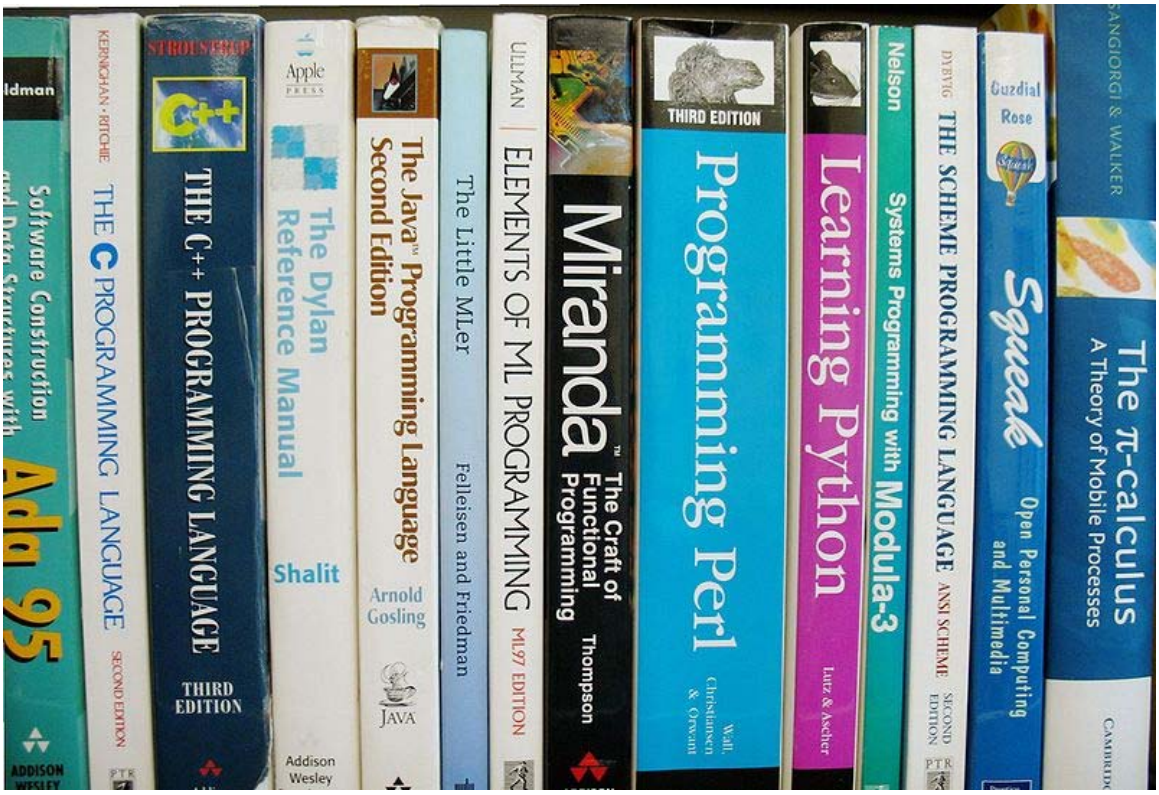
There is no overarching classification scheme for programming languages. A given programming language does not usually have a single ancestor language. Languages commonly arise by combining the elements of several predecessor languages with new ideas in circulation at the time. Ideas that originate in one language will diffuse throughout a family of related languages, and then leap suddenly across familial gaps to appear in an entirely different family.

The task is further complicated by the fact that languages can be classified along multiple axes. For example, Java is both an object-oriented language (because it encourages object-oriented organization) and a concurrent language (because it contains built-in constructs for running multiple threads in parallel). Python is an object-oriented scripting language.

In broad strokes, programming languages divide into *programming paradigms* and a classification by *intended domain of use*. Traditionally, programming languages have been regarded as describing computation in terms of imperative sentences, i.e. issuing commands. These are generally called imperative programming languages. A great deal of research in programming languages has been aimed at blurring the distinction between a program as a set of instructions and a program as an assertion about the desired answer, which is the main feature of declarative programming. More refined paradigms include procedural programming, object-oriented programming, functional programming, and logic programming; some languages are hybrids of paradigms or multi-paradigmatic. An assembly language is not so much a paradigm as a direct model of an underlying machine architecture. By purpose, programming languages might be considered general purpose, system programming languages, scripting languages, domain-specific languages, or concurrent/distributed languages (or a combination of these). Some general purpose languages were designed largely with educational goals.

A programming language may also be classified by factors unrelated to programming paradigm. For instance, most programming languages use English language keywords, while a minority do not. Other languages may be classified as being esoteric or not.

History



A selection of textbooks that teach programming, in languages both popular and obscure. These are only a few of the thousands of programming languages and dialects that have been designed in history.

Early developments

The first programming languages predate the modern computer. The 19th century had "programmable" looms and player piano scrolls which implemented what are today recognized as examples of domain-specific languages. By the beginning of the twentieth century, punch cards encoded data and directed mechanical processing. In the 1930s and 1940s, the formalisms of Alonzo Church's lambda calculus and Alan Turing's Turing machines provided mathematical abstractions for expressing algorithms; the lambda calculus remains influential in language design.

In the 1940s, the first electrically powered digital computers were created. The first high-level programming language to be designed for a computer was Plankalkül, developed for the German Z3 by Konrad Zuse between 1943 and 1945. However, it was not implemented until 1998 and 2000.

Programmers of early 1950s computers, notably UNIVAC I and IBM 701, used machine language programs, that is, the first generation language (1GL). 1GL programming was

quickly superseded by similarly machine-specific, but mnemonic, second generation languages (2GL) known as assembly languages or "assembler". Later in the 1950s, assembly language programming, which had evolved to include the use of macro instructions, was followed by the development of "third generation" programming languages (3GL), such as FORTRAN, LISP, and COBOL. 3GLs are more abstract and are "portable", or at least implemented similarly on computers that do not support the same native machine code. Updated versions of all of these 3GLs are still in general use, and each has strongly influenced the development of later languages. At the end of the 1950s, the language formalized as ALGOL 60 was introduced, and most later programming languages are, in many respects, descendants of Algol. The format and use of the early programming languages was heavily influenced by the constraints of the interface.

Refinement

The period from the 1960s to the late 1970s brought the development of the major language paradigms now in use, though many aspects were refinements of ideas in the very first Third-generation programming languages:

- APL introduced *array programming* and influenced functional programming.
- PL/I (NPL) was designed in the early 1960s to incorporate the best ideas from FORTRAN and COBOL.
- In the 1960s, Simula was the first language designed to support *object-oriented programming*; in the mid-1970s, Smalltalk followed with the first "purely" object-oriented language.
- C was developed between 1969 and 1973 as a *system programming* language, and remains popular.
- Prolog, designed in 1972, was the first *logic programming* language.
- In 1978, ML built a polymorphic type system on top of Lisp, pioneering *statically typed functional programming* languages.

Each of these languages spawned an entire family of descendants, and most modern languages count at least one of them in their ancestry.

The 1960s and 1970s also saw considerable debate over the merits of *structured programming*, and whether programming languages should be designed to support it. Edsger Dijkstra, in a famous 1968 letter published in the Communications of the ACM, argued that GOTO statements should be eliminated from all "higher level" programming languages.

The 1960s and 1970s also saw expansion of techniques that reduced the footprint of a program as well as improved productivity of the programmer and user. The card deck for an early 4GL was a lot smaller for the same functionality expressed in a 3GL deck.

Consolidation and growth

The 1980s were years of relative consolidation. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems programming language derived from Pascal and intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called "fifth generation" languages that incorporated logic programming constructs. The functional languages community moved to standardize ML and Lisp. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade.

One important trend in language design for programming large-scale systems during the 1980s was an increased focus on the use of *modules*, or large-scale organizational units of code. Modula-2, Ada, and ML all developed notable module systems in the 1980s, although other languages, such as PL/I, already had extensive support for modular programming. Module systems were often wedded to generic programming constructs.

The rapid growth of the Internet in the mid-1990s created opportunities for new languages. Perl, originally a Unix scripting tool first released in 1987, became common in dynamic websites. Java came to be used for server-side programming, and bytecode virtual machines became popular again in commercial settings with their promise of "Write once, run anywhere" (UCSD Pascal had been popular for a time in the early 1980s). These developments were not fundamentally novel, rather they were refinements to existing languages and paradigms, and largely based on the C family of programming languages.

Programming language evolution continues, in both industry and research. Current directions include security and reliability verification, new kinds of modularity (mixins, delegates, aspects), and database integration such as Microsoft's LINQ.

The 4GLs are examples of languages which are domain-specific, such as SQL, which manipulates and returns sets of data rather than the scalar values which are canonical to most programming languages. Perl, for example, with its 'here document' can hold multiple 4GL programs, as well as multiple JavaScript programs, in part of its own perl code and use variable interpolation in the 'here document' to support multi-language programming.

Chapter 3

Programmer

A **programmer**, **computer programmer** or **coder** is someone who writes computer software. The term *computer programmer* can refer to a specialist in one area of computer programming or to a generalist who writes code for many kinds of software. One who practices or professes a formal approach to programming may also be known as a programmer analyst. A programmer's primary computer language (C, C++, Java, Lisp, Delphi etc.) is often prefixed to the above titles, and those who work in a web environment often prefix their titles with *web*. The term *programmer* can be used to refer to a software developer, software engineer, computer scientist, or software analyst. However, members of these professions typically possess other software engineering skills, beyond programming; for this reason, the term *programmer* is sometimes considered an insulting or derogatory oversimplification of these other professions. This has sparked much debate amongst developers, analysts, computer scientists, programmers, and outsiders who continue to be puzzled at the subtle differences in these occupations.

Ada Lovelace is popularly credited as history's first programmer. She was the first to express an algorithm intended for implementation on a computer, Charles Babbage's analytical engine, in October 1842. Her work never ran, though that of Konrad Zuse did in 1941. The ENIAC programming team, consisting of Kay McNulty, Betty Jennings, Betty Snyder, Marlyn Wescoff, Fran Bilas and Ruth Lichterman were the first working programmers.

In 2009, the government of Russia decreed a professional annual holiday known as Programmers' Day to be celebrated on September 13 (September 12 in leap year). It had also been an unofficial international holiday before that.

Nature of the work

Computer programmers write, test, debug, and maintain the detailed instructions, called computer programs, that computers must follow to perform their functions. Programmers also conceive, design, and test logical structures for solving problems by computer. Many technical innovations in programming — advanced computing technologies and sophisticated new languages and programming tools — have redefined the role of a

programmer and elevated much of the programming work done today. Job titles and descriptions may vary, depending on the organization.

Programmers work in many settings, including corporate information technology departments, big software companies, and small service firms. Many professional programmers also work for consulting companies at client' sites as contractors. Licensing is not typically required to work as a programmer, although professional certifications are commonly held by programmers. Programming is widely considered a profession (although some authorities disagree on the grounds that only careers with legal licensing requirements count as a profession).

Programmers' work varies widely depending on the type of business they are writing programs for. For example, the instructions involved in updating financial records are very different from those required to duplicate conditions on an aircraft for pilots training in a flight simulator. Although simple programs can be written in a few hours, programs that use complex mathematical formulas whose solutions can only be approximated or that draw data from many existing systems may require more than a year of work. In most cases, several programmers work together as a team under a senior programmer’s supervision.

Programmers write programs according to the specifications determined primarily by more senior programmers and by systems analysts. After the design process is complete, it is the job of the programmer to convert that design into a logical series of instructions that the computer can follow. The programmer codes these instructions in one of many programming languages. Different programming languages are used depending on the purpose of the program. COBOL, for example, is commonly used for business applications which are run on mainframe and midrange computers, whereas Fortran is used in science and engineering. C++ is widely used for both scientific and business applications. Java, C# and PHP are popular programming languages for Web and business applications. Programmers generally know more than one programming language and, because many languages are similar, they often can learn new languages relatively easily. In practice, programmers often are referred to by the language they know, e.g. as *Java programmers*, or by the type of function they perform or environment in which they work: for example, *database programmers*, *mainframe programmers*, or Web developers.

When making changes to the source code that programs are made up of, programmers need to make other programmers aware of the task that the routine is to perform. They do this by inserting comments in the source code so that others can understand the program more easily. To save work, programmers often use libraries of basic code that can be modified or customized for a specific application. This approach yields more reliable and consistent programs and increases programmers' productivity by eliminating some routine steps.

Testing and debugging

Programmers test a program by running it and looking for bugs. As they are identified, the programmer usually makes the appropriate corrections, then rechecks the program until an acceptably low level and severity of bugs remain. This process is called testing and debugging. These are important parts of every programmer's job. Programmers may continue to fix these problems throughout the life of a program. Updating, repairing, modifying, and expanding existing programs sometimes called *maintenance programmer*. Programmers may contribute to user guides and online help, or they may work with technical writers to do such work.

Certain scenarios or execution paths may be difficult to test, in which case the programmer may elect to test by inspection which involves a human inspecting the code on the relevant execution path, perhaps hand executing the code. Test by inspection is also sometimes used as a euphemism for inadequate testing. It may be difficult to properly assess whether the term is being used euphemistically.

Application versus system programming

Computer programmers often are grouped into two broad types: application programmers and systems programmers. Application programmers write programs to handle a specific job, such as a program to track inventory within an organization. They also may revise existing packaged software or customize generic applications which are frequently purchased from independent software vendors. Systems programmers, in contrast, write programs to maintain and control computer systems software, such as operating systems and database management systems. These workers make changes in the instructions that determine how the network, workstations, and CPU of the system handle the various jobs they have been given and how they communicate with peripheral equipment such as printers and disk drives.

Types of software

Programmers in software development companies may work directly with experts from various fields to create software — either programs designed for specific clients or packaged software for general use — ranging from computer and video games to educational software to programs for desktop publishing and financial planning. Programming of packaged software constitutes one of the most rapidly growing segments of the computer services industry.

In some organizations, particularly small ones, workers commonly known as *programmer analysts* are responsible for both the systems analysis and the actual programming work. The transition from a mainframe environment to one that is based primarily on personal computers (PCs) has blurred the once rigid distinction between the programmer and the user. Increasingly, adept end users are taking over many of the tasks previously performed by programmers. For example, the growing use of packaged

software, such as spreadsheet and database management software packages, allows users to write simple programs to access data and perform calculations.

In addition, the rise of the Internet has made Web development a huge part of the programming field. More and more software applications nowadays are Web applications that can be used by anyone with a Web browser. Examples of such applications include the Google search service, the Hotmail e-mail service, and the Flickr photo-sharing service.

Types of Computer Program

Source code

```
/**
 * Simple HelloButton() method.
 * @version 1.0
 * @author john doe <doe.j@example.com>
 */
HelloButton()
{
  JButton hello = new JButton( "Hello, wor
  hello.addActionListener( new HelloBtnList

  // use the JFrame type until support for t
  // new component is finished
  JFrame frame = new JFrame( "Hello Button"
  Container pane = frame.getContentPane();
  pane.add( hello );
  frame.pack();

} frame.show();           // display the fra
```

An illustration of Java source code with prologue comments indicated in red, inline comments indicated in green, and program code indicated in blue.

In computer science, **source code** is any collection of statements or declarations written in some human-readable computer programming language. Source code is the means most often used by programmers to specify the actions to be performed by a computer.

The source code which constitutes a program is usually held in one or more text files, sometimes stored in databases as stored procedures and may also appear as code snippets printed in books or other media. A large collection of source code files may be organized into a directory tree, in which case it may also be known as a **source tree**.

A computer program's source code is the collection of files needed to convert from human-readable form to some kind of computer-executable form. The source code may be converted into an executable file by a compiler, or executed on the fly from the human readable form with the aid of an interpreter.

The annual working conference Source Code Analysis and Manipulation defines source code as

"For the purpose of clarity '**source code**' is taken to mean any fully executable description of a software system. It is therefore so construed as to include machine code, very high level languages and executable graphical representations of systems. "

The **code base** of a programming project is the larger collection of all the source code of all the computer programs which make up the project.

Organization

The source code for a particular piece of software may be contained in a single file or many files. Though the practice is uncommon, a program's source code can be written in different programming languages. For example, a program written primarily in the C programming language, might have portions written in assembly language for optimization purposes. It is also possible for some components of a piece of software to be written and compiled separately, in an arbitrary programming language, and later integrated into the software using a technique called library linking. This is the case in some languages, such as Java: each class is compiled separately into a file and linked by the interpreter at runtime.

Yet another method is to make the main program an interpreter for a programming language, either designed specifically for the application in question or general-purpose, and then write the bulk of the actual user functionality as macros or other forms of add-ins in this language, an approach taken for example by the GNU Emacs text editor.

Moderately complex software customarily requires the compilation or assembly of several, sometimes dozens or even hundreds, of different source code files. In these cases, instructions for compilations, such as a Makefile, are included with the source code. These describe the relationships among the source code files, and contain information about how they are to be compiled.

The revision control system is another tool frequently used by developers for source code maintenance.

Purposes

Source code is primarily used as input to the process that produces an executable program (i.e., it is compiled or interpreted). It is also used as a method of communicating algorithms between people (e.g., code snippets in books).

Programmers often find it helpful to review existing source code to learn about programming techniques. The sharing of source code between developers is frequently cited as a contributing factor to the maturation of their programming skills. Some people consider source code an expressive artistic medium.

Porting software to other computer platforms is usually prohibitively difficult without source code. Without the source code for a particular piece of software, portability is generally computationally expensive. Possible porting options include binary translation and emulation of the original platform.

Decompilation of an executable program can be used to generate source code, either in assembly code or in a high level language.

Programmers frequently adapt source code from one piece of software to use in other projects, a concept known as software reusability.

Licensing

Software, and its accompanying source code, typically falls within one of two licensing paradigms: free software and proprietary software.

Generally speaking, software is *free* if the source code is free to use, distribute, modify and study, and *proprietary* if the source code is kept secret, or is privately owned and restricted. Note that "free" refers to freedom, not price. Under many licenses it is acceptable to charge for "free software". The first free software license to be published and to explicitly grant these freedoms was the GNU General Public License in 1989. The GNU GPL was originally intended to be used with the GNU operating system. The GNU GPL was later adopted by other non-GNU software projects such as the Linux kernel.

For proprietary software, the provisions of the various copyright laws, trade secrecy and patents are used to keep the source code closed. Additionally, many pieces of retail software come with an end-user license agreement (EULA) which typically prohibits decompilation, reverse engineering, analysis, modification, or circumventing of copy protection. Types of source code protection — beyond traditional compilation to object code — include code encryption, code obfuscation or code morphing.

Legal issues in the United States

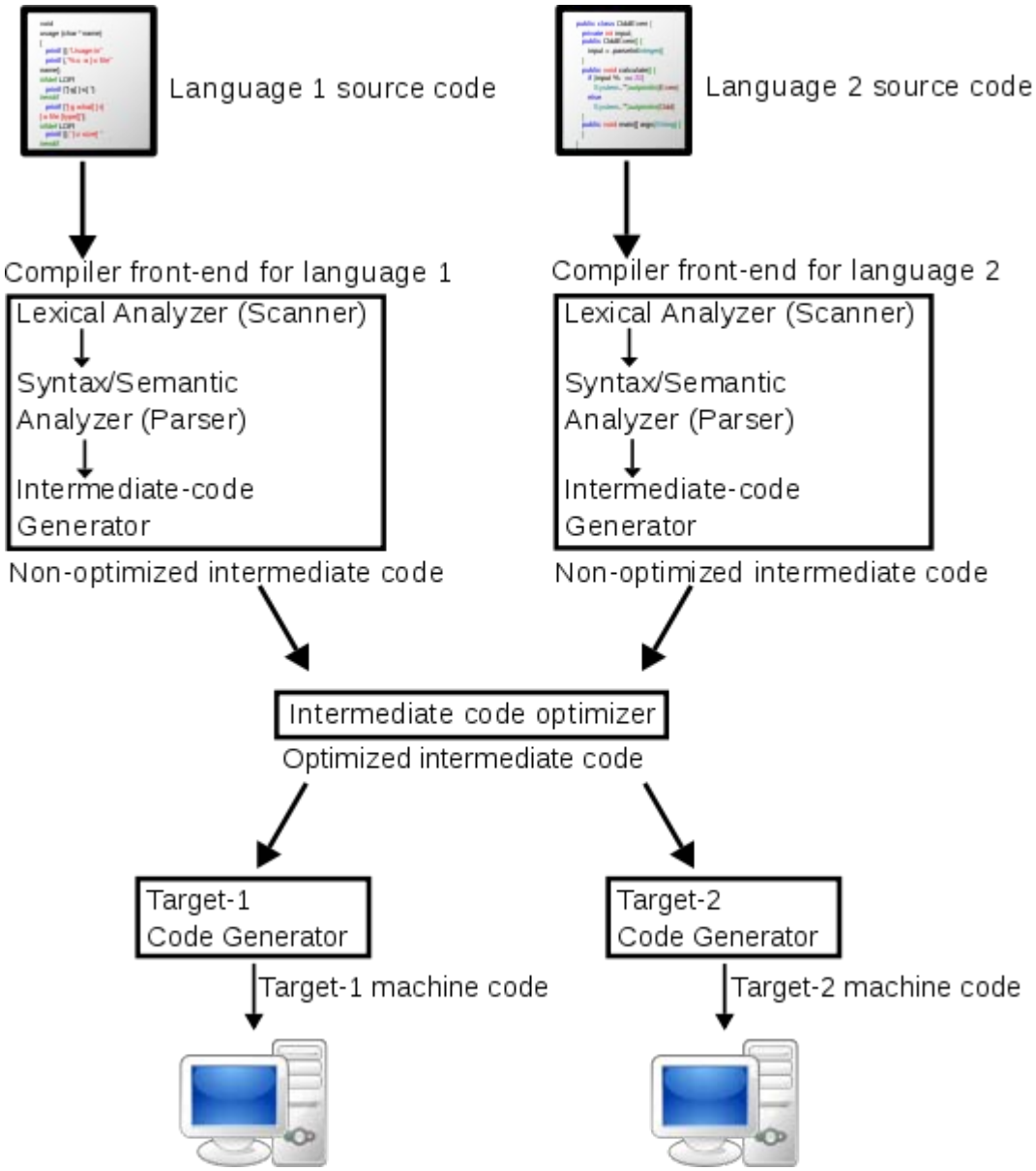
In a 2003 court case in the United States, it was ruled that source code should be considered a constitutionally protected form of free speech. Proponents of free speech argued that because source code conveys information to programmers, is written in a language, and can be used to share humour and other artistic pursuits, it is a protected form of communication.

One of the first court cases regarding the nature of source code as free speech involved University of California mathematics professor Dan Bernstein, who had published on the internet the source code for an encryption program that he created. At the time, encryption algorithms were classified as munitions by the United States government; exporting encryption to other countries was considered an issue of national security, and had to be approved by the State Department. The Electronic Frontier Foundation sued the U.S. government on Bernstein's behalf; the court ruled that source code was free speech, protected by the First Amendment.

Quality

The way a program is written can have important consequences for its maintainers. Coding conventions, which stress readability and some language-specific conventions, are aimed at the maintenance of the software source code, which involves debugging and updating. Other priorities, such as the speed of the programs execution, or the ability to compile the program for multiple architectures, often make code readability a less important consideration, since code *quality* depends entirely on its *purpose*.

Compiler



A diagram of the operation of a typical multi-language, multi-target compiler.

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code*). The most common reason for wanting to transform source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or

machine code). If the compiled program can only run on a computer whose CPU or operating system is different from the one on which the compiler runs the compiler is known as a cross-compiler. A program that translates from a low level language to a higher level one is a *decompiler*. A program that translates between high-level languages is usually called a *language translator*, *source to source translator*, or *language converter*. A *language rewriter* is usually a program that translates the form of expressions without a change of language.

A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization.

Program faults caused by incorrect compiler behavior can be very difficult to track down and work around and compiler implementors invest a lot of time ensuring the correctness of their software.

The term compiler-compiler is sometimes used to refer to a parser generator, a tool often used to help create the lexer and parser.

History

Software for early computers was primarily written in assembly language for many years. Higher level programming languages were not invented until the benefits of being able to reuse software on different kinds of CPUs started to become significantly greater than the cost of writing a compiler. The very limited memory capacity of early computers also created many technical problems when implementing a compiler.

Towards the end of the 1950s, machine-independent programming languages were first proposed. Subsequently, several experimental compilers were developed. The first compiler was written by Grace Hopper, in 1952, for the A-0 programming language. The FORTRAN team led by John Backus at IBM is generally credited as having introduced the first complete compiler in 1957. COBOL was an early language to be compiled on multiple architectures, in 1960.

In many application domains the idea of using a higher level language quickly caught on. Because of the expanding functionality supported by newer programming languages and the increasing complexity of computer architectures, compilers have become more and more complex.

Early compilers were written in assembly language. The first *self-hosting* compiler — capable of compiling its own source code in a high-level language — was created for Lisp by Tim Hart and Mike Levin at MIT in 1962. Since the 1970s it has become common practice to implement a compiler in the language it compiles, although both Pascal and C have been popular choices for implementation language. Building a self-hosting compiler is a bootstrapping problem—the first such compiler for a language must

be compiled either by a compiler written in a different language, or (as in Hart and Levin's Lisp compiler) compiled by running the compiler in an interpreter.

Compilers in education

Compiler construction and compiler optimization are taught at universities and schools as part of the computer science curriculum. Such courses are usually supplemented with the implementation of a compiler for an educational programming language. A well-documented example is Niklaus Wirth's PL/0 compiler, which Wirth used to teach compiler construction in the 1970s. In spite of its simplicity, the PL/0 compiler introduced several influential concepts to the field:

1. Program development by stepwise refinement (also the title of a 1971 paper by Wirth)
2. The use of a recursive descent parser
3. The use of EBNF to specify the syntax of a language
4. A code generator producing portable P-code
5. The use of T-diagrams in the formal description of the bootstrapping problem

Compilation

Compilers enabled the development of programs that are machine-independent. Before the development of FORTRAN (FORmula TRANslator), the first higher-level language, in the 1950s, machine-dependent assembly language was widely used. While assembly language produces more reusable and relocatable programs than machine code on the same architecture, it has to be modified or rewritten if the program is to be executed on different hardware architecture.

With the advance of high-level programming languages soon followed after FORTRAN, such as COBOL, C, BASIC, programmers can write machine-independent source programs. A compiler translates the high-level source programs into target programs in machine languages for the specific hardwares. Once the target program is generated, the user can execute the program.

Structure of compiler

Compilers bridge source programs in high-level languages with the underlying hardwares. A compiler requires 1) to recognize legitimacy of programs, 2) to generate correct and efficient code, 3) run-time organization, 4) to format output according to assembler or linker conventions. A compiler consists of three main parts: frontend, middle-end, and backend.

Frontend checks whether the program is correctly written in terms of the programming language syntax and semantics. Here legal and illegal programs are recognized. Errors are reported, if any, in a useful way. Type checking is also performed by collecting type information. Frontend generates IR (intermediate representation) for the middle-end.

Optimization of this part is almost complete so much are already automated. There are efficient algorithms typically in $O(n)$ or $O(n \log n)$.

Middle-end is where the optimizations for performance take place. Typical transformations for optimization are removal of useless or unreachable code, discovering and propagating constant values, relocation of computation to a less frequently executed place (e.g., out of a loop), or specializing a computation based on the context. Middle-end generates IR for the following backend. Most optimization efforts are focused on this part.

Backend is responsible for translation of IR into the target assembly code. The target instruction(s) are chosen for each IR instruction. Variables are also selected for the registers. Backend utilizes the hardware by figuring out how to keep parallel FUs busy, filling delay slots, and so on. Although most algorithms for optimization are in NP, heuristic techniques are well-developed.

Compiler output

One classification of compilers is by the platform on which their generated code executes. This is known as the *target platform.*

A *native* or *hosted* compiler is one whose output is intended to directly run on the same type of computer and operating system that the compiler itself runs on. The output of a cross compiler is designed to run on a different platform. Cross compilers are often used when developing software for embedded systems that are not intended to support a software development environment.

The output of a compiler that produces code for a virtual machine (VM) may or may not be executed on the same platform as the compiler that produced it. For this reason such compilers are not usually classified as native or cross compilers.

Compiled versus interpreted languages

Higher-level programming languages are generally divided for convenience into compiled languages and interpreted languages. However, in practice there is rarely anything about a language that *requires* it to be exclusively compiled or exclusively interpreted, although it is possible to design languages that rely on re-interpretation at run time. The categorization usually reflects the most popular or widespread implementations of a language — for instance, BASIC is sometimes called an interpreted language, and C a compiled one, despite the existence of BASIC compilers and C interpreters.

Modern trends toward just-in-time compilation and bytecode interpretation at times blur the traditional categorizations of compilers and interpreters.

Some language specifications spell out that implementations *must* include a compilation facility; for example, Common Lisp. However, there is nothing inherent in the definition

of Common Lisp that stops it from being interpreted. Other languages have features that are very easy to implement in an interpreter, but make writing a compiler much harder; for example, APL, SNOBOL4, and many scripting languages allow programs to construct arbitrary source code at runtime with regular string operations, and then execute that code by passing it to a special evaluation function. To implement these features in a compiled language, programs must usually be shipped with a runtime library that includes a version of the compiler itself.

Hardware compilation

The output of some compilers may target hardware at a very low level, for example a Field Programmable Gate Array (FPGA) or structured Application-specific integrated circuit (ASIC). Such compilers are said to be *hardware compilers* or synthesis tools because the source code they compile effectively control the final configuration of the hardware and how it operates; the output of the compilation are not instructions that are executed in sequence - only an interconnection of transistors or lookup tables. For example, XST is the Xilinx Synthesis Tool used for configuring FPGAs. Similar tools are available from Altera, Synplicity, Synopsys and other vendors.

Compiler design

In the early days, the approach taken to compiler design used to be directly affected by the complexity of the processing, the experience of the person(s) designing it, and the resources available.

A compiler for a relatively simple language written by one person might be a single, monolithic piece of software. When the source language is large and complex, and high quality output is required the design may be split into a number of relatively independent phases. Having separate phases means development can be parceled up into small parts and given to different people. It also becomes much easier to replace a single phase by an improved one, or to insert new phases later (e.g., additional optimizations).

The division of the compilation processes into phases was championed by the Production Quality Compiler-Compiler Project (PQCC) at Carnegie Mellon University. This project introduced the terms *front end*, *middle end*, and *back end*.

All but the smallest of compilers have more than two phases. However, these phases are usually regarded as being part of the front end or the back end. The point at which these two *ends* meet is open to debate. The front end is generally considered to be where syntactic and semantic processing takes place, along with translation to a lower level of representation (than source code).

The middle end is usually designed to perform optimizations on a form other than the source code or machine code. This source code/machine code independence is intended to enable generic optimizations to be shared between versions of the compiler supporting different languages and target processors.

The back end takes the output from the middle. It may perform more analysis, transformations and optimizations that are for a particular computer. Then, it generates code for a particular processor and OS.

This front-end/middle/back-end approach makes it possible to combine front ends for different languages with back ends for different CPUs. Practical examples of this approach are the GNU Compiler Collection, LLVM, and the Amsterdam Compiler Kit, which have multiple front-ends, shared analysis and multiple back-ends.

One-pass versus multi-pass compilers

Classifying compilers by number of passes has its background in the hardware resource limitations of computers. Compiling involves performing lots of work and early computers did not have enough memory to contain one program that did all of this work. So compilers were split up into smaller programs which each made a pass over the source (or some representation of it) performing some of the required analysis and translations.

The ability to compile in a single pass has classically been seen as a benefit because it simplifies the job of writing a compiler and one pass compilers generally compile faster than multi-pass compilers. Thus, partly driven by the resource limitations of early systems, many early languages were specifically designed so that they could be compiled in a single pass (e.g., Pascal).

In some cases the design of a language feature may require a compiler to perform more than one pass over the source. For instance, consider a declaration appearing on line 20 of the source which affects the translation of a statement appearing on line 10. In this case, the first pass needs to gather information about declarations appearing after statements that they affect, with the actual translation happening during a subsequent pass.

The disadvantage of compiling in a single pass is that it is not possible to perform many of the sophisticated optimizations needed to generate high quality code. It can be difficult to count exactly how many passes an optimizing compiler makes. For instance, different phases of optimization may analyse one expression many times but only analyse another expression once.

Splitting a compiler up into small programs is a technique used by researchers interested in producing provably correct compilers. Proving the correctness of a set of small programs often requires less effort than proving the correctness of a larger, single, equivalent program.

While the typical multi-pass compiler outputs machine code from its final pass, there are several other types:

- A "source-to-source compiler" is a type of compiler that takes a high level language as its input and outputs a high level language. For example, an automatic parallelizing compiler will frequently take in a high level language

program as an input and then transform the code and annotate it with parallel code annotations (e.g. OpenMP) or language constructs (e.g. Fortran's `DOALL` statements).

- Stage compiler that compiles to assembly language of a theoretical machine, like some Prolog implementations
    - This Prolog machine is also known as the Warren Abstract Machine (or WAM).
    - Bytecode compilers for Java, Python, and many more are also a subtype of this.
- Just-in-time compiler, used by Smalltalk and Java systems, and also by Microsoft .NET's Common Intermediate Language (CIL)
    - Applications are delivered in bytecode, which is compiled to native machine code just prior to execution.

Front end

The front end analyzes the source code to build an internal representation of the program, called the intermediate representation or *IR*. It also manages the symbol table, a data structure mapping each symbol in the source code to associated information such as location, type and scope. This is done over several phases, which includes some of the following:

1. **Line reconstruction**. Languages which strop their keywords or allow arbitrary spaces within identifiers require a phase before parsing, which converts the input character sequence to a canonical form ready for the parser. The top-down, recursive-descent, table-driven parsers used in the 1960s typically read the source one character at a time and did not require a separate tokenizing phase. Atlas Autocode, and Imp (and some implementations of Algol and Coral66) are examples of stropped languages whose compilers would have a *Line Reconstruction* phase.
2. Lexical analysis breaks the source code text into small pieces called *tokens*. Each token is a single atomic unit of the language, for instance a keyword, identifier or symbol name. The token syntax is typically a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it. This phase is also called lexing or scanning, and the software doing lexical analysis is called a lexical analyzer or scanner.
3. Preprocessing. Some languages, e.g., C, require a preprocessing phase which supports macro substitution and conditional compilation. Typically the preprocessing phase occurs before syntactic or semantic analysis; e.g. in the case of C, the preprocessor manipulates lexical tokens rather than syntactic forms. However, some languages such as Scheme support macro substitutions based on syntactic forms.
4. Syntax analysis involves parsing the token sequence to identify the syntactic structure of the program. This phase typically builds a parse tree, which replaces the linear sequence of tokens with a tree structure built according to the rules of a

formal grammar which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.

5. Semantic analysis is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

Back end

The term *back end* is sometimes confused with *code generator* because of the overlapped functionality of generating assembly code. Some literature uses *middle end* to distinguish the generic analysis and optimization phases in the back end from the machine-dependent code generators.

The main phases of the back end include the following:

1. Analysis: This is the gathering of program information from the intermediate representation derived from the input. Typical analyses are data flow analysis to build use-define chains, dependence analysis, alias analysis, pointer analysis, escape analysis etc. Accurate analysis is the basis for any compiler optimization. The call graph and control flow graph are usually also built during the analysis phase.
2. Optimization: the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are inline expansion, dead code elimination, constant propagation, loop transformation, register allocation and even automatic parallelization.
3. Code generation: the transformed intermediate language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated addressing modes.

Compiler analysis is the prerequisite for any compiler optimization, and they tightly work together. For example, dependence analysis is crucial for loop transformation.

In addition, the scope of compiler analysis and optimizations vary greatly, from as small as a basic block to the procedure/function level, or even over the whole program (interprocedural optimization). Obviously, a compiler can potentially do a better job using a broader view. But that broad view is not free: large scope analysis and optimizations are very costly in terms of compilation time and memory space; this is especially true for interprocedural analysis and optimizations.

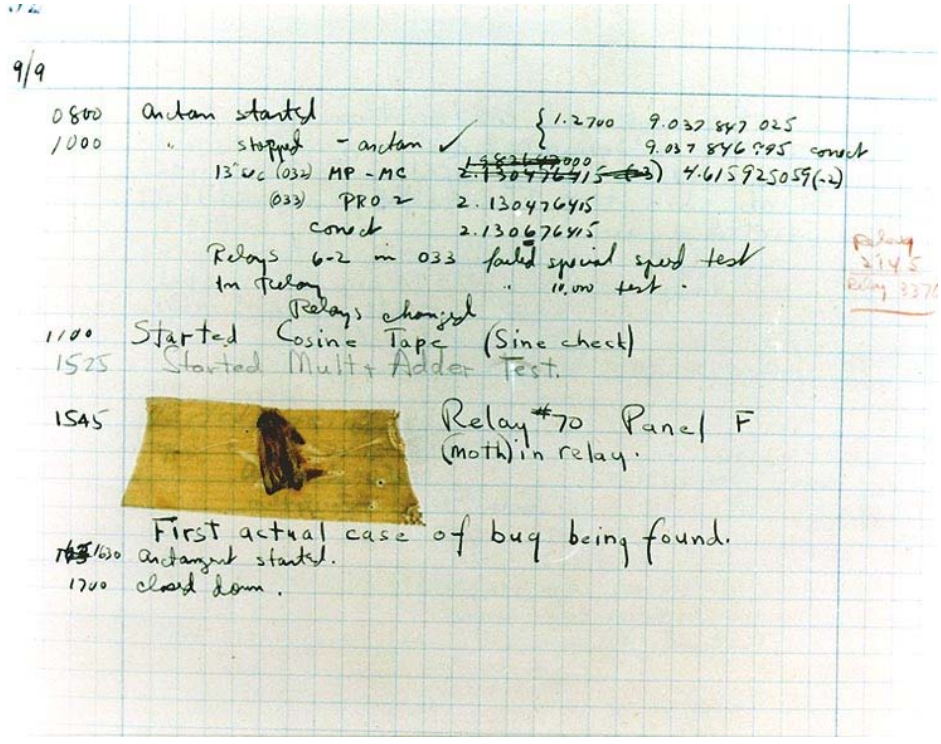
Interprocedural analysis and optimizations are common in modern commercial compilers from HP, IBM, SGI, Intel, Microsoft, and Sun Microsystems. The open source GCC was criticized for a long time for lacking powerful interprocedural optimizations, but it is changing in this respect. Another open source compiler with full analysis and optimization infrastructure is Open64, which is used by many organizations for research and commercial purposes.

Due to the extra time and space needed for compiler analysis and optimizations, some compilers skip them by default. Users have to use compilation options to explicitly tell the compiler which optimizations should be enabled.

Compiler correctness

Compiler correctness is the branch of software engineering that deals with trying to show that a compiler behaves according to its language specification. Techniques include developing the compiler using formal methods and using rigorous testing (often called compiler validation) on an existing compiler.

Debugging



A photo of the apocryphally "first" real bug, which was debugged in 1947

Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware, thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another. Many books have been written about debugging, as it involves numerous aspects, including: interactive debugging, control flow, integration testing, log files, monitoring, memory dumps, Statistical Process Control, and special design tactics to improve detection while simplifying changes.

Origin

There is some controversy over the origin of the term "debugging."

The terms "bug" and "debugging" are both popularly attributed to Admiral Grace Hopper in the 1940s. While she was working on a Mark II Computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. However the term "bug" in the meaning of technical error dates back at least to 1878 and Thomas Edison, and "debugging" seems to have been used as a term in aeronautics before entering the world of computers. Indeed, in an interview Grace Hopper remarked that she was not coining the term. The moth fit the already existing terminology, so she saved it.

The Oxford English Dictionary entry for "debug" quotes the term "debugging" used in reference to airplane engine testing in a 1945 article in the Journal of the Royal Aeronautical Society, Hopper's bug was found on the 9th of September in 1947. The term was not adopted by computer programmers until the early 1950s. The seminal article by Gill in 1951 is the earliest in-depth discussion of programming errors, but it does not use the term "bug" or "debugging". In the ACM's digital library, the term "debugging" is first used in three papers from 1952 ACM National Meetings. Two of the three use the term in quotation marks. By 1963, "debugging" was a common enough term to be mentioned in passing without explanation on page 1 of the CTSS manual.

Kidwell's article *Stalking the Elusive Computer Bug* discusses the etymology of "bug" and "debug" in greater detail.

The scope of debugging

As software and electronic systems have become generally more complex, the various common debugging techniques have expanded with more methods to detect anomalies, assess impact, and schedule software patches or full updates to a system. The words "anomaly" and "discrepancy" can be used, as being more neutral terms, to avoid the words "error" and "defect" or "bug" where there might be an implication that all so-called *errors*, *defects* or *bugs* must be fixed (at all costs). Instead, an impact assessment can be made to determine if changes to remove an *anomaly* (or *discrepancy*) would be cost-effective for the system, or perhaps a scheduled new release might render the change(s) unnecessary. Not all issues are life-critical or mission-critical in a system. Also, it is

important to avoid the situation where a change might be more upsetting to users, long-term, than living with the known problem(s) (where the "cure would be worse than the disease"). Basing decisions of the acceptability of some anomalies can avoid a culture of a "zero-defects" mandate, where people might be tempted to deny the existence of problems so that the result would appear as zero *defects*. Considering the collateral issues, such as the cost-versus-benefit impact assessment, then broader debugging techniques will expand to determine the frequency of anomalies (how often the same "bugs" occur) to help assess their impact to the overall system.

Tools

Debugging ranges, in complexity, from fixing simple errors to performing lengthy and tiresome tasks of data collection, analysis, and scheduling updates. The debugging skill of the programmer can be a major factor in the ability to debug a problem, but the difficulty of software debugging varies greatly with the complexity of the system, and also depends, to some extent, on the programming language(s) used and the available tools, such as *debuggers*. Debuggers are software tools which enable the programmer to monitor the execution of a program, stop it, re-start it, set breakpoints, change values in memory and even, in some cases, go back in time. The term *debugger* can also refer to the person who is doing the debugging.

Generally, high-level programming languages, such as Java, make debugging easier, because they have features such as exception handling that make real sources of erratic behaviour easier to spot. In lower-level programming languages such as C or assembly, bugs may cause silent problems such as memory corruption, and it is often difficult to see where the initial problem happened. In those cases, memory debugger tools may be needed.

In certain situations, general purpose software tools that are language specific in nature can be very useful. These take the form of *static code analysis tools*. These tools look for a very specific set of known problems, some common and some rare, within the source code. All such issues detected by these tools would rarely be picked up by a compiler or interpreter, thus they are not syntax checkers, but more semantic checkers. Some tools claim to be able to detect 300+ unique problems. Both commercial and free tools exist in various languages. These tools can be extremely useful when checking very large source trees, where it is impractical to do code walkthroughs. A typical example of a problem detected would be a variable dereference that occurs *before* the variable is assigned a value. Another example would be to perform strong type checking when the language does not require such. Thus, they are better at locating likely errors, versus actual errors. As a result, these tools have a reputation of false positives. The old Unix *lint* program is an early example.

For debugging electronic hardware (e.g., computer hardware) as well as low-level software (e.g., BIOSes, device drivers) and firmware, instruments such as oscilloscopes, logic analyzers or in-circuit emulators (ICEs) are often used, alone or in combination. An

ICE may perform many of the typical software debugger's tasks on low-level software and firmware.

Typical debugging process

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.

After the bug is reproduced, the input of the program may need to be simplified to make it easier to debug. For example, a bug in a compiler can make it crash when parsing some large source file. However, after simplification of the test case, only few lines from the original source file can be sufficient to reproduce the same crash. Such simplification can be made manually, using a divide-and-conquer approach. The programmer will try to remove some parts of 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.

After the test case is sufficiently simplified, a programmer can use a debugger tool to examine program states (values of variables, plus the call stack) and track down the origin of the problem(s). Alternatively, tracing can be used. In simple cases, tracing is just a few print statements, which output the values of variables at certain points of program execution.

Various debugging techniques

- Print (or tracing) debugging is the act of watching (live or recorded) trace statements, or print statements, that indicate the flow of execution of a process.

- Remote debugging is the process of debugging a program running on a system different than the debugger. To start remote debugging, debugger connects to a remote system over a network. Once connected, debugger can control the execution of the program on the remote system and retrieve information about its state.

- Post-mortem debugging is debugging of the program after it has already crashed. Related techniques often include various tracing techniques (for example,) and/or analysis of memory dump (or core dump) of the crashed process. The dump of the process could be obtained automatically by the system (for example, when process has terminated due to an unhandled exception), or by a programmer-inserted instruction, or manually by the interactive user.

- *Delta Debugging* - technique of automating test case simplification. [:p.123]

- *Saff Squeeze* - technique of isolating failure within the test using progressive inlining of parts of the failing test .

Debugging for embedded systems

In contrast to the general purpose computer software design environment, a primary characteristic of embedded environments is the sheer number of different platforms available to the developers (CPU architectures, vendors, operating systems and their variants). Embedded systems are, by definition, not general-purpose designs: they are typically developed for a single task (or small range of tasks), and the platform is chosen specifically to optimize that application. Not only does this fact make life tough for embedded system developers, it also makes debugging and testing of these systems harder as well, since different debugging tools are needed in different platforms. Put simply, embedded systems debuggers have two key requirements.

- to identify and fix bugs in the system (e.g. logical or synchronization problems in the code, or a design error in the hardware);
- to collect information about the operating states of the system that may then be used to analyze the system: to find ways to boost its performance or to optimize other important characteristics (e.g. energy consumption, reliability, real-time response etc.).

Anti-debugging

Anti-debugging is "the implementation of one or more techniques within computer code that hinders attempts at reverse engineering or debugging a target process". It is actively used in legitimate copy-protection schemas, but is also used by malware to complicate its detection and elimination . Techniques used in anti-debugging include:

- API-based: check for the existence of a debugger using system information
- Exception-based: check to see if exceptions are interfered with
- Process and thread blocks: check whether process and thread blocks have been manipulated
- Modified code: check for code modifications made by a debugger handling software breakpoints
- Hardware- and register-based: check for hardware breakpoints and CPU registers
- Timing and latency: check the time taken for the execution of instructions
- Detecting and penalizing debugger

Chapter 4

Algorithmic Skeleton and Control Table

Algorithmic skeleton

In computing, **algorithmic skeletons** (a.k.a. Parallelism Patterns) are a high-level parallel programming model for parallel and distributed computing.

Algorithmic skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by combining the basic ones.

Overview

The most outstanding feature of algorithmic skeletons, which differentiates them from other high-level parallel programming models, is that orchestration and synchronization of the parallel activities is implicitly defined by the skeleton patterns. Programmers do not have to specify the synchronizations between the application's sequential parts. This yields two implications. First, as the communication/data access patterns are known in advance, cost models can be applied to schedule skeletons programs. Second, that algorithmic skeleton programming reduces the number of errors when compared to traditional lower-level parallel programming models (Threads, MPI).

History

Algorithmic skeletons were first introduced by Cole in 1989. Several frameworks have been proposed by different research groups using different techniques such as functional, imperative, custom and object oriented languages.

Well-known skeleton patterns

This section describes some well known Algorithmic Skeleton patterns. Additionally, the patterns signature in the Skandium library is provided for clarity.

- **FARM** is also known as **master-slave**. Farm represents task replication where the execution of different tasks by the same farm are replicated and executed in parallel.

```
Farm(Skeleton<P,R> skeleton){...}
```

- **PIPE** represents staged computation. Different tasks can be computed simultaneously on different pipe stages. A pipe can have a variable number of stages, each stage of a pipe may be a nested skeleton pattern. Note that an n-stage pipe can be composed by nesting n-1 2-stage pipes.

```
<X> Pipe(Skeleton<P,X> stage1, Skeleton<P,X> stage2){...}
```

- **FOR** represents fixed iteration, where a task is executed a fixed number of times. In some implementations the executions may take place in parallel.

```
For(Skeleton<P,X> skeleton, int times){...}
```

- **WHILE** represents conditional iteration, where a given skeleton is executed until a condition is met.

```
public While(Skeleton<P,P> skeleton, Condition<P> condition){...}
```

- **IF** represents conditional branching, where the execution choice between two skeleton patterns is decided by a condition.

```
If(Condition<P> condition, Skeleton<P,R> trueCase, Skeleton<P,R>
falseCase){...}
```

- **MAP** represents *split*, *execute*, *merge* computation. A task is divided into sub-tasks, sub-tasks are executed in parallel according to a given skeleton, and finally sub-task's results are merged to produce the original task's result.

```
<X,Y> Map(Split<P,X> split, Skeleton<X,Y> skeleton, Merge<Y,R>
merge){...}
```

- **D&C** represents divide and conquer parallelism. A task is recursively sub-divided until a condition is met, then the sub-task is executed and results are merged while the recursion is unwound.

```
DaC(Condition<P> condition, Split<P,P> split, Skeleton<P,R> skeleton,
Merge<R,R> merge){...}
```

- **SEQ** does not represent parallelism, but it is often used a convenient tool to wrap code as the leafs of the skeleton nesting.

```
public Seq(Execute <P,R> execute){...}
```

Example program

The following example is based on the Java Skandium library for parallel programming.

The objective is to implement an Algorithmic Skeleton based parallel version of the **QuickSort** algorithm using the Divide and Conquer pattern. Notice that the high-level approach hides Thread management from the programmer.

```
// 1. Define the skeleton program
Skeleton<Range, Range> sort =   new DaC<Range, Range>(
    new ShouldSplit(threshold, maxTimes),
    new SplitList(),
    new Sort(),
    new MergeList());

// 2. Input parameters
Future<Range> future = sort.input(new Range(generate(...)));

// 3. Do something else here.
// ...

// 4. Block for the results
Range result = future.get();
```

1. The first thing is to define a new instance of the skeleton with the functional code that fills the pattern (ShouldSplit, SplitList, Sort, MergeList). The functional code is written by the programmer without parallelism concerns.
2. The second step is the input of data which triggers the computation. In this case Range is a class holding an array and two indexes which allow the representation of a subarray. For every data entered into the framework a new Future object is created. More than one Future can be entered into a skeleton simultaneously.
3. The Future allows for asynchronous computation, as other tasks can be performed while the results are computed.
4. We can retrieve the result of the computation, blocking if necessary (i.e. results not yet available).

The functional codes in this example correspond to four types Condition, Split, Execute, and Merge.

```
public class ShouldSplit implements Condition<Range>{

  int threshold, maxTimes, times;

  public ShouldSplit(int threshold, int maxTimes){
    this.threshold = threshold;
    this.maxTimes  = maxTimes;
    this.times     = 0;
  }

  @Override
  public synchronized boolean condition(Range r){
```

```
    return r.right - r.left > threshold &&
           times++ < this.maxTimes;
  }
}
```

The ShouldSplit class implements the Condition interface. The function receives an input, Range r in this case, and returning true or false. In the context of the Divide and Conquer where this function will be used, this will decide whether a sub-array should be subdivided again or not.

The SplitList class implements the split interface, which in this case divides an (sub-)array into smaller sub-arrays. The class uses a helper function `partition(...)` which implements the well known QuickSort pivot and swap scheme.

```
public class SplitList implements Split<Range, Range>{

 @Override
 public Range[] split(Range r){

    int i = partition(r.array, r.left, r.right);

    Range[] intervals = {new Range(r.array, r.left, i-1),
                         new Range(r.array, i+1, r.right)};

    return intervals;
 }
}
```

The Sort class implements and Execute interface, and is in charge of sorting the sub-array specified by `Range r`. In this case we simply invoke Java's default (Arrays.sort) method for the given sub-array.

```
public class Sort implements Execute<Range, Range> {

  @Override
  public Range execute(Range r){

    if (r.right <= r.left) return r;

    Arrays.sort(r.array, r.left, r.right+1);

    return r;
  }
}
```

Finally, once a set of sub-arrays are sorted we merge the sub-array parts into a bigger array with the MergeList class which implements a Merge interface.

```
public class MergeList implements Merge<Range, Range>{

  @Override
  public Range merge(Range[] r){
```

```
    Range result = new Range( r[0].array, r[0].left, r.right);

    return result;
  }
}
```

Frameworks and libraries

ASSIST

ASSIST is a programming environment which provides programmers with a structured coordination language. The coordination language can express parallel programs as an arbitrary graph of software modules. The module graph describes how a set of modules interact with each other using a set of typed data streams. The modules can be sequential or parallel. Sequential modules can be written in C, C++, or Fortran; and parallel modules are programmed with a special ASSIST parallel module (*parmod*).

AdHoc , a hierarchical and fault-tolerant Distributed Shared Memory (DSM) system is used to interconnect streams of data between processing elements by providing a repository with: get/put/remove/execute operations. Research around AdHoc has focused on transparency, scalability, and fault-tolerance of the data repository.

While not a classical skeleton framework, in the sense that no skeletons are provided, ASSIST's generic *parmod* can be specialized into classical skeletons such as: *farm*, *map*, etc. ASSIST also supports autonomic control of *parmod*s, and can be subject to a performance contract by dynamically adapting the number of resources used.

CO2P3S

CO2P3S (Correct Object-Oriented Pattern-based Parallel Programming System), is a pattern oriented development environment , which achieves parallelism using threads in Java.

CO2P3S is concerned with the complete development process of a parallel application. Programmers interact through a programming GUI to choose a pattern and its configuration options. Then, programmers fill the hooks required for the pattern, and new code is generated as a framework in Java for the parallel execution of the application. The generated framework uses three levels, in descending order of abstraction: patterns layer, intermediate code layer, and native code layer. Thus, advanced programmers may intervene the generated code at multiple levels to tune the performance of their applications. The generated code is mostly type safe, using the types provided by the programmer which do not require extension of superclass, but fails to be completely type safe such as in the reduce(..., Object reducer) method in the mesh pattern.

The set of patterns supported in CO2P3S corresponds to method-sequence, distributor, mesh, and wavefront. Complex applications can be built by composing frameworks with

their object references. Nevertheless, if no pattern is suitable, the MetaCO2P3S graphical tool addresses extensibility by allowing programmers to modify the pattern designs and introduce new patterns into CO2P3S.

Support for distributed memory architectures in CO2P3S was introduced in later . To use a distributed memory pattern, programmers must change the pattern's memory option from shared to distributed, and generate the new code. From the usage perspective, the distributed memory version of the code requires the management of remote exceptions.

Calcium & Skandium

Calcium is greatly inspired by Lithium and Muskel. As such, it provides algorithmic skeleton programming as a Java library. Both task and data parallel skeletons are fully nestable; and are instantiated via parametric skeleton objects, not inheritance.

Calcium supports the execution of skeleton applications on top of the ProActive environment for distributed cluster like infrastructure. Additionally, Calcium has three distinctive features for algorithmic skeleton programming. First, a performance tuning model which helps programmers identify code responsible for performance bugs. Second, a type system for nestable skeletons which is proven to guaranty subject reduction properties and is implemented using Java Generics. Third, a transparent algorithmic skeleton file access model, which enables skeletons for data intensive applications.

Skandium is a complete re-implementation of **Calcium** for multi-core computing. Programs written on **Skandium** may take advantage of shared memory to simplify parallel programming .

Eden

Eden is a parallel programming language for distributed memory environments, which extends Haskell. Processes are defined explicitly to achieve parallel programming, while their communications remain implicit. Processes communicate through unidirectional channels, which connect one writer to exactly one reader. Programmers only need to specify which data a processes depends on. Eden's process model provides direct control over process granularity, data distribution and communication topology.

Eden is not a skeleton language in the sense that skeletons are not provided as language constructs. Instead, skeletons are defined on top of Eden's lower-level process abstraction, supporting both task and data parallelism. So, contrary to most other approaches, Eden lets the skeletons be defined in the same language and at the same level, the skeleton instantiation is written: Eden itself. Because Eden is an extension of a functional language, Eden skeletons are higher order functions. Eden introduces the concept of implementation skeleton, which is an architecture independent scheme that describes a parallel implementation of an algorithmic skeleton.

eSkel

The **Edinburgh Skeleton Library** (**eSkel**) is provided in C and runs on top of MPI. The first version of eSkel was described in , while a later version is presented in .

In , nesting-mode and interaction-mode for skeletons are defined. The nesting-mode can be either transient or persistent, while the interaction-mode can be either implicit or explicit. Transient nesting means that the nested skeleton is instantiated for each invocation and destroyed Afterwards, while persistent means that the skeleton is instantiated once and the same skeleton instance will be invoked throughout the application. Implicit interaction means that the flow of data between skeletons is completely defined by the skeleton composition, while explicit means that data can be generated or removed from the flow in a way not specified by the skeleton composition. For example, a skeleton that produces an output without ever receiving an input has explicit interaction.

Performance prediction for scheduling and resource mapping, mainly for pipe-lines, has been explored by Benoit et al. They provided a performance model for each mapping, based on process algebra, and determine the best scheduling strategy based on the results of the model.

More recent works have addressed the problem of adaptation on structured parallel programming , in particular for the pipe skeleton.

HDC

Higher-order Divide and Conquer (**HDC**) is a subset of the functional language Haskell . Functional programs are presented as polymorphic higher-order functions, which can be compiled into C/MPI, and linked with skeleton implementations. The language focus on divide and conquer paradigm, and starting from a general kind of divide and conquer skeleton, more specific cases with efficient implementations are derived. The specific cases correspond to: fixed recursion depth, constant recursion degree, multiple block recursion, elementwise operations, and correspondent communications

HDC pays special attention to the subproblem's granularity and its relation with the number of Available processors. The total number of processors is a key parameter for the performance of the skeleton program as HDC strives to estimate an adequate assignment of processors for each part of the program. Thus, the performance of the application is strongly related with the estimated number of processors leading to either exceeding number of subproblems, or not enough parallelism to exploit available processors.

HOC-SA

HOC-SA is an Globus Incubator project.
HOC-SA stands for Higher-Order Components-Service Architecture. Higher-Order Components (HOCs) have the aim of simplifying Grid application development.
The objective of HOC-SA is to provide Globus users, who do not want to know about all the details of the Globus middleware (GRAM RSL documents, Web services and resource configuration etc.), with HOCs that provide a higher-level interface to the Grid than the core Globus Toolkit.
HOCs are Grid-enabled skeletons, implemented as components on top of the Globus Toolkit, remotely accessibly via Web Services.

JaSkel

JaSkel is a Java based skeleton framework providing skeletons such as farm, pipe and heartbeat. Skeletons are specialized using inheritance. Programmers implement the abstract methods for each skeleton to provide their application specific code. Skeletons in JaSkel are provided in both sequential, concurrent and dynamic versions. For example, the concurrent farm can be used in shared memory environments (threads), but not in distributed environments (clusters) where the distributed farm should be used. To change from one version to the other, programmers must change their classes' signature to inherit from a different skeleton. The nesting of skeletons uses the basic Java Object class, and therefore no type system is enforced during the skeleton composition.

The distribution aspects of the computation are handled in **JaSkel** using AOP, more specifically the AspectJ implementation. Thus, **JaSkel** can be deployed on both cluster and Grid like infrastructures . Nevertheless, a drawback of the **JaSkel** approach is that the nesting of the skeleton strictly relates to the deployment infrastructure. Thus, a double nesting of farm yields a better performance than a single farm on hierarchical infrastructures. This defeats the purpose of using AOP to separate the distribution and functional concerns of the skeleton program.

Lithium & Muskel

Lithium and its successor **Muskel** are skeleton frameworks developed at University of Pisa, Italy. Both of them provide nestable skeletons to the programmer as Java libraries.The evaluation of a skeleton application follows a formal definition of operational semantics introduced by Aldinucci and Danelutto , which can handle both task and data parallelism. The semantics describe both functional and parallel behavior of the skeleton language using a labeled transition system. Additionally, several performance optimization are applied such as: skeleton rewriting techniques [18, 10], task lookahead, and server-to-server lazy binding.

At the implementation level, Lithium exploits macro-data flow to achieve parallelism. When the input stream receives a new parameter, the skeleton program is processed to obtain a macro-data flow graph. The nodes of the graph are macro-data flow instructions

(MDFi) which represent the sequential pieces of code provided by the programmer. Tasks are used to group together several MDFi, and are consumed by idle processing elements from a task pool. When the computation of the graph is concluded, the result is placed into the output stream and thus delivered back to the user.

Muskel also provides non-functional features such as Quality of Service (QoS); security between task pool and interpreters ; and resource discovery, load balancing, and fault tolerance when interfaced with Java / Jini Parallel Framework (JJPF) , a distributed execution framework. **Muskel** also provides support for combining structured with unstructured programming and recent research has addressed extensibility .

Mallba

Mallba is a library for combinatorial optimizations supporting exact, heuristic and hybrid search strategies. Each strategy is implemented in Mallba as a generic skeleton which can be used by prodiving the required code. On the exact search algorithms Mallba provides branch-and-bound and dynamic-optimization skeletons. For local search heuristics Mallba supports: hill climbing, metropolis, simulated annealing, and tabu search; and also population based heuristics derived from evolutionary algorithms such as genetic algorithms, evolution strategy, and others (CHC). The hybrid skeletons combine strategies, such as: GASA, a mixture of genetic algorithm and simulated annealing, and CHCCES which combines CHC and ES.

The skeletons are provided as a C++ library and are not nestable but type safe. A custom MPI abstraction layer is used, NetStream, which takes care of primitive data type marshalling, synchronization, etc. A skeleton may have multiple lower-level parallel implementations depending on the target architectures: sequential, LAN, and WAN. For example: centralized master-slave, distributed master-slave, etc.

Mallba also provides state variables which hold the state of the search skeleton. The state links the search with the environment, and can be accessed to inspect the evolution of the search and decide on future actions. For example, the state can be used to store the best solution found so far, or α, β values for branch and bound pruning .

Compared with other frameworks, Mallba's usage of skeletons concepts is unique. Skeletons are provided as parametric search strategies rather than parametric parallelization patterns.

Muesli

The Muenster Skeleton Library **Muesli** is a C++ template library which re-implements many of the ideas and concepts introduced in Skil, e.g. higher order functions, currying, and polymorphic types . It is build on top of MPI 1.2 and OpenMP 2.5 and supports, unlike many other skeleton libraries, both task and data parallel skeletons. Skeleton nesting (composition) is similar to the two tier approach of P3L, i.e. task parallel skeletons can be nested arbitrarily while data parallel skeletons cannot, but may be used

at the leaves of a task parallel nesting tree . C++ templates are used to render skeletons polymorphic, but no type system is enforced. However, the library implements an automated serialization mechanism inspired by such that, in addition to the standard MPI data types, arbitrary user-defined data types can be used within the skeletons. The supported task parallel skeletons are Branch & Bound , Divide & Conquer , Farm , and Pipe, auxiliary skeletons are Filter, Final, and Initial. Data parallel skeletons, such as fold (reduce), map, permute, zip, and their variants are implemented as higher order member functions of a distributed data structure. Currently, Muesli supports distributed data structures for arrays, matrices, and sparse matrices .

As a unique feature, Muesli's data parallel skeletons automatically scale both on single- as well as on multi-core, multi-node cluster architectures . Here, scalability across nodes and cores is ensured by simultaneously using MPI and OpenMP, respectively. However, this feature is optional in the sense that a program written with Muesli still compiles and runs on a single-core, multi-node cluster computer without changes to the source code, i.e. backward compatibility is guaranteed. This is ensured by providing a very thin OpenMP abstraction layer such that the support of multi-core architectures can be switched on/off by simply providing/omitting the OpenMP compiler flag when compiling the program. By doing so, virtually no overhead is introduced at runtime.

P3L, SkIE, SKElib

P3L (Pisa Parallel Programming Language) is a skeleton based coordination language. **P3L** provides skeleton constructs which are used to coordinate the parallel or sequential execution of C code. A compiler named Anacleto is provided for the language. Anacleto uses implementation templates to compile P3 L code into a target architecture. Thus, a skeleton can have several templates each optimized for a different architecture. A template implements a skeleton on a specific architecture and provides a parametric process graph with a performance model. The performance model can then be used to decide program transformations which can lead to performance optimizations .

A **P3L** module corresponds to a properly defined skeleton construct with input and output streams, and other sub-modules or sequential C code. Modules can be nested using the two tier model, where the outer level is composed of task parallel skeletons, while data parallel skeletons may be used in the inner level. Type verification is performed at the data flow level, when the programmer explicitly specifies the type of the input and output streams, and by specifying the flow of data between sub-modules.

SkIE (Skeleton-based Integrated Environment) is quite similar to **P3L**, as it is also based on a coordination language, but provides advanced features such as debugging tools, performance analysis, visualization and graphical user interface. Instead of directly using the coordination language, programmers interact with a graphical tool, where parallel modules based on skeletons can be composed.

SKELib builds upon the contributions of **P3L** and **SkIE** by inheriting, among others, the template system. It differs from them because a coordination language is no longer used,

but instead skeletons are provided as a library in C, with performance similar as the one achieved in **P3L**. Contrary to **Skil**, another C like skeleton framework, type safety is not addressed in **SKELib**.

PAS and EPAS

PAS (Parallel Architectural Skeletons) is a framework for skeleton programming developed in C++ and MPI . Programmers use an extension of C++ to write their skeleton applications1 . The code is then passed through a Perl script which expands the code to pure C++ where skeletons are specialized through inheritance.

In **PAS**, every skeleton has a Representative (Rep) object which must be provided by the programmer and is in charge of coordinating the skeleton's execution. Skeletons can be nested in a hierarchical fashion via the Rep objects. Besides the skeleton's execution, the Rep also explicitly manages the reception of data from the higher level skeleton, and the sending of data to the sub-skeletons. A parametrized communication/synchronization protocol is used to send and receive data between parent and sub-skeletons.

An extension of PAS labeled as **SuperPas** and later as **EPAS** addresses skeleton extensibility concerns. With the **EPAS** tool, new skeletons can be added to **PAS**. A Skeleton Description Language (SDL) is used to describe the skeleton pattern by specifying the topology with respect to a virtual processor grid. The SDL can then be compiled into native C++ code, which can be used as any other skeleton.

SBASCO

SBASCO (**Skeleton-BAsed Scientific COmponents**) is a programming environment oriented towards efficient development of parallel and distributed numerical applications . **SBASCO** aims at integrating two programming models: skeletons and components with a custom composition language. An application view of a component provides a description of its interfaces (input and output type); while a configuration view provides, in addition, a description of the component's internal structure and processor layout. A component's internal structure can be defined using three skeletons: farm, pipe and multi-block.

SBASCO's addresses domain decomposable applications through its multi-block skeleton. Domains are specified through arrays (mainly two dimensional), which are decomposed into sub-arrays with possible overlapping boundaries. The computation then takes place in an iterative BSP like fashion. The first stage consists of local computations, while the second stage performs boundary exchanges. A use case is presented for a reaction-diffusion problem in .

Two type of components are presented in . Scientific Components (SC) which provide the functional code; and Communication Aspect Components (CAC) which encapsulate non-functional behavior such as communication, distribution processor layout and replication. For example, SC components are connected to a CAC component which can

act as a manager at runtime by dynamically re-mapping processors assigned to a SC. A use case showing improved performance when using CAC components is shown in.

SCL

The **Structured Coordination Language** (**SCL**) was one of the first languages introduced for skeleton programming . SCL is considered a base language, and was designed to be integrated with a host language, for example Fortran. In **SCL**, skeletons are classified into three types: **configuration**, **elementary** and **computation**. Configuration skeletons abstract patterns for commonly used data structures such as distributed arrays (ParArray). Elementary skeletons correspond to data parallel skeletons such as map, scan, and fold. Computation skeletons which abstract the control flow and correspond mainly to task parallel skeletons such as farm, SPMD, and iterateUntil.

SKiPPER & QUAFF

SKiPPER is a domain specific skeleton library for vision applications which provides skeletons in CAML, and thus relies on CAML for type safety. Skeletons are presented in two ways: declarative and operational. Declarative skeletons are directly used by programmers, while their operational versions provide an architecture specific target implementation. From the runtime environment, CAML skeleton specifications, and application specific functions (provided in C by the programmer), new C code is generated and compiled to run the application on the target architecture. One of the interesting things about **SKiPPER** is that the skeleton program can be executed sequentially for debugging.

Different approaches have been explored in **SKiPPER** for writing operational skeletons: static data-flow graphs, parametric process networks, hierarchical task graphs, and tagged-token data- flow graphs.

QUAFF is a more recent skeleton library written in C++ and MPI. QUAFF relies on template-based meta-programming techniques to reduce runtime overheads and perform skeleton expansions and optimizations at compilation time. Skeletons can be nested and sequential functions are stateful. Besides type checking, QUAFF takes advantage of C++ templates to generate, at compilation time, new C/MPI code. QUAFF is based on the CSP-model, where the skeleton program is described as a process network and production rules (single, serial, par, join).

SkeTo

The **SkeTo** project is a C++ library which achieves parallelization using MPI. SkeTo is different to other skeleton libraries because instead of providing nestable parallelism patterns, SkeTo provides parallel skeletons for parallel data structures such as: lists, trees , and matrices . The data structures are typed using templates, and several parallel operations can be invoked on them. For example the list structure provides parallel operations such as: map, reduce, scan, zip, shift, etc...

Additional research around SkeTo has also focused on optimizations strategies by transforma- tion, and more recently domain specific optimizations . For example, **SkeTo** provides a fusion transformation which merges two successive function invocations into a single one, thus decreasing the function call overheads and avoiding the creation of intermediate data structures passed between functions.

Skil

Skil is an imperative language for skeleton programming. Skeletons are not directly part of the language but are implemented with it. **Skil** uses a subset of C language which provides functional language like features such as higher order functions, curring and polymorphic types. When **Skil** is compiled, such features are eliminated and a regular C code is produced. Thus, **Skil** transforms polymorphic high order functions into monomorphic first order C functions. **Skil** does not support nestable composition of skeletons. Data parallelism is achieved using specific data parallel structures, for example to spread arrays among available processors. Filter skeletons can be used.

Frameworks comparison

- **Activity Years** is the known activity years span. The dates represented in this column correspond to the first and last publication date of a related article in a scientific journal or conference proceeding. Note that a project may still be active beyond the activity span, and that we have failed to find a publication for it beyond the given date.
- **Programming Language** is the interface with which programmers interact to code their skeleton applications. These languages are diverse, encompassing paradigms such as: functional languages, coordination languages, markup languages, imperative languages, object oriented languages, and even graphical user interfaces. Inside the programming language, skeletons have been provided either as language constructs or libraries. Providing skeletons as language construct implies the development of a custom domain specific language and its compiler. This was clearly the stronger trend at the begging of skeleton research. The more recent trend is to provide skeletons as libraries, in particular with object oriented languages such as C++ and Java.

- **Execution Language** is the language in which the skeleton applications are run or compiled. It was recognized very early that the programming languages (specially in the functional cases), were not efficient enough to execute the skeleton programs. Therefore, skeleton programming languages were simplified by executing skeleton application on other languages. Transformation processes were introduced to convert the skeleton applications (defined in the programming language) into an equivalent application on the target execution language. Different transformation processes were introduced, such as code generation or instantiation of lowerlevel skeletons (sometimes called operational skeletons) which were capable of interacting with a library in the execution language. The transformed application also gave the opportunity to introduce target architecture

code, customized for performance, into the transformed application. Table 1 shows that a favorite for execution language has been the C language.

- **Distribution Library** provides the functionality to achieve parallel/distributed computations. The big favorite in this sense has been MPI, which is not surprising since it integrates well with the C language, and is probably the most used tool for parallelism in cluster computing. The dangers of directly programming with the distribution library are, of course, safely hidden away from the programmers who never interact with the distribution library. Recently, the trend has been to develop skeleton frameworks capable of interacting with more than one distribution library. For example, CO2 P3 S can use Threads, RMI or Sockets; Mallba can use Netstream or MPI; or JaSkel which uses AspectJ to execute the skeleton applications on different skeleton frameworks.

- **Type Safety** refers to the capability of detecting type incompatibility errors in skeleton program. Since the first skeleton frameworks were built on functional languages such as Haskell, type safety was simply inherited from the host language. Nevertheless, as custom languages were developed for skeleton programming, compilers had to be written to take type checking into consideration; which was not as difficult as skeleton nesting was not fully supported. Recently however, as we begun to host skeleton frameworks on object oriented languages with full nesting, the type safety issue has resurfaced. Unfortunately, type checking has been mostly overlooked (with the exception of QUAFF), and specially in Java based skeleton frameworks.

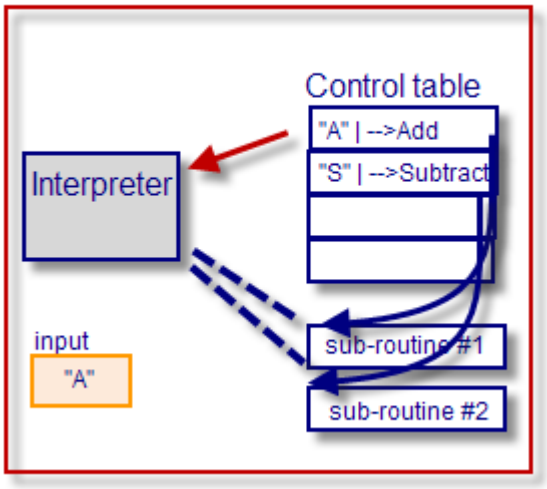
- **Skeleton Nesting** is the capability of hierarchical composition of skeleton patterns. Skeleton Nesting was identified as an important feature in skeleton programming from the very beginning, because it allows the composition of more complex patterns starting from a basic set of simpler patterns. Nevertheless, it has taken the community a long time to fully support arbitrary nesting of skeletons, mainly because of the scheduling and type verification difficulties. The trend is clear that recent skeleton frameworks support full nesting of skeletons.

- **File Access** is the capability to access and manipulate files from an application. In the past, skeleton programming has proven useful mostly for computational intensive applications, where small amounts of data require big amounts of computation time. Nevertheless, many distributed applications require or produce large amounts of data during their computation. This is the case for astrophysics, particle physics, bio-informatics, etc. Thus, providing file transfer support that integrates with skeleton programming is a key concern which has been mostly overlooked.

- **Skeleton Set** is the list of supported skeleton patterns. Skeleton sets vary greatly from one framework to the other, and more shocking, some skeletons with the same name have different semantics on different frameworks. The most common skeleton patterns in the literature are probably farm, pipe, and map.

Control table

Control tables are tables that control the program flow or play a major part in program control. There are no rigid rules concerning the structure or content of a control table - its only qualifying attribute is its ability to direct program flow in some way through its 'execution' by an associated interpreter. The design of such tables is sometimes referred to as **table driven design**. In some cases, control tables can be specific implementations of finite state machine-based automata-based programming.



This simple control table directs program flow according to the value of the single input variable. Each table entry holds a possible input value to be tested for equality (implied) and a relevant subroutine to perform in the action column. The name of the subroutine could be replaced by a relative subroutine number if pointers are not supported

Control tables often have the equivalent of conditional expressions or function references embedded in them, usually implied by their relative column position in the association list. Control tables reduce the need for programming similar structures or program statements over and over again. The two-dimensional nature of most tables makes them easier to view and update than the one-dimensional nature of program code. In some cases, non-programmers can be assigned to maintain the control tables.

Typical usage

- Transformation of input values to:
    - an index ,for later branching or pointer lookup
    - a program name, relative subroutine number, program label or program offset, to alter control flow

- Controlling a main loop in event-driven programming using a control variable for state transitions
- Controlling the program cycle for Online transaction processing applications

More advanced usage

- Acting as virtual instructions for a virtual machine processed by an interpreter

  similar to bytecode - but usually with operations implied by the table structure itself

Table structure

The tables can have multiple dimensions, of fixed or variable lengths and are usually portable between computer platforms, requiring only a change to the interpreter, not the algorithm itself - the logic of which is essentially embodied within the table structure and content. The structure of the table may be similar to a multimap associative array, where a data value (or combination of data values) may be mapped to one or more functions to be performed.

One dimensional tables

In perhaps its simplest implementation, a control table may sometimes be a one-dimensional table for *directly* translating a raw data value to a corresponding subroutine offset, index or pointer using the raw data value either directly as the index to the array, or by performing some basic arithmetic on the data beforehand. This can be achieved in constant time (without a linear search or binary search using a typical lookup table on an associative array). In most architectures, this can be accomplished in two or three machine instructions - without any comparisons or loops. The technique is known as a "trivial hash function" or, when used specifically for branch tables, "double dispatch". For this to be feasible, the range of all possible values of the data needs to be small (e.g. an ASCII or EBCDIC character value which have a range of hexadecimal '00' - 'FF'. If the actual range is *guaranteed* to be smaller than this, the array can be truncated to less than 256 bytes).

Table to translate raw ASCII values (A,D,M,S) to new subroutine index (1,4,3,2) in constant time using one-dimensional array

(gaps in the range are shown as '..' for this example, meaning 'all hex values up to next row'. The first two columns are not part of the array)

ASCII	Hex	Array
null	00	00
..	..	00

@	40	00
A	41	**01**
..	..	00
D	44	**04**
..	..	00
M	4D	**03**
..	..	00
S	53	**02**

In automata-based programming and pseudoconversational transaction processing, if the number of distinct program states is small, a "dense sequence" control variable can be used to efficiently dictate the entire flow of the main program loop.

A two byte raw data value would require a *minimum* table size of 65,534 bytes - to handle all input possibilities - whilst allowing just 256 different output values. However, this direct translation technique provides an extremely fast validation & conversion to a (relative) subroutine pointer if the heuristics, together with sufficient fast access memory, permits its use.

Branch tables

A branch table is a one dimensional 'array' of contiguous machine code branch/jump instructions to effect a multiway branch to a program label when branched into by an immediately preceding, and indexed branch. It is sometimes generated by an optimizing compiler to execute a switch statement - provided that the input range is small and dense, with few gaps (as created by the previous array example).

Although quite compact - compared to the multiple equivalent `If` statements - the branch instructions still carry some redundancy, since the branch opcode and condition code mask are repeated alongside the branch offsets. Control tables containing only the offsets to the program labels can be constructed to overcome this redundancy (at least in assembly languages) and yet requiring only minor execution time overhead compared to a conventional branch table.

Multi-dimensional tables

More usually, a control table can be thought of as a Truth table or as an executable ("binary") implementaion of a printed decision table (or a tree of decision tables, at several levels). They contain (often implied) propositions, together with one or more associated 'actions'. These actions are usually performed by generic or custom-built subroutines that are called by an "interpreter" program. The interpreter in this instance effectively functions as a virtual machine, that 'executes' the control table entries and thus provides a higher level of abstraction than the underlying code of the interpreter.

A control table can be constructed along similar lines to a language dependent switch statement but with the added possibility of testing for combinations of input values (using boolean style AND/OR conditions) and potentially calling multiple subroutines (instead of just a single set of values and 'branch to' program labels). (The switch statement construct in any case may not be available, or has confusingly differing implementations in high level languages (HLL). The control table concept, by comparison, has no intrinsic language dependencies, but might nevertheless be *implemented* differently according to the available data definition features of the chosen programming language.)

Table content

A control table essentially embodies the 'essence' of a conventional program, stripped of its programming language syntax and platform dependent components (e.g. IF/THEN DO.., FOR.., DO WHILE.., SWITCH, GOTO, CALL) and 'condensed' to its variables (e.g. input1), values (e.g. 'A','S','M' and 'D'), and subroutine identities (e.g. 'Add','subtract,..' or #1, #2,..). The structure of the table itself typically *implies* the (default) logical operations involved - such as 'testing for equality', performing a subroutine and 'next operation' or following the default sequence (rather than these being explicitly stated within program statements - as required in other programming paradigms).

A multi-dimensional control table will normally, as a minimum, contain value/action pairs and may additionally contain operators and type information such as, the location, size and format of input or output data, whether data conversion (or other run-time processing nuances) is required before or after processing (if not already implicit in the function itself). The table may or may not contain indexes or relative or absolute pointers to generic or customized primitives or subroutines to be executed depending upon other values in the "row".

The table illustrated below applies only to 'input1' since no specific input is specified in the table.

conditions and actions implied by structure

(implied) IF =	**(implied) perform**
value	action
value	action

(This side-by-side pairing of value and action has similarities to constructs in Event-driven programming, namely 'event-detection' and 'event-handling' but without (necessarily) the asynchronous nature of the event itself)

The variety of values that can be encoded within a control table is largely dependent upon the computer language used. Assembly language provides the widest scope for data types including (for the actions), the option of directly executable machine code. Typically a

control table will contain values for each possible matching class of input together with a corresponding pointer to an action subroutine. Some languages claim not to support pointers (directly) but nevertheless can instead support an index which can be used to represent a 'relative subroutine number' to perform conditional execution, controlled by the value in the table entry (e.g. for use in an optimized SWITCH statement - designed with zero gaps (i.e. a multiway branch)).

Comments positioned above each column (or even embedded textual documentation) can render a decision table 'human readable' even after 'condensing down' (encoding) to its essentials (and still broadly in-line with the original program specification - especially if a printed decision table, enumerating each unique action, is created before coding begins). The table entries can also optionally contain counters to collect run-time statistics for 'in-flight' or later optimization

Table location

Control tables can reside in static storage, on auxiliary storage, such as a flat file or on a database or may alternatively be partially or entirely built dynamically at program initialization time from parameters (which themselves may reside in a table). For optimum efficiency, the table should be memory resident when the interpreter begins to use it.

The interpreter and subroutines

The interpreter can be written in any suitable programming language including a high level language. A suitably designed generic interpreter, together with a well chosen set of generic subroutines (able to process the most commonly occurring primitives), would require additional conventional coding only for new custom subroutines (in addition to specifying the control table itself). The interpreter, optionally, may only apply to some well-defined sections of a complete application program (such as the main control loop) and not other, 'less conditional', sections (such as program initialization, termination and so on).

The interpreter does not need to be unduly complex, or produced by a programmer with the advanced knowledge of a compiler writer, and can be written just as any other application program - except that it is usually designed with efficiency in mind. Its primary function is to "execute" the table entries as a set of "instructions". There need be no requirement for parsing of control table entries and these should therefore be designed, as far as possible, to be 'execution ready', requiring only the "plugging in" of variables from the appropriate columns to the already compiled generic code of the interpreter. The program instructions are, in theory, infinitely extensible and constitute (possibly arbitrary) values within the table that are meaningful only to the interpreter. The control flow of the interpreter is normally by sequential processing of each table row but may be modified by specific actions in the table entries.

These arbitrary values can thus be designed with efficiency in mind - by selecting values that can be used as direct indexes to data or function pointers. For particular platforms/language, they can be specifically designed to minimize instruction path lengths using branch table values or even, in some cases such as in JIT compilers, consist of directly executable machine code "snippets" (or pointers to them).

The subroutines may be coded either in the same language as the interpreter itself or any other supported program language (provided that suitable inter-language 'Call' linkage mechanisms exist). The choice of language for the interpreter and/or subroutines will usually depend upon how portable it needs to be across various platforms. There may be several versions of the interpreter to enhance the portability of a control table. A subordinate control table pointer may optionally substitute for a subroutine pointer in the 'action' column(s) if the interpreter supports this construct, representing a conditional 'drop' to a lower logical level, mimicking a conventional structured program structure.

Performance considerations

At first sight, the use of control tables would appear to add quite a lot to a program's overhead, requiring, as it does, an interpreter process before the 'native' programming language statements are executed. This however is not always the case. By separating (or 'encapsulating') the executable coding from the logic, as expressed in the table, it can be more readily targeted to perform its function most efficiently. This may be experienced most obviously in a spreadsheet application - where the underlying spreadsheet software transparently converts complex logical 'formulae' in the most efficient manner it is able, in order to display its results.

The examples below have been chosen partly to illustrate potential performance gains that may not only *compensate* significantly for the additional tier of abstraction, but also *improve* upon - what otherwise might have been - less efficient, less maintainable and lengthier code. Although the examples given are for a 'low level' assembly language and for the 'C' language, it can be seen, in both cases, that very few lines of code are required to implement the control table approach and yet can achieve very significant constant time performance improvements, reduce repetitive source coding and aid clarity, as compared with verbose conventional program language constructs.

Examples of control tables

The following examples are arbitrary (and based upon just a single input for simplicity), however the intention is merely to demonstrate how control flow can be effected via the use of tables instead of regular program statements. It should be clear that this technique can easily be extended to deal with multiple inputs, either by increasing the number of columns or utilizing multiple table entries (with optional and/or operator). Similarly, by using (hierarchical) 'linked' control tables, structured programming can be accomplished (optionally using indentation to help highlight subordinate control tables).

"CT1" is an example of a control table that is a simple lookup table. The first column represents the input value to be tested (by an implied 'IF input1 = x') and, if TRUE, the corresponding 2nd column (the 'action') contains a subroutine address to perform by a call (or jump to - similar to a SWITCH statement). It is, in effect, a multiway branch with return (a form of "dynamic dispatch"). The last entry is the default case where no match is found.

CT1

input 1	**pointer**
A	-->Add
S	-->Subtract
M	-->Multiply
D	-->Divide
?	-->Default

For programming languages that support pointers within data structures alongside other data values, the above table (CT1) can be used to direct control flow to an appropriate subroutines according to matching value from the table (without a column to indicate otherwise, equality is assumed in this simple case).

Assembly language example for IBM/360 (maximum 16Mb address range) or Z/Architecture

No attempt is made to optimize the lookup in coding for this first example ,and it uses instead a simple linear search technique - purely to illustrate the concept and demonstrate fewer source lines. To handle all 256 different input values, approximately 265 lines of source code would be required (mainly single line table entries) whereas multiple 'compare and branch' would have normally required around 512 source lines (the size of the binary is also approximately halved, each table entry requiring only 4 bytes instead of approximately 8 bytes for a series of 'compare immediate'/branch instructions (For larger input variables, the saving is even greater).

```
  * ------------------ interpreter ----------------------------------
--------*
           LM     R14,R0,=A(4,CT1,N)               Set R14=4, R15 -->
table, and R0 =no. of entries in table (N)
  TRY      CLC    INPUT1,0(R15)         *********  Found value in table
entry ?
           BE     ACTION                * loop  *  YES, Load register
pointer to sub-routine from table
           AR     R15,R14               *       *  NO ,Point to next
entry in CT1 by adding R14 (=4)
           BCT    R0,TRY                *********  Back until count
exhausted, then drop through
  .             default action                        ... none of the
values in table match, do something else
```

```
           LA     R15,4(R15)                        point to default
entry (beyond table end)
  ACTION   L      R15,0(R15)                        get pointer into
R15,from where R15 points
           BALR   R14,R15                           Perform the sub-
routine ("CALL" and return)
           B      END                               go terminate this
program
  * ------------------ control table ---------------------------------
-------*
  *                 | this column of allowable EBCDIC or ASCII values
is tested '=' against variable 'input1'
  *                 |      | this column is the 3-byte address of the
appropriate subroutine
  *                 v      v
  CT1      DC     C'A',AL3(ADD)                     START of Control
Table (4 byte entry length)
           DC     C'S',AL3(SUBTRACT)
           DC     C'M',AL3(MULTIPLY)
           DC     C'D',AL3(DIVIDE)
  N        EQU    (*-CT1)/4                         number of valid
entries in table (total length / entry length)
           DC     C'?',AL3(DEFAULT)                 default entry - used
on drop through to catch all
  INPUT1   DS     C                                 input variable is in
this variable
  * ------------------ sub-routines ----------------------------------
-------*
  ADD      CSECT                                    sub-routine #1
(shown as separate CSECT here but might
  .
alternatively be in-line code)
  .            instruction(s) to add
           BR     R14                               return
  SUBTRACT CSECT                                    sub-routine #2
  .            instruction(s) to subtract
           BR     R14                               return
  . etc..
```

improving the performance of the interpreter in above example

To make a selection in the example above, the average instruction path length (excluding the subroutine code) is '4n/2 +3' , but can easily be reduced, where n = 1 to 64, to a constant time $O(1)$ with a path length of '5' with *zero comparisons*, if a 256 byte translate table is first utilized to create a *direct* index to CT1 from the raw EBCDIC data. Where n = 6, this would then be equivalent to just 3 sequential compare & branch instructions. However, where n<=64, on average it would need approximately 13 *times* less instructions than using multiple compares. Where n=1 to 256, on average it would use approximately 42 *times* less instructions - since, in this case, one additional instruction would be required (to multiply the index by 4).

Improved interpreter (up to **26 times less executed instructions** than the above example on average, where n= 1 to 64 and up to 13 times less than would be needed using multiple comparisons).

To handle 64 different input values, approximately 85 lines of source code (or less) are required (mainly single line table entries) whereas multiple 'compare and branch' would require around 128 lines (the size of the binary is also almost halved - despite the additional 256 byte table required to extract the 2nd index).

```
  * ------------------ interpreter ------------------------------------
--------*
           SR     R14,R14               *********  Set R14=0
  CALC     IC     R14,INPUT1            * calc  *  put EBCDIC byte into
lo order bits (24-31) of R14
           IC     R14,CT1X(R14)         *       *  use EBCDIC value as
index on table 'CT1X' to get new index
  FOUND    L      R15,CT1(R14)          *********  get pointer to
subroutine using index (0,4, 8 etc.)
           BALR   R14,R15                          Perform the sub-
routine ("CALL" and return or Default)
           B      END                              go terminate this
program
  * --------------- additional translate table (EBCDIC --> pointer
table INDEX)  256 bytes----*
  CT1X     DC
12AL1(00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00)   12 identical
sets of 16 bytes of x'00
  *
representing X'00 - x'BF'
           DC     AL1(00,04,00,00,16,00,00,00,00,00,00,00,00,00,00,00)
..x'C0' - X'CF'
           DC     AL1(00,00,00,00,12,00,00,00,00,00,00,00,00,00,00,00)
..x'D0' - X'DF'
           DC     AL1(00,00,08,00,00,00,00,00,00,00,00,00,00,00,00,00)
..x'E0' - X'EF'
           DC     AL1(00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00)
..x'F0' - X'FF'
  * the assembler can be used to automatically calculate the index
values and make the values more user friendly
  * (for e.g. '04' could be replaced with the symbolic expression
'PADD-CT1' in table CT1X above)
  * modified CT1 (added a default action when index = 00, single
dimension, full 31 bit address)
  CT1      DC     A(DEFAULT)          index       =00      START of
Control Table (4 byte address constants)
  PADD     DC     A(ADD)                          =04
  PSUB     DC     A(SUBTRACT)                     =08
  PMUL     DC     A(MULTIPLY)                     =12
  PDIV     DC     A(DIVIDE)                       =16
  * the rest of the code remains the same as first example
```

Further improved interpreter (up to **21 times less executed instructions (where n>=64)** than the first example on average and up to 42 *times* less than would be needed using multiple comparisons).

To handle 256 different input values, approximately 280 lines of source code or less, would be required (mainly single line table entries), whereas multiple 'compare and branch' would require around 512 lines (the size of the binary is also almost halved once more).

```
  * ------------------ interpreter ----------------------------------
--------*
          SR     R14,R14               *********  Set R14=0
  CALC    IC     R14,INPUT1            * calc  *  put EBCDIC byte into
lo order bits (24-31) of R14
          IC     R14,CT1X(R14)         *       *  use EBCDIC value as
index on table 'CT1X' to get new index
          SLL    R14,2                 *       *  multiply index by 4
(additional instruction)
  FOUND   L      R15,CT1(R14)          *********  get pointer to
subroutine using index (0,4, 8 etc.)
          BALR   R14,R15                          Perform the sub-
routine ("CALL" and return or Default)
          B      END                              go terminate this
program
  * --------------- additional translate table (EBCDIC --> pointer
table INDEX)  256 bytes----*
  CT1X    DC
12AL1(00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00)   12 identical
sets of 16 bytes of x'00'
  *
representing X'00 - x'BF'
          DC     AL1(00,01,00,00,04,00,00,00,00,00,00,00,00,00,00,00)
..x'C0' - X'CF'
          DC     AL1(00,00,00,00,03,00,00,00,00,00,00,00,00,00,00,00)
..x'D0' - X'DF'
          DC     AL1(00,00,02,00,00,00,00,00,00,00,00,00,00,00,00,00)
..x'E0' - X'EF'
          DC     AL1(00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00)
..x'F0' - X'FF'
  * the assembler can be used to automatically calculate the index
values and make the values more user friendly
  * (for e.g. '01' could be replaced with the symbolic expression
'PADD-CT1/4' in table CT1X above)
  * modified CT1 (index now based on 0,1,2,3,4  not 0,4,8,12,16 to
allow all 256 variations)
  CT1     DC     A(DEFAULT)          index       =00      START of
Control Table (4 byte address constants)
  PADD    DC     A(ADD)                          =01
  PSUB    DC     A(SUBTRACT)                     =02
  PMUL    DC     A(MULTIPLY)                     =03
  PDIV    DC     A(DIVIDE)                       =04
  * the rest of the code remains the same as the 2nd example
```

C language example This example in 'C' uses two tables , the first (CT1) is a simple linear search one-dimensional lookup table - to obtain an index by matching the input (x), and the second, associated table (CT1p), is a table of addresses of labels to jump to.

```
 static const char  CT1[] = {  "A",   "S",        "M",        "D" };
/* permitted input  values */
 static const void *CT1p[] = { &&Add, &&Subtract, &&Multiply, &&Divide,
&&Default};           /* labels to goto & default*/
 for (int i = 0; i < sizeof(CT1); i++)      /* loop thru ASCII values
*/
   {if (x == CT1[i]) goto *CT1p[i]; }       /* found --> appropriate
label                                              */
 goto *CT1p[i+1];                           /* not found --> default
label                                              */
```

This can be made more efficient if a 256 byte table is used to translate the raw ASCII value (x) directly to a dense sequential index value for use in directly locating the branch address from CT1p (i.e. "index mapping" with a byte-wide array). It will then execute in constant time for all possible values of x (If CT1p contained the names of functions instead of labels, the jump could be replaced with a dynamic function call, eliminating the switch-like goto - but decreasing performance by the additional cost of function housekeeping).

```
 static const void *CT1p[] = {&&Default, &&Add, &&Subtract, &&Multiply,
&&Divide};
 /* the 256 byte table, below, holds values (1,2,3,4), in corresponding
ASCII positions (A,S,M,D), all others set to 0x00 */
 static const char CT1x[]={

'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x01','\x00','\x00','\x04','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x03','\x00','\x00',

'\x00','\x00','\x00','\x02','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',
```

```
'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x00','\x00','\x00','\x03','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00',

'\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','\x00','
\x00','\x00','\x00','\x00','\x00','\x00'};
 /* the following code will execute in constant time, irrespective of
the value of the input character (x)                    */
 i = CT1x(x);            /* extract the correct subroutine index from
table CT1x using its ASCII value as an index initially  */
 goto *CT1p[i];          /* goto (Switch to) the label corresponding to
the index (0=default,1= Add,2= Subtract,.) - see CT1p */
```

The next example below illustrates how a similar effect can be achieved in languages that do **not** support pointer definitions in data structures but **do** support indexed branching to a subroutine - contained within a (0-based) array of subroutine pointers. The table (CT2) is used to extract the index (from 2nd column) to the pointer array (CT2P). If pointer arrays are *not* supported, a SWITCH statement or equivalent can be used to alter the control flow to one of a sequence of program labels (e.g.: case0,case1,case2,case3,case4) which then either process the input directly, or else perform a call (with return) to the appropriate subroutine (default,Add,Subtract,Multiply or Divide,..) to deal with it.

CT2

input 1	**subr #**
A	**1**
S	**2**
M	**3**
D	**4**
?	**0**

As in above examples, it is possible to very efficiently translate the potential ASCII input values (A,S,M,D or unknown) into a pointer array index without actually using a table lookup, but is shown here as a table for consistency with the first example.

CT2P pointer array

pointer array

-->default
-->Add
-->Subtract
-->Multiply
-->Divide
-->?other

Multi-dimensional control tables can be constructed (i.e. customized) that can be 'more complex' than the above examples that might test for multiple conditions on multiple inputs or perform more than one 'action', based on some matching criteria. An 'action' can include a pointer to another subordinate control table. The simple example below has had an *implicit* 'OR' condition incorporated as an extra column (to handle lower case input, however in this instance, this could equally have been handled simply by having an extra entry for each of the lower case characters specifying the same subroutine identifier as the upper case characters). An extra column to count the actual run-time events for each input as they occur is also included.

CT3

input 1	alternate	subr #	count
A	**a**	**1**	**0**
S	**s**	**2**	**0**
M	**m**	**3**	**0**
D	**d**	**4**	**0**
?	**?**	**0**	**0**

The control table entries are then much more similar to conditional statements in procedural languages but, crucially, without the actual (language dependent) conditional statements (i.e. instructions) being present (the generic code is *physically* in the interpreter that processes the table entries, not in the table itself - which simply embodies the program logic via its structure and values).

In tables such as these, where a series of similar table entries defines the entire logic, a table entry number or pointer may effectively take the place of a program counter in more conventional programs and may be reset in an 'action', also specified in the table entry. The example below (CT4) shows how extending the earlier table, to include a 'next' entry (and/or including an 'alter flow' (jump) subroutine) can create a loop (This example is actually not the most efficient way to construct such a control table but, by demonstrating a gradual 'evolution' from the first examples above, shows how additional columns can be used to modify behaviour.) The fifth column demonstrates that more than one action can be initiated with a single table entry - in this case an action to be performed *after* the normal processing of each entry ('-' values mean 'no conditions' or 'no action').

Structured programming or "Goto-less" code, (incorporating the equivalent of 'DO WHILE' or 'for loop' constructs), can also be accommodated with suitably designed and 'indented' control table structures.

CT4 (a complete 'program' to read input1 and process, repeating until 'E' encountered)

input 1	alternate	subr #	count	jump
-	-	5	0	-
E	e	7	0	-
A	a	1	0	-
S	s	2	0	-
M	m	3	0	-
D	d	4	0	-
?	?	0	0	-
-	-	6	0	1

CT4P pointer array

pointer array
-->Default
-->Add
-->Subtract
-->Multiply
-->Divide
-->Read Input1
-->Alter flow
-->End

Table-driven rating

In the specialist field of telecommunications rating (concerned with the determining the cost of a particular call), **table-driven rating** techniques illustrate the use of control tables in applications where the rules may change frequently because of market forces. The tables that determine the charges may be changed at short notice by non-programmers in many cases..

If the algorithms are not pre-built into the interpreter (and therefore require additional runtime interpretation of an expression held in the table), it is known as "Rule-based Rating" rather than table-driven rating (and consequently consumes significantly more overhead).

Spreadsheets

A spreadsheet data sheet can be thought of as a two dimensional control table, with the non empty cells representing data to the underlying spreadsheet program (the interpreter). The cells containing formula are usually prefixed with an equals sign and simply designate a special type of data input that dictates the processing of other referenced cells - by altering the control flow within the interpreter. It is the externalization of formulae from the underlying interpreter that clearly identifies both spreadsheets, and the above cited "rule based rating" example as readily identifiable instances of the use of control tables by non programmers.

Programming paradigm

If the control tables technique could be said to belong to any particular programming paradigm, the closest analogy might be Automata-based programming or "reflective" (a form of metaprogramming - since the table entries could be said to 'modify' the behaviour of the interpreter). The interpreter itself however, and the subroutines, can be programmed using any one of the available paradigms or even a mixture. The table itself can be essentially a collection of "raw data" values that do not even need to be compiled and could be read in from an external source (except in specific, platform dependent, implementations using memory pointers directly for greater efficiency).

Analogy to bytecode / virtual machine instruction set

A multi-dimensional control table has some conceptual similarities to bytecode operating on a virtual machine, in that a platform dependent "interpreter" program is usually required to perform the actual execution (that is largely conditionally determined by the tables content). There are also some conceptual similarities to the recent Common Intermediate Language (CIL) in the aim of creating a common intermediate 'instruction set' that is independent of platform (but unlike CIL, no pretentions to be used as a common resource for other languages).

Instruction fetch

When a multi-dimensional control table is used to determine program flow, the normal "hardware" Program Counter function is effectively simulated with either a pointer to the first (or next) table entry or else an index to it. "Fetching" the instruction involves decoding the *data* in that table entry - without necessarily copying all or some of the data within the entry first. Programming languages that are able to use pointers have the dual advantage that less overhead is involved, both in accessing the contents and also advancing the counter to point to the next table entry after execution. Calculating the next 'instruction' address (i.e. table entry) can even be performed as an optional additional action of every individual table entry allowing loops and or jump instructions at any stage.

Monitoring control table execution

The interpreter program can optionally save the program counter (and other relevant details depending upon instruction type) at each stage to record a full or partial trace of the actual program flow for debugging purposes, hot spot detection, code coverage analysis and performance analysis.

Advantages

- clarity - Information tables are ubiquitous and mostly inherently understood even by the general public (especially fault diagnostic tables in product guides)
- portability - can be designed to be 100% language independent (and platform independent - except for the interpreter)
- flexibility - ability to execute either primitives or subroutines transparently and be custom designed to suit the problem
- compactness - table usually shows condition/action pairing side-by-side (without the usual platform/language implementation dependencies), often also resulting in
    - binary file - reduced in size through less duplication of instructions
    - source file - reduced in size through elimination of multiple conditional statements
    - improved program load (or download) speeds
- maintainability - tables often reduce the number of source lines needed to be maintained v. multiple compares
- locality of reference - compact tables structures result in tables remaining in cache
- code re-use - the "interpreter" is usually reusable. Frequently it can be easily adapted to new programming tasks using precisely the same technique and can grow 'organically' becoming, in effect, a standard library of tried and tested subroutines, controlled by the table definitions.
- efficiency - systemwide optimization possible. Any performance improvement to the interpreter usually improves *all* applications using it.
- extensible - new 'instructions' can be added - simply by extending the interpreter
- interpreter can be written like an application program

Optionally:-

- the interpreter can be introspective and "self optimize" using runtime metrics collected within the table itself. The interpreter can also optionally choose the most efficient lookup technique dynamically from metrics gathered at run-time (e.g. size of array, range of values, sorted or unsorted)
- dynamic dispatch - common functions can be pre-loaded and less common functions fetched only on first encounter to reduce memory usage. In-table memoization can be employed to achieve this.

- The interpreter can have debugging, trace and monitor features built-in - that can then be switched on or off at will according to test or 'live' mode
- control tables can be built 'on-the-fly' (according to some user input or from parameters) and then executed by the interpreter (without building code literally).

Disadvantages

- training requirement - application programmers are not usually trained to produce generic solutions

The following mainly apply to their use in multi-dimensional tables, not the one dimensional tables discussed earlier.

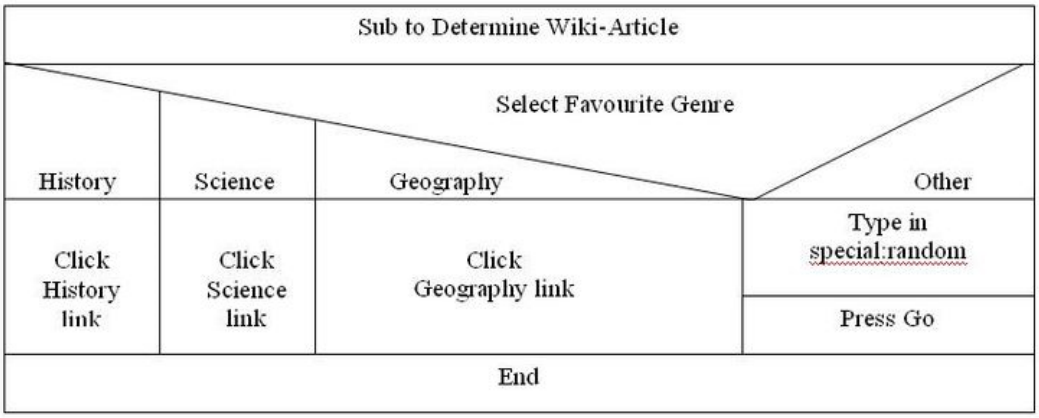
- overhead - some increase because of extra level of indirection caused by virtual instructions having to be 'interpreted' (this however can usually be more than offset by a well designed generic interpreter taking full advantage of efficient direct translate, search and conditional testing techniques that may not otherwise have been utilized)
- Complex expressions cannot always be used *directly* in data table entries for comparison purposes

  (these 'intermediate values' can however be calculated beforehand instead within a subroutine and their values referred to in the conditional table entries. Alternatively, a subroutine can perform the complete complex conditional test (as an unconditional 'action') and, by setting a truth flag as its result, it can then be tested in the next table entry.)

Chapter 5

Nassi–Shneiderman Diagram

In computer programming, a **Nassi–Shneiderman diagram (NSD)** is a graphical design representation for structured programming.



Example of a Nassi–Shneiderman diagram

Developed in 1972 by Isaac Nassi and Ben Shneiderman, these diagrams are also called *structograms*, as they show a program's structures.

Overview

Following a top-down design, the problem at hand is reduced into smaller and smaller subproblems, until only simple statements and control flow constructs remain. Nassi–Shneiderman diagrams reflect this top-down decomposition in a straight-forward way, using nested boxes to represent subproblems. Consistent with the philosophy of structured programming, Nassi–Shneiderman diagrams have no representation for a GOTO statement.

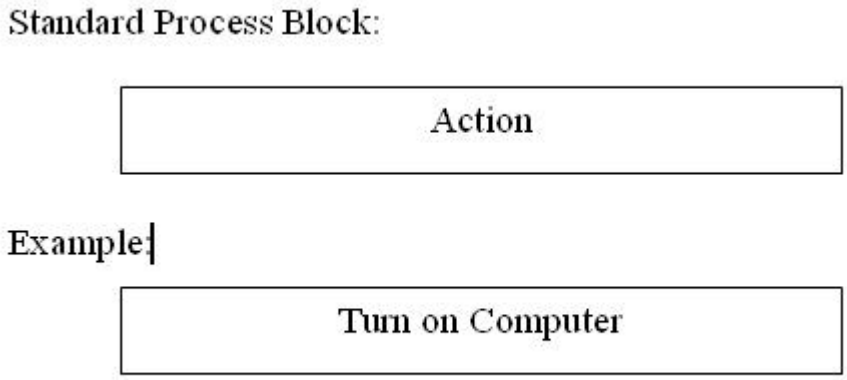
Nassi–Shneiderman diagrams are only rarely used for formal programming. Their abstraction level is close to structured program code and modifications require the whole

diagram to be redrawn. Nonetheless, they can be useful for sketching processes and high-level designs.

Nassi–Shneiderman diagrams are (almost) isomorphic with flowcharts. Everything you can represent with a Nassi–Shneiderman diagram you can also represent with a flowchart. For flowcharts of programs, almost everything you can represent with a flowchart you can also represent with a Nassi–Shneiderman diagram. The exceptions are things like goto, and the C programming language loop break and continue statements.

Diagrams

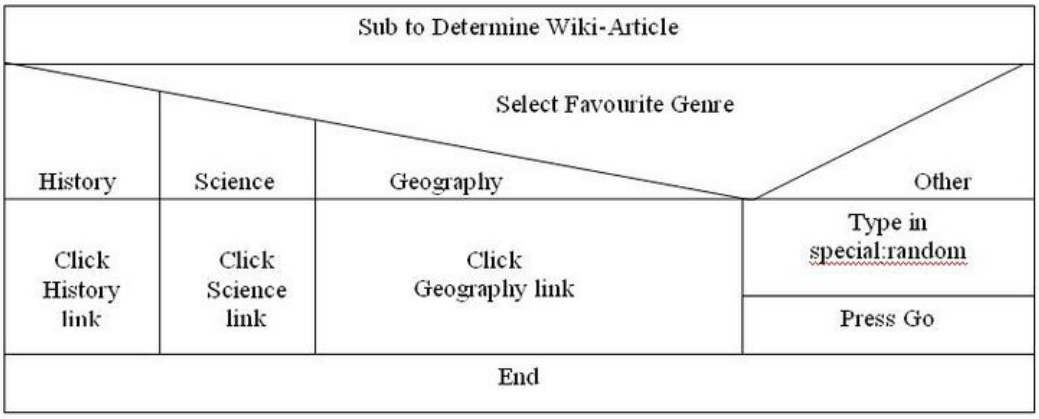
Process blocks: the process block represents the simplest of steps and requires no analyses. When a process block is encountered the action inside the block is performed and we move onto the next block.



Process blocks

Branching blocks: there are two types of branching blocks. First is the simple True/False or Yes/No branching block which offers the program two paths to take depending on whether or not a condition has been fulfilled. These blocks can be used as a looping procedure stopping the program from continuing until a condition has been fulfilled.

The second type of branching block is a multiple branching block. This block is used when a select case is needed in a program. The block usually contains a question or select case. The block provides the program with an array of choices and is often used in conjunction with sub process blocks to save space.

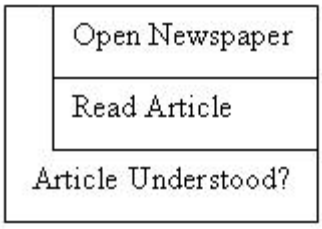


Multiple branching blocks

Testing loops: this block allows the program to loop one or a set of process until a particular condition is fulfilled. The process blocks covered by each loop are subset with a side-bar extending out from the condition.

There are two main types of testing loops, test first and test last blocks. The only difference between the two is the order in which the steps involved are completed. In the test first situation, when the program encounters the block it tests to see if the condition is fulfilled, then, if it is not completes the process blocks and then loops back. The test is performed again and, if the condition is still unfulfilled, the processes again. If at any stage the condition is fulfilled the program skips the process blocks and continues onto the next block.

The test last block is simply reversed, the process blocks are completed before the test is performed. The test last loop allows for the process blocks to be performed at least once before the first test.



Test last loop block

Chapter 6

Profiling (Computer Programming) and Integrated Development Environment

Profiling (computer programming)

In software engineering, **program profiling**, **software profiling** or simply **profiling**, a form of dynamic program analysis (as opposed to static code analysis), is the investigation of a program's behavior using information gathered as the program executes. The usual purpose of this analysis is to determine which sections of a program to optimize - to increase its overall speed, decrease its memory requirement or sometimes both.

- A **(code) profiler** is a **performance analysis** tool that, most commonly, measures only the frequency and duration of function calls, but there are other specific types of profilers (e.g. memory profilers) in addition to more comprehensive profilers, capable of gathering extensive performance data.
- An instruction set simulator which is also — by necessity — a profiler, can measure the totality of a program's behaviour from invocation to termination.

Gathering program events

Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, instruction set simulation, operating system hooks, and performance counters. The usage of profilers is 'called out' in the performance engineering process.

Use of profilers

Program analysis tools are extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical sections of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing... (ATOM, PLDI, '94)

The output of a profiler may be:-

- A statistical *summary* of the events observed (a **profile**)

  Summary profile information is often shown annotated against the source code statements where the events occur, so the size of measurement data is linear to the code size of the program.

```
/* ------------ source------------------------ count */
0001             IF X = "A"                     0055
0002                THEN DO
0003                  ADD 1 to XCOUNT           0032
0004                ELSE
0005             IF X = "B"                     0055
```

- A stream of recorded events (a **trace**)

  For sequential programs, a summary profile is usually sufficient, but performance problems in parallel programs (waiting for messages or synchronization issues) often depend on the time relationship of events, thus requiring a full trace to get an understanding of what is happening.
  The size of a (full) trace is linear to the program's instruction path length, making it somewhat impractical. A trace may therefore be initiated at one point in a program and terminated at another point to limit the output.

- An ongoing interaction with the hypervisor (continuous or periodic monitoring via on-screen display for instance)

  This provides the opportunity to switch a trace on or off at any desired point during execution in addition to viewing on-going metrics about the (still executing) program. It also provides the opportunity to suspend asynchronous processes at critical points to examine interactions with other parallel processes in more detail.

History

Performance analysis tools existed on IBM/360 and IBM/370 platforms from the early 1970s, usually based on timer interrupts which recorded the Program status word (PSW) at set timer intervals to detect "hot spots" in executing code. This was an early example of sampling (see below). In early 1974, Instruction Set Simulators permitted full trace and other performance monitoring features.

Profiler-driven program analysis on Unix dates back to at least 1979, when Unix systems included a basic tool "prof" that listed each function and how much of program execution time it used. In 1982, gprof extended the concept to a complete call graph analysis

In 1994, Amitabh Srivastava and Alan Eustace of Digital Equipment Corporation published a paper describing ATOM . ATOM is a platform for converting a program into its own profiler. That is, at compile time, it inserts code into the program to be analyzed. That inserted code outputs analysis data. This technique - modifying a program to analyze itself - is known as "instrumentation".

In 2004, both the Gprof and ATOM papers appeared on the list of the 50 most influential PLDI papers of all time.

Profiler types based on output

Flat profiler

Flat profilers compute the average call times, from the calls, and do not break down the call times based on the callee or the context.

Call-graph profiler

Call graph profilers show the call times, and frequencies of the functions, and also the call-chains involved based on the callee. However context is not preserved.

Methods of data gathering

Event based profilers

The programming languages listed here have event-based profilers:

- Java: the JVMTI (JVM Tools Interface) API, formerly JVMPI (JVM Profiling Interface), provides hooks to profilers, for trapping events like calls, class-load, unload, thread enter leave.
- .NET: Can attach a profiling agent as a COM server to the CLR. Like Java, the runtime then provides various callbacks into the agent, for trapping events like method JIT / enter / leave, object creation, etc. Particularly powerful in that the profiling agent can rewrite the target application's bytecode in arbitrary ways.
- Python: Python profiling includes the profile module, hotshot (which is call-graph based), and using the 'sys.setprofile()' module to trap events like c_{call,return,exception}, python_{call,return,exception}.
- Ruby: Ruby also uses a similar interface like Python for profiling. Flat-profiler in profile.rb, module, and ruby-prof a C-extension are present.

Statistical profilers

Some profilers operate by sampling. A sampling profiler probes the target program's program counter at regular intervals using operating system interrupts. Sampling profiles are typically less numerically accurate and specific, but allow the target program to run at near full speed.

The resulting data are not exact, but a statistical approximation. *The actual amount of error is usually more than one sampling period. In fact, if a value is n times the sampling period, the expected error in it is the square-root of n sampling periods.*

In practice, sampling profilers can often provide a more accurate picture of the target program's execution than other approaches, as they are not as intrusive to the target program, and thus don't have as many side effects (such as on memory caches or instruction decoding pipelines). Also since they don't affect the execution speed as much, they can detect issues that would otherwise be hidden. They are also relatively immune to over-evaluating the cost of small, frequently called routines or 'tight' loops. They can show the relative amount of time spent in user mode versus interruptible kernel mode such as system call processing.

Still, kernel code to handle the interrupts entails a minor loss of cpu cycles, diverted cache usage, and is unable to distinguish the various tasks occurring in uninterruptible kernel code (microsecond-range activity).

Dedicated hardware can go beyond this: some recent MIPS processors JTAG interface have a PCSAMPLE register, which samples the program counter in a truly undetectable manner.

Some of the most commonly used statistical profilers are AMD CodeAnalyst, Apple Inc. Shark, gprof, Intel VTune and Parallel Amplifier (part of Intel Parallel Studio).

Instrumenting profilers

Some profilers **instrument** the target program with additional instructions to collect the required information.

Instrumenting the program can cause changes in the performance of the program, potentially causing inaccurate results and heisenbugs. Instrumenting will always have some impact on the program execution, typically always slowing it. However, instrumentation can be very specific and be carefully controlled to have a minimal impact. The impact on a particular program depends on the placement of instrumentation points and the mechanism used to capture the trace. Hardware support for trace capture means that on some targets, instrumentation can be on just one machine instruction. The impact of instrumentation can often be deducted (i.e. eliminated by subtraction) from the results.

gprof is an example of a profiler that uses both instrumentation and sampling. Instrumentation is used to gather caller information and the actual timing values are obtained by statistical sampling.

Instrumentation

- **Manual**: Performed by the programmer, e.g. by adding instructions to explicitly calculate runtimes, simply count events or calls to measurement APIs such as the Application Response Measurement standard.
- **Automatic source level**: instrumentation added to the source code by an automatic tool according to an instrumentation policy.
- **Compiler assisted**: Example: "gcc -pg ..." for gprof, "quantify g++ ..." for Quantify
- **Binary translation**: The tool adds instrumentation to a compiled binary. Example: ATOM
- **Runtime instrumentation**: Directly before execution the code is instrumented. The program run is fully supervised and controlled by the tool. Examples: Pin, Valgrind
- **Runtime injection**: More lightweight than runtime instrumentation. Code is modified at runtime to have jumps to helper functions. Example: DynInst

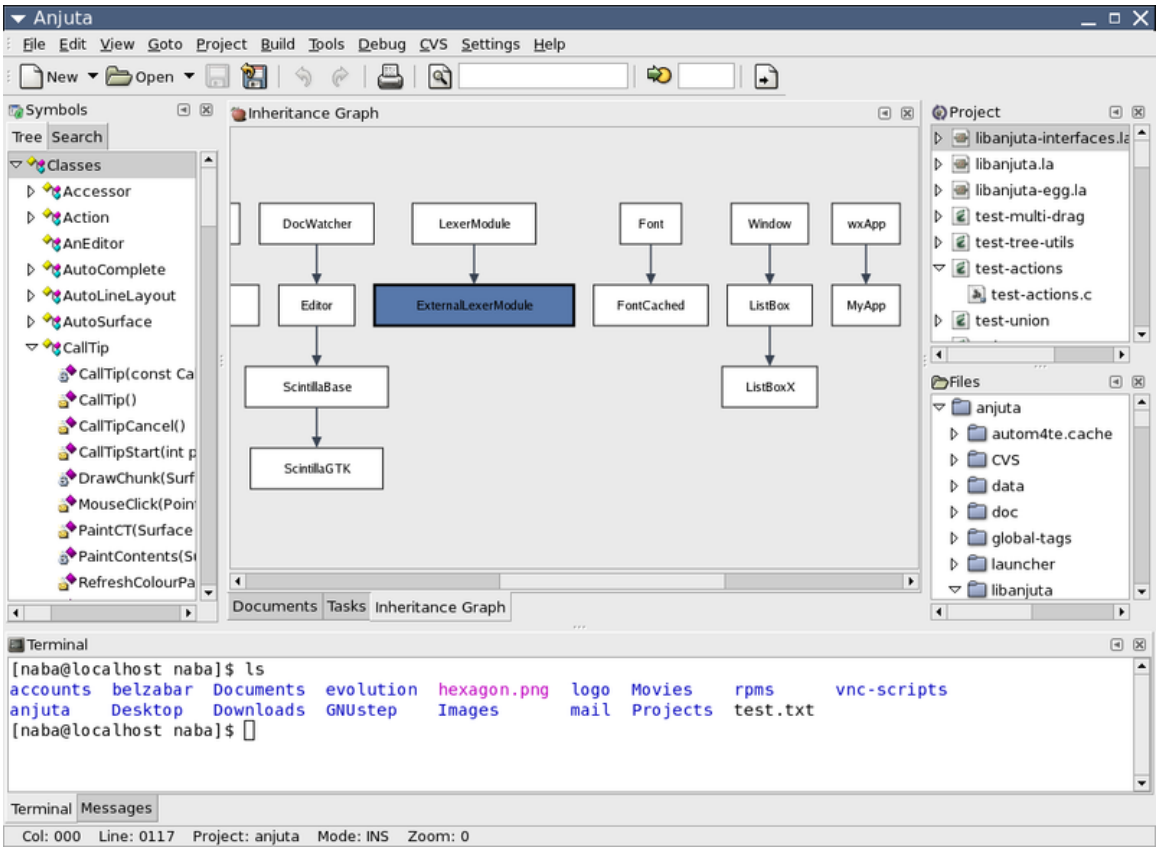
Interpreter instrumentation

- **Interpreter debug** options can enable the collection of performance metrics as the interpreter encounters each target statement. A bytecode, control table or JIT interpreters are three examples that usually have complete control over execution of the target code, thus enabling extremely comprehensive data collection opportunities.

Hypervisor/Simulator

- **Hypervisor**: Data are collected by running the (usually) unmodified program under a hypervisor. Example: SIMMON
- **Simulator** and **Hypervisor**: Data collected interactively and selectively by running the unmodified program under an Instruction Set Simulator. Examples: SIMON and OLIVER.

Integrated development environment



Anjuta, a C and C++ IDE for the GNOME environment

An **integrated development environment (IDE)** also known as *integrated design environment* or *integrated debugging environment* is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of:

- a source code editor
- a compiler and/or an interpreter
- build automation tools
- a debugger

Sometimes a version control system and various tools are integrated to simplify the construction of a GUI. Many modern IDEs also have a class browser, an object inspector, and a class hierarchy diagram, for use with object-oriented software development.

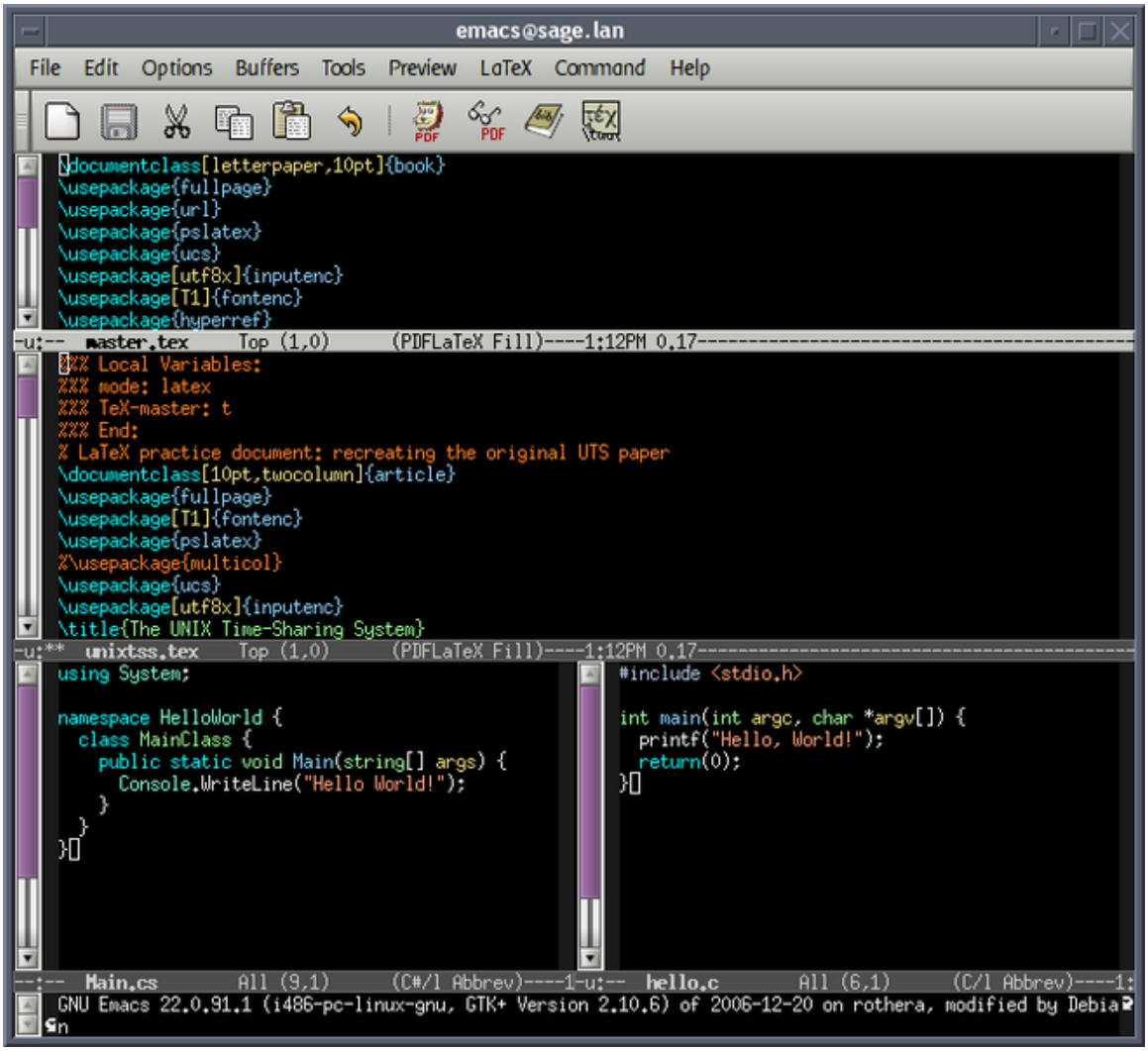
Overview

IDEs are designed to maximize programmer productivity by providing tightly-knit components with similar user interfaces. This should mean that the programmer has much less mode switching to do than when using discrete development programs. However, because an IDE is by its very nature a complicated piece of software, this high productivity only occurs after a lengthy learning process.

Typically an IDE is dedicated to a specific programming language, so as to provide a feature set which most closely matches the programming paradigms of the language. However, some multiple-language IDEs are in use, such as Eclipse, ActiveState Komodo, recent versions of NetBeans, Microsoft Visual Studio, WinDev, and Xcode.

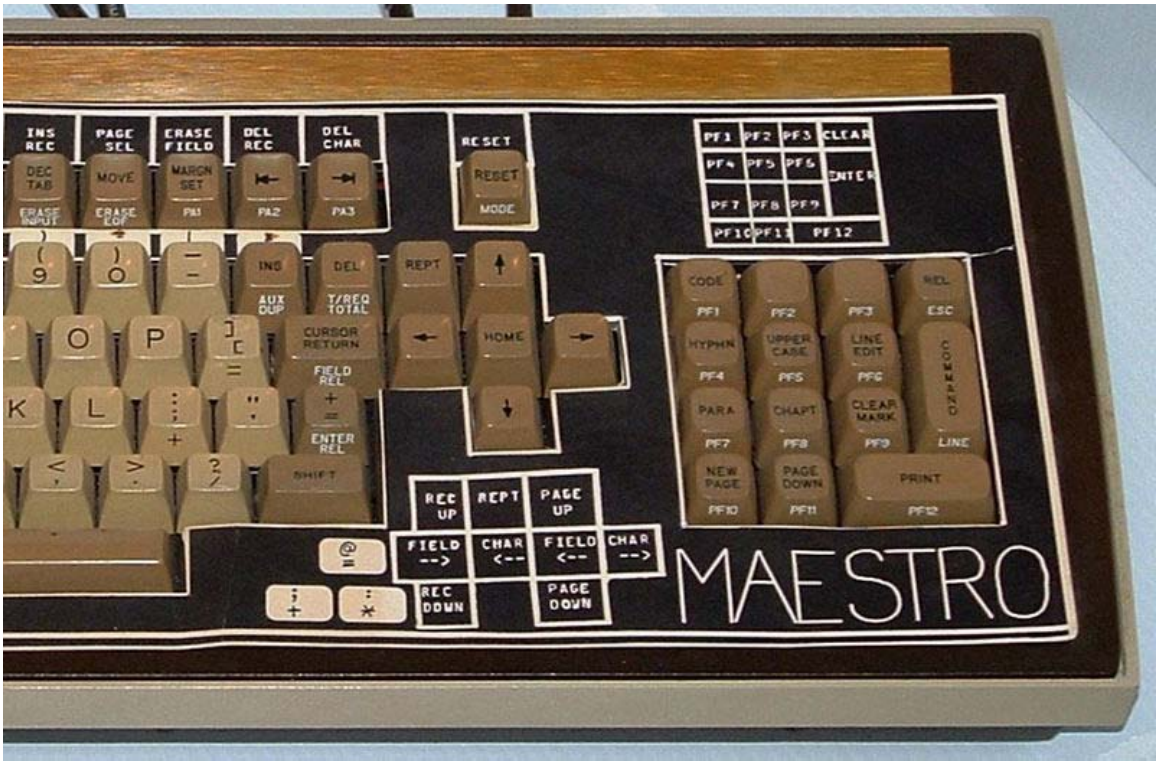
IDEs typically present a single program in which all development is done. This program typically provides many features for authoring, modifying, compiling, deploying and debugging software. The aim is to abstract the configuration necessary to piece together command line utilities in a cohesive unit, which theoretically reduces the time to learn a language, and increases developer productivity. It is also thought that the tight integration of development tasks can further increase productivity. For example, code can be compiled while being written, providing instant feedback on syntax errors. While most modern IDEs are graphical, IDEs in use before the advent of windowing systems (such as Microsoft Windows or X11) were text-based, using function keys or hotkeys to perform various tasks (Turbo Pascal is a common example). This contrasts with software development using unrelated tools, such as vi, GCC or make.

History



GNU Emacs, an extensible editor which is commonly used as an IDE on Unix-like systems

IDEs initially became possible when developing via a console or terminal. Early systems could not support one, since programs were prepared using flowcharts, entering programs with punched cards (or paper tape, etc.) before submitting them to a compiler. Dartmouth BASIC was the first language to be created with an IDE (and was also the first to be designed for use while sitting in front of a console or terminal). Its IDE (part of the Dartmouth Time Sharing System) was command-based, and therefore did not look much like the menu-driven, graphical IDEs prevalent today. However it integrated editing, file management, compilation, debugging and execution in a manner consistent with a modern IDE.



Keyboard Maestro

Maestro I is a product from Softlab Munich and was the world's first integrated development environment 1975 for software. Maestro I was installed for 22,000 programmers worldwide. Until 1989, 6,000 installations existed in the Federal Republic of Germany. Maestro I was arguably the world leader in this field during the 1970s and 1980s. Today one of the last Maestro I can be found in the Museum of Information Technology at Arlington.

One of the first IDEs with a plug-in concept was Softbench. In 1995 Computerwoche commented that the use of an IDE was not well received by developers since it would fence in their creativity.

Topics

Visual programming

Visual programming is a usage scenario in which an IDE is generally required. Visual IDEs allow users to create new applications by moving programming building blocks or code nodes to create flowcharts or structure diagrams which are then compiled or interpreted. These flowcharts often are based on the Unified Modeling Language.

This interface has been popularized with the Lego Mindstorms system, and is being actively pursued by a number of companies wishing to capitalize on the power of custom

browsers like those found at Mozilla. KTechlab supports flowcode and is a popular opensource IDE and Simulator for developing software for microcontrollers. Visual programming is also responsible for the power of distributed programming (cf. LabVIEW and EICASLAB software). An early visual programming system, Max, was modelled after analog synthesizer design and has been used to develop real-time music performance software since the 1980s. Another early example was Prograph, a dataflow-based system originally developed for the Macintosh. The graphical programming environment "Grape" is used to program qfix robot kits.

This approach is also used in specialist software such as Openlab, where the end users want the flexibility of a full programming language, without the traditional learning curve associated with one.

An open source visual programming system is Mindscript, which has extended functionality for cryptology, database interfacing,

Language support

Some IDEs support multiple languages, such as Eclipse or NetBeans, both based on Java, or MonoDevelop, based on C#.

Support for alternative languages is often provided by plugins, allowing them to be installed on the same IDE at the same time. For example, Eclipse and Netbeans have plugins for C/C++, Ada, Perl, Python, Ruby, and PHP, among other languages.

Attitudes across different computing platforms

Many Unix programmers argue that traditional command-line POSIX tools constitute an IDE, though one with a different style of interface and under the Unix environment. Many programmers still use makefiles and their derivatives. Also, many Unix programmers use Emacs or Vim, which integrate support for many of the standard Unix build tools. Data Display Debugger is intended to be an advanced graphical front-end for many text-based debugger standard tools.

On the various Microsoft Windows platforms, command-line tools for development are seldom used. Accordingly, there are many commercial and non-commercial solutions, however each has a different design commonly creating incompatibilities. Most major compiler vendors for Windows still provide free copies of their command-line tools, including Microsoft (Visual C++, Platform SDK, Microsoft .NET Framework SDK, nmake utility), Embarcadero Technologies (bcc32 compiler, make utility).

Additionally, the free software GNU tools (gcc, gdb, GNU make) are available on many platforms, including Windows etc.

IDEs have always been popular on the Apple Macintosh's Mac OS, dating back to Macintosh Programmer's Workshop, Turbo Pascal, THINK Pascal and THINK C

environments in the mid-1980s. Currently Mac OS X programmers can choose between limited IDEs, including native IDEs like Xcode, older IDEs like CodeWarrior, and open-source tools, such as Eclipse and Netbeans. ActiveState Komodo is a proprietary IDE supported on the Mac OS.

Some open-source IDEs such as Code::Blocks, Eclipse, Lazarus, KDevelop and Netbeans, which themselves are developed using a cross-platform language (e.g., Free Pascal or Java), run on multiple platforms including Windows, GNU/Linux, and Mac OS.

Chapter 7

Code Coverage and Compiler

Code coverage

Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested. It is a form of testing that inspects the code directly and is therefore a form of white box testing. In time, the use of code coverage has been extended to the field of digital hardware, the contemporary design methodology of which relies on hardware description languages (HDLs).

Code coverage was among the first methods invented for systematic software testing. The first published reference was by Miller and Maloney in *Communications of the ACM* in 1963.

Code coverage is one consideration in the safety certification of avionics equipment. The standard by which avionics gear is certified by the Federal Aviation Administration (FAA) is documented in DO-178B.

Coverage criteria

To measure how well the program is exercised by a test suite, one or more *coverage criteria* are used.

Basic coverage criteria

There are a number of coverage criteria, the main ones being:

- **Function coverage** - Has each function (or subroutine) in the program been called?
- **Statement coverage** - Has each node in the program been executed?
- **Decision coverage** or **branch coverage** - Has every edge in the program been executed? For instance, have the requirements of each branch of each control structure (such as in IF and CASE statements) been met as well as not met?

- **Condition coverage** (or predicate coverage) - Has each boolean sub-expression evaluated both to true and false? This does not necessarily imply decision coverage.
- **Condition/decision coverage** - Both decision and condition coverage should be satisfied.

For example, consider the following C++ function:

```
int foo(int x, int y)
{
    int z = 0;
    if ((x>0) && (y>0)) {
        z = x;
    }
    return z;
}
```

Assume this function is a part of some bigger program and this program was run with some test suite.

- If during this execution function 'foo' was called at least once, then *function coverage* for this function is satisfied.
- *Statement coverage* for this function will be satisfied if it was called e.g. as `foo(1,1)`, as in this case, every line in the function is executed including `z = x;`.
- Tests calling `foo(1,1)` and `foo(1,0)` will satisfy *decision coverage*, as in the first case the `if` condition is satisfied and `z = x;` is executed, and in the second it is not.
- *Condition coverage* can be satisfied with tests that call `foo(1,1)`, `foo(1,0)` and `foo(0,0)`. These are necessary as in the first two cases `(x>0)` evaluates to `true` while in the third it evaluates `false`. At the same time, the first case makes `(y>0)` `true` while the second and third make it `false`.

In languages, like Pascal, where standard boolean operations are not short circuited, condition coverage does not necessarily imply decision coverage. For example, consider the following fragment of code:

```
if a and b then
```

Condition coverage can be satisfied by two tests:

- `a=true`, `b=false`
- `a=false`, `b=true`

However, this set of tests does not satisfy decision coverage as in neither case will the `if` condition be met.

Fault injection may be necessary to ensure that all conditions and branches of exception handling code have adequate coverage during testing.

Modified condition/decision coverage

For safety-critical applications (e.g. for avionics software) it is often required that **modified condition/decision coverage (MC/DC)** is satisifed. This criteria extends condition/decision criteria with requirements that each condition should affect the decision outcome independently. For example, consider the following code:

```
if (a or b) and c then
```

The condition/decision criteria will be satisfied by the following set of tests:

- a=true, b=true, c=true
- a=false, b=false, c=false

However, the above tests set will not satisfy modified condition/decision coverage, since in the first test, the value of 'b' and in the second test the value of 'c' would not influence the output. So, the following test set is needed to satisfy MC/DC:

- a=false, b=true, c=true
- a=true, b=false, c=true
- a=true, b=true, c=false

Multiple condition coverage

This criteria requires that all combinations of conditions inside each decision are tested. For example, the code fragment from the previous section will require eight tests:

- a=false, b=false, c=false
- a=false, b=false, c=true
- a=false, b=true, c=false
- a=false, b=true, c=true
- a=true, b=false, c=false
- a=true, b=false, c=true
- a=true, b=true, c=false
- a=true, b=true, c=true

Other coverage criteria

There are further coverage criteria, which are used less often:

- **Linear Code Sequence and Jump (LCSAJ) coverage** - has every LCSAJ been executed?
- **JJ-Path coverage** - have all jump to jump paths (aka LCSAJs) been executed?
- **Path coverage** - Has every possible route through a given part of the code been executed?

- **Entry/exit coverage** - Has every possible call and return of the function been executed?
- **Loop coverage** - Has every possible loop been executed zero times, once, and more than once?

Safety-critical applications are often required to demonstrate that testing achieves 100% of some form of code coverage.

Some of the coverage criteria above are connected. For instance, path coverage implies decision, statement and entry/exit coverage. Decision coverage implies statement coverage, because every statement is part of a branch.

Full path coverage, of the type described above, is usually impractical or impossible. Any module with a succession of n decisions in it can have up to 2^n paths within it; loop constructs can result in an infinite number of paths. Many paths may also be infeasible, in that there is no input to the program under test that can cause that particular path to be executed. However, a general-purpose algorithm for identifying infeasible paths has been proven to be impossible (such an algorithm could be used to solve the halting problem). Methods for practical path coverage testing instead attempt to identify classes of code paths that differ only in the number of loop executions, and to achieve "basis path" coverage the tester must cover all the path classes.

In practice

The target software is built with special options or libraries and/or run under a special environment such that every function that is exercised (executed) in the program(s) is mapped back to the function points in the source code. This process allows developers and quality assurance personnel to look for parts of a system that are rarely or never accessed under normal conditions (error handling and the like) and helps reassure test engineers that the most important conditions (function points) have been tested. The resulting output is then analyzed to see what areas of code have not been exercised and the tests are updated to include these areas as necessary. Combined with other code coverage methods, the aim is to develop a rigorous, yet manageable, set of regression tests.

In implementing code coverage policies within a software development environment one must consider the following:

- What are coverage requirements for the end product certification and if so what level of code coverage is required? The typical level of rigor progression is as follows: Statement, Branch/Decision, Modified Condition/Decision Coverage(MC/DC), LCSAJ (Linear Code Sequence and Jump)
- Will code coverage be measured against tests that verify requirements levied on the system under test (DO-178B)?
- Is the object code generated directly traceable to source code statements? Certain certifications, (ie. DO-178B Level A) require coverage at the assembly level if

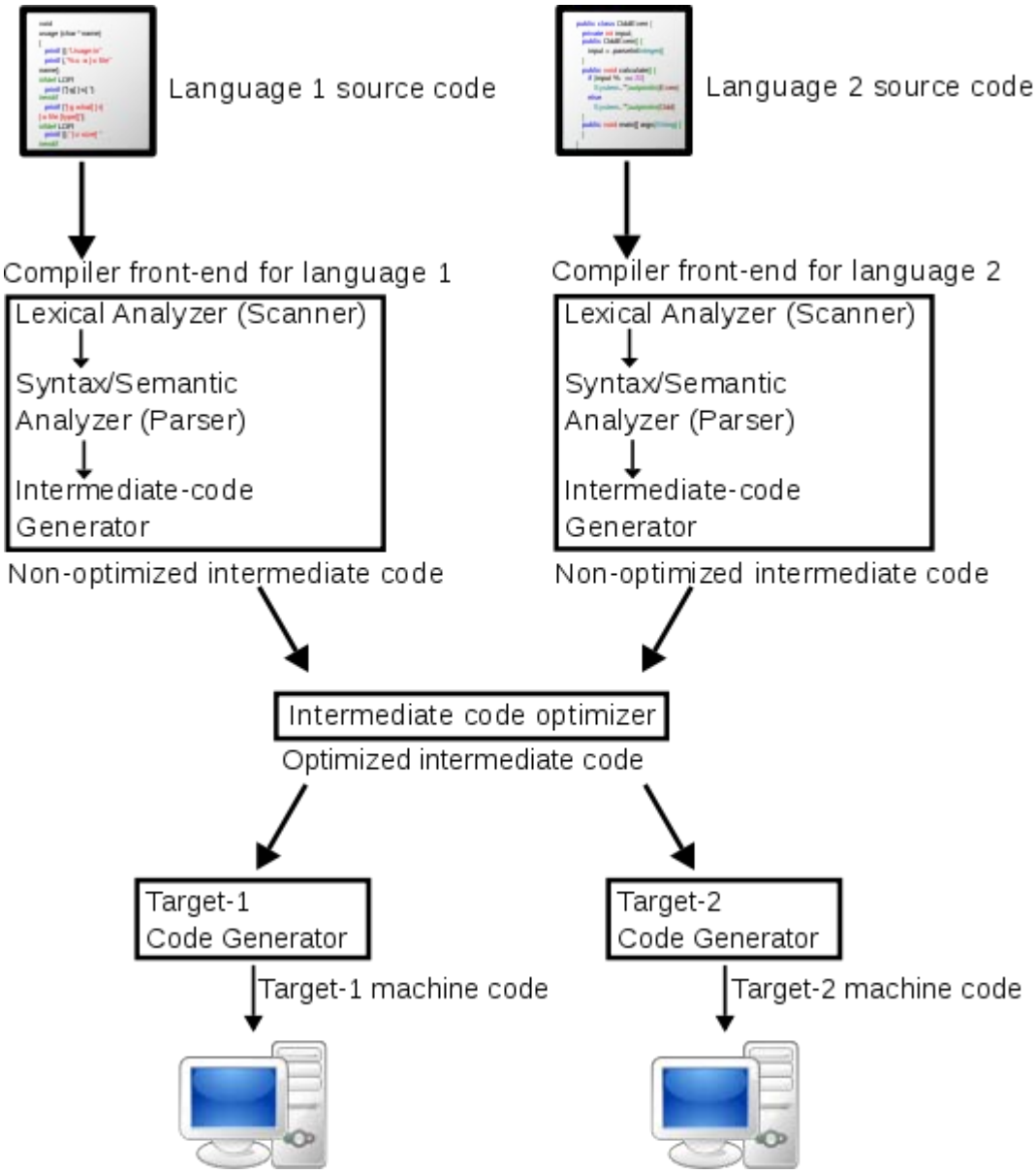
this is not the case: "Then, additional verification should be performed on the object code to establish the correctness of such generated code sequences" (DO-178B) para-6.4.4.2.

Test engineers can look at code coverage test results to help them devise test cases and input or configuration sets that will increase the code coverage over vital functions. Two common forms of code coverage used by testers are statement (or line) coverage and path (or edge) coverage. Line coverage reports on the execution footprint of testing in terms of which lines of code were executed to complete the test. Edge coverage reports which branches or code decision points were executed to complete the test. They both report a coverage metric, measured as a percentage. The meaning of this depends on what form(s) of code coverage have been used, as 67% path coverage is more comprehensive than 67% statement coverage.

Generally, code coverage tools and libraries exact a performance and/or memory or other resource cost which is unacceptable to normal operations of the software. Thus, they are only used in the lab. As one might expect, there are classes of software that cannot be feasibly subjected to these coverage tests, though a degree of coverage mapping can be approximated through analysis rather than direct testing.

There are also some sorts of defects which are affected by such tools. In particular, some race conditions or similar real time sensitive operations can be masked when run under code coverage environments; and conversely, some of these defects may become easier to find as a result of the additional overhead of the testing code.

Compiler



A diagram of the operation of a typical multi-language, multi-target compiler

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code*). The most common reason for wanting to transform source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or

machine code). If the compiled program can only run on a computer whose CPU or operating system is different from the one on which the compiler runs the compiler is known as a cross-compiler. A program that translates from a low level language to a higher level one is a *decompiler*. A program that translates between high-level languages is usually called a *language translator*, *source to source translator*, or *language converter*. A *language rewriter* is usually a program that translates the form of expressions without a change of language.

A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization.

Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementors invest a lot of time ensuring the correctness of their software.

The term compiler-compiler is sometimes used to refer to a parser generator, a tool often used to help create the lexer and parser.

History

Software for early computers was primarily written in assembly language for many years. Higher level programming languages were not invented until the benefits of being able to reuse software on different kinds of CPUs started to become significantly greater than the cost of writing a compiler. The very limited memory capacity of early computers also created many technical problems when implementing a compiler.

Towards the end of the 1950s, machine-independent programming languages were first proposed. Subsequently, several experimental compilers were developed. The first compiler was written by Grace Hopper, in 1952, for the A-0 programming language. The FORTRAN team led by John Backus at IBM is generally credited as having introduced the first complete compiler in 1957. COBOL was an early language to be compiled on multiple architectures, in 1960.

In many application domains the idea of using a higher level language quickly caught on. Because of the expanding functionality supported by newer programming languages and the increasing complexity of computer architectures, compilers have become more and more complex.

Early compilers were written in assembly language. The first *self-hosting* compiler — capable of compiling its own source code in a high-level language — was created for Lisp by Tim Hart and Mike Levin at MIT in 1962. Since the 1970s it has become common practice to implement a compiler in the language it compiles, although both Pascal and C have been popular choices for implementation language. Building a self-hosting compiler is a bootstrapping problem—the first such compiler for a language must

be compiled either by a compiler written in a different language, or (as in Hart and Levin's Lisp compiler) compiled by running the compiler in an interpreter.

Compilers in education

Compiler construction and compiler optimization are taught at universities and schools as part of the computer science curriculum. Such courses are usually supplemented with the implementation of a compiler for an educational programming language. A well-documented example is Niklaus Wirth's PL/0 compiler, which Wirth used to teach compiler construction in the 1970s. In spite of its simplicity, the PL/0 compiler introduced several influential concepts to the field:

1. Program development by stepwise refinement (also the title of a 1971 paper by Wirth)
2. The use of a recursive descent parser
3. The use of EBNF to specify the syntax of a language
4. A code generator producing portable P-code
5. The use of T-diagrams in the formal description of the bootstrapping problem

Compilation

Compilers enabled the development of programs that are machine-independent. Before the development of FORTRAN (FORmula TRANslator), the first higher-level language, in the 1950s, machine-dependent assembly language was widely used. While assembly language produces more reusable and relocatable programs than machine code on the same architecture, it has to be modified or rewritten if the program is to be executed on different hardware architecture.

With the advance of high-level programming languages soon followed after FORTRAN, such as COBOL, C, BASIC, programmers can write machine-independent source programs. A compiler translates the high-level source programs into target programs in machine languages for the specific hardwares. Once the target program is generated, the user can execute the program.

Structure of compiler

Compilers bridge source programs in high-level languages with the underlying hardwares. A compiler requires 1) to recognize legitimacy of programs, 2) to generate correct and efficient code, 3) run-time organization, 4) to format output according to assembler or linker conventions. A compiler consists of three main parts: frontend, middle-end, and backend.

Frontend checks whether the program is correctly written in terms of the programming language syntax and semantics. Here legal and illegal programs are recognized. Errors are reported, if any, in a useful way. Type checking is also performed by collecting type information. Frontend generates IR (intermediate representation) for the middle-end.

Optimization of this part is almost complete so much are already automated. There are efficient algorithms typically in $O(n)$ or $O(n \log n)$.

Middle-end is where the optimizations for performance take place. Typical transformations for optimization are removal of useless or unreachable code, discovering and propagating constant values, relocation of computation to a less frequently executed place (e.g., out of a loop), or specializing a computation based on the context. Middle-end generates IR for the following backend. Most optimization efforts are focused on this part.

Backend is responsible for translation of IR into the target assembly code. The target instruction(s) are chosen for each IR instruction. Variables are also selected for the registers. Backend utilizes the hardware by figuring out how to keep parallel FUs busy, filling delay slots, and so on. Although most algorithms for optimization are in NP, heuristic techniques are well-developed.

Compiler output

One classification of compilers is by the platform on which their generated code executes. This is known as the *target platform.*

A *native* or *hosted* compiler is one whose output is intended to directly run on the same type of computer and operating system that the compiler itself runs on. The output of a cross compiler is designed to run on a different platform. Cross compilers are often used when developing software for embedded systems that are not intended to support a software development environment.

The output of a compiler that produces code for a virtual machine (VM) may or may not be executed on the same platform as the compiler that produced it. For this reason such compilers are not usually classified as native or cross compilers.

Compiled versus interpreted languages

Higher-level programming languages are generally divided for convenience into compiled languages and interpreted languages. However, in practice there is rarely anything about a language that *requires* it to be exclusively compiled or exclusively interpreted, although it is possible to design languages that rely on re-interpretation at run time. The categorization usually reflects the most popular or widespread implementations of a language — for instance, BASIC is sometimes called an interpreted language, and C a compiled one, despite the existence of BASIC compilers and C interpreters.

Modern trends toward just-in-time compilation and bytecode interpretation at times blur the traditional categorizations of compilers and interpreters.

Some language specifications spell out that implementations *must* include a compilation facility; for example, Common Lisp. However, there is nothing inherent in the definition

of Common Lisp that stops it from being interpreted. Other languages have features that are very easy to implement in an interpreter, but make writing a compiler much harder; for example, APL, SNOBOL4, and many scripting languages allow programs to construct arbitrary source code at runtime with regular string operations, and then execute that code by passing it to a special evaluation function. To implement these features in a compiled language, programs must usually be shipped with a runtime library that includes a version of the compiler itself.

Hardware compilation

The output of some compilers may target hardware at a very low level, for example a Field Programmable Gate Array (FPGA) or structured Application-specific integrated circuit (ASIC). Such compilers are said to be *hardware compilers* or synthesis tools because the source code they compile effectively control the final configuration of the hardware and how it operates; the output of the compilation are not instructions that are executed in sequence - only an interconnection of transistors or lookup tables. For example, XST is the Xilinx Synthesis Tool used for configuring FPGAs. Similar tools are available from Altera, Synplicity, Synopsys and other vendors.

Compiler design

In the early days, the approach taken to compiler design used to be directly affected by the complexity of the processing, the experience of the person(s) designing it, and the resources available.

A compiler for a relatively simple language written by one person might be a single, monolithic piece of software. When the source language is large and complex, and high quality output is required the design may be split into a number of relatively independent phases. Having separate phases means development can be parceled up into small parts and given to different people. It also becomes much easier to replace a single phase by an improved one, or to insert new phases later (e.g., additional optimizations).

The division of the compilation processes into phases was championed by the Production Quality Compiler-Compiler Project (PQCC) at Carnegie Mellon University. This project introduced the terms *front end*, *middle end*, and *back end*.

All but the smallest of compilers have more than two phases. However, these phases are usually regarded as being part of the front end or the back end. The point at which these two *ends* meet is open to debate. The front end is generally considered to be where syntactic and semantic processing takes place, along with translation to a lower level of representation (than source code).

The middle end is usually designed to perform optimizations on a form other than the source code or machine code. This source code/machine code independence is intended to enable generic optimizations to be shared between versions of the compiler supporting different languages and target processors.

The back end takes the output from the middle. It may perform more analysis, transformations and optimizations that are for a particular computer. Then, it generates code for a particular processor and OS.

This front-end/middle/back-end approach makes it possible to combine front ends for different languages with back ends for different CPUs. Practical examples of this approach are the GNU Compiler Collection, LLVM, and the Amsterdam Compiler Kit, which have multiple front-ends, shared analysis and multiple back-ends.

One-pass versus multi-pass compilers

Classifying compilers by number of passes has its background in the hardware resource limitations of computers. Compiling involves performing lots of work and early computers did not have enough memory to contain one program that did all of this work. So compilers were split up into smaller programs which each made a pass over the source (or some representation of it) performing some of the required analysis and translations.

The ability to compile in a single pass has classically been seen as a benefit because it simplifies the job of writing a compiler and one pass compilers generally compile faster than multi-pass compilers. Thus, partly driven by the resource limitations of early systems, many early languages were specifically designed so that they could be compiled in a single pass (e.g., Pascal).

In some cases the design of a language feature may require a compiler to perform more than one pass over the source. For instance, consider a declaration appearing on line 20 of the source which affects the translation of a statement appearing on line 10. In this case, the first pass needs to gather information about declarations appearing after statements that they affect, with the actual translation happening during a subsequent pass.

The disadvantage of compiling in a single pass is that it is not possible to perform many of the sophisticated optimizations needed to generate high quality code. It can be difficult to count exactly how many passes an optimizing compiler makes. For instance, different phases of optimization may analyse one expression many times but only analyse another expression once.

Splitting a compiler up into small programs is a technique used by researchers interested in producing provably correct compilers. Proving the correctness of a set of small programs often requires less effort than proving the correctness of a larger, single, equivalent program.

While the typical multi-pass compiler outputs machine code from its final pass, there are several other types:

- A "source-to-source compiler" is a type of compiler that takes a high level language as its input and outputs a high level language. For example, an automatic parallelizing compiler will frequently take in a high level language

program as an input and then transform the code and annotate it with parallel code annotations (e.g. OpenMP) or language constructs (e.g. Fortran's `DOALL` statements).
- Stage compiler that compiles to assembly language of a theoretical machine, like some Prolog implementations
    - This Prolog machine is also known as the Warren Abstract Machine (or WAM).
    - Bytecode compilers for Java, Python, and many more are also a subtype of this.
- Just-in-time compiler, used by Smalltalk and Java systems, and also by Microsoft .NET's Common Intermediate Language (CIL)
    - Applications are delivered in bytecode, which is compiled to native machine code just prior to execution.

Front end

The front end analyzes the source code to build an internal representation of the program, called the intermediate representation or *IR*. It also manages the symbol table, a data structure mapping each symbol in the source code to associated information such as location, type and scope. This is done over several phases, which includes some of the following:

1. **Line reconstruction**. Languages which strop their keywords or allow arbitrary spaces within identifiers require a phase before parsing, which converts the input character sequence to a canonical form ready for the parser. The top-down, recursive-descent, table-driven parsers used in the 1960s typically read the source one character at a time and did not require a separate tokenizing phase. Atlas Autocode, and Imp (and some implementations of Algol and Coral66) are examples of stropped languages whose compilers would have a *Line Reconstruction* phase.
2. Lexical analysis breaks the source code text into small pieces called *tokens*. Each token is a single atomic unit of the language, for instance a keyword, identifier or symbol name. The token syntax is typically a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it. This phase is also called lexing or scanning, and the software doing lexical analysis is called a lexical analyzer or scanner.
3. Preprocessing. Some languages, e.g., C, require a preprocessing phase which supports macro substitution and conditional compilation. Typically the preprocessing phase occurs before syntactic or semantic analysis; e.g. in the case of C, the preprocessor manipulates lexical tokens rather than syntactic forms. However, some languages such as Scheme support macro substitutions based on syntactic forms.
4. Syntax analysis involves parsing the token sequence to identify the syntactic structure of the program. This phase typically builds a parse tree, which replaces the linear sequence of tokens with a tree structure built according to the rules of a

formal grammar which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.

5. Semantic analysis is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

Back end

The term *back end* is sometimes confused with *code generator* because of the overlapped functionality of generating assembly code. Some literature uses *middle end* to distinguish the generic analysis and optimization phases in the back end from the machine-dependent code generators.

The main phases of the back end include the following:

1. Analysis: This is the gathering of program information from the intermediate representation derived from the input. Typical analyses are data flow analysis to build use-define chains, dependence analysis, alias analysis, pointer analysis, escape analysis etc. Accurate analysis is the basis for any compiler optimization. The call graph and control flow graph are usually also built during the analysis phase.
2. Optimization: the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are inline expansion, dead code elimination, constant propagation, loop transformation, register allocation and even automatic parallelization.
3. Code generation: the transformed intermediate language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated addressing modes.

Compiler analysis is the prerequisite for any compiler optimization, and they tightly work together. For example, dependence analysis is crucial for loop transformation.

In addition, the scope of compiler analysis and optimizations vary greatly, from as small as a basic block to the procedure/function level, or even over the whole program (interprocedural optimization). Obviously, a compiler can potentially do a better job using a broader view. But that broad view is not free: large scope analysis and optimizations are very costly in terms of compilation time and memory space; this is especially true for interprocedural analysis and optimizations.

Interprocedural analysis and optimizations are common in modern commercial compilers from HP, IBM, SGI, Intel, Microsoft, and Sun Microsystems. The open source GCC was criticized for a long time for lacking powerful interprocedural optimizations, but it is changing in this respect. Another open source compiler with full analysis and optimization infrastructure is Open64, which is used by many organizations for research and commercial purposes.

Due to the extra time and space needed for compiler analysis and optimizations, some compilers skip them by default. Users have to use compilation options to explicitly tell the compiler which optimizations should be enabled.

Compiler correctness

Compiler correctness is the branch of software engineering that deals with trying to show that a compiler behaves according to its language specification. Techniques include developing the compiler using formal methods and using rigorous testing (often called compiler validation) on an existing compiler.

Related techniques

Assembly language is a type of low-level language and a program that compiles it is more commonly known as an *assembler*, with the inverse program known as a *disassembler*.

A program that translates from a low level language to a higher level one is a *decompiler*.

A program that translates between high-level languages is usually called a *language translator*, *source to source translator*, *language converter*, or *language rewriter*. The last term is usually applied to translations that do not involve a change of language.

International conferences and organizations

Every year, the **European Joint Conferences on Theory and Practice of Software** (ETAPS) sponsors the **International Conference on Compiler Construction** (CC), with papers from both the academic and industrial sectors.

Chapter 8

Programming Style and Text Editor

Programming style

Programming style is a set of rules or guidelines used when writing the source code for a computer program. It is often claimed that following a particular programming style will help programmers to read and understand source code conforming to the style, and help to avoid introducing errors.

A classic work on the subject was *The Elements of Programming Style*, written in the 1970s, and illustrated with examples from the Fortran and PL/I languages prevalent at the time.

The programming style used in a particular program may be derived from the Coding conventions of a company or other computing organization, as well as the preferences of the author of the code. Programming styles are often designed for a specific programming language (or language family): style considered good in C source code may not be appropriate for BASIC source code, and so on. However, some rules are commonly applied to many languages.

Elements of good style

Good style is a subjective matter, and is difficult to define. However, there are several elements common to a large number of programming styles. The issues usually considered as part of programming style include the layout of the source code, including indentation; the use of white space around operators and keywords; the capitalization or otherwise of keywords and variable names; the style and spelling of user-defined identifiers, such as function, procedure and variable names; and the use and style of comments.

Code appearance

Programming styles commonly deal with the visual appearance of source code, with the goal of requiring less human cognitive effort to extract information about the program. Software has long been available that formats source code automatically, leaving coders

to concentrate on naming, logic, and higher techniques. As a practical point, using a computer to format source code saves time, and it is possible to then enforce company-wide standards without debates.

Indentation

Indent styles assist in identifying control flow and blocks of code. In some programming languages indentation is used to delimit logical blocks of code; correct indentation in these cases is more than a matter of style. In other languages indentation and whitespace do not affect function, although logical and consistent indentation makes code more readable. Compare:

```
if (hours < 24 && minutes < 60 && seconds < 60) {
    return true;
} else {
    return false;
}
```

or

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
    return true;
}
else
{
    return false;
}
```

with something like

```
if  (    hours<
24  && minutes<
60  && seconds<
60  )
{return    true
;}         else
{return   false
;}
```

The first two examples are probably much easier to read because they are indented in an established way (a "hanging paragraph" style). This indentation style is especially useful when dealing with multiple nested constructs.

ModuLiq

The ModuLiq Zero Indent Style groups with carriage returns rather than indents. Compare all of the above to:

```
if (hours < 24 && minutes < 60 && seconds < 60)
```

```
return true;

else
return false;
```

Python

Python uses indentation to indicate control structures, so correct indentation is *required*. By doing this, the need for bracketing with curly braces ({ and }) is eliminated. On the other hand copying and pasting Python code can lead to problems, because the indentation level of the pasted code may not be the same as the indentation level of the current line. Such reformatting is tedious to do by hand, but some text editors and IDEs have features to do it automatically. There are also problems when Python code could be rendered unusable when posted on a forum or web page that removes whitespace, though this problem can be avoided where it is possible to enclose code in whitespace-preserving tags such as "<pre> ... </pre>" (for HTML), "[code]" ... "[/code]" (for bbcode), etc.

Haskell

Haskell similarly has the off-side rule which lets indentation define blocks; however, unlike in Python, indentation is not compulsory in Haskell — curly braces and semicolons can be (and occasionally are) used instead.

Vertical alignment

It is often helpful to align similar elements vertically, to make typo-generated bugs more obvious. Compare:

```
$search = array('a', 'b', 'c', 'd', 'e');
$replacement = array('foo', 'bar', 'baz', 'quux');

// Another example:

$value = 0;
$anothervalue = 1;
$yetanothervalue = 2;
```

with:

```
$search      = array('a',   'b',   'c',   'd',   'e');
$replacement = array('foo', 'bar', 'baz', 'quux');

// Another example:

          $value = 0;
   $anothervalue = 1;
$yetanothervalue = 2;
```

The latter example makes two things intuitively clear that were not clear in the former:

- the search and replace terms are related and match up: they are not discrete variables;
- there is one more search term than there are replacement terms. If this is a bug, it is now more likely to be spotted.

Arguments against vertical alignment are

- **Inter-line false dependencies**; tabular formatting creates dependencies across lines. For example, if an identifier with a long name is added to a tabular layout, the column width may have to be increased to accommodate it. This forces a bigger change to the source code than necessary, and the essential change may be lost in the noise. This is detrimental to source code control where inspecting differences between versions is essential.

- **Brittleness**; if a programmer does not neatly format the table when making a change; maybe legitimately with the previous point in mind; the result becomes a mess that deteriorates with further such changes.

- **Resistance to change**; tabular formatting requires more effort to maintain. This may put off a programmer from making a beneficial change, such as adding, correcting or improving the name of an identifier, because it will mess up the formatting.

- **Reliance on mono-spaced font**; tabular formatting assumes that the editor uses a fixed-width font. Most modern code editors support proportional fonts, and the programmer may prefer to use a proportional font for readability.

- **Tool dependence**; some of the effort of maintaining alignment can be alleviated by tools (e.g. a source code editor that supports elastic tabstops), although that creates a reliance on tools.

Spaces

In those situations where some whitespace is required the grammars of most free-format languages are unconcerned with the amount that appears. Style related to whitespace is commonly used to enhance readability. There are currently no known hard facts (conclusions from studies) about which of the whitespace styles are having the best readability.

For instance, compare the following syntactically equivalent examples of C code.

```
int i;
for(i=0;i<10;++i){
    printf("%d",i*i+i);
}
```

versus

```
int i;
for (i=0; i<10; ++i) {
    printf("%d", i*i+i);
}
```

versus

```
int i;
for (i = 0; i < 10; ++i) {
    printf ("%d", i * i + i);
}
```

versus

```
int i;
for( i = 0; i < 10; ++i ) {
    printf( "%d", i * i + i );
}
```

Tabs

The use of tabs to create white space presents particular issues when not enough care is taken because the location of the tabulation point can be different depending on the tools being used and even the preferences of the user.

As an example, one programmer prefers tab stops of four and has his toolset configured this way, and uses these to format his code.

```
int     ix;     // Index to scan array
long    sum;    // Accumulator for sum
```

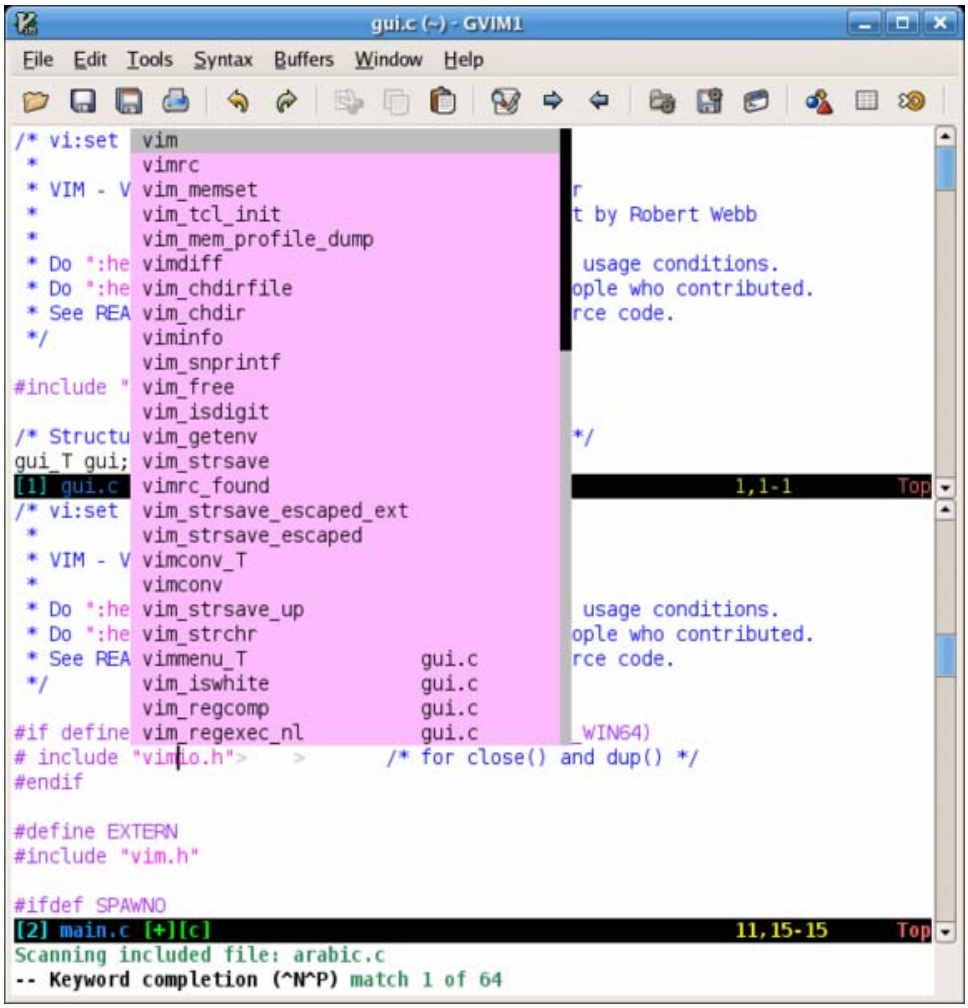
Another programmer prefers tab stops of eight, and his toolset is configured this way. When he examines his code, he may well find it difficult to read.

```
int             ix;             // Index to scan array
long    sum;    // Accumulator for sum
```

One widely used solution to this issue may involve forbidding the use of tabs for alignment or rules on how tab stops must be set. Note that tabs work fine provided they are used consistently, restricted to logical indentation, and not used for alignment:

```
class MyClass {
        int foobar(
                int qux, // first parameter
                int quux); // second parameter
        int foobar2(
                int qux, // first parameter
                int quux, // second parameter
                int quuux); // third parameter
};
```

Text editor



An example of a text editor, Vim

A **text editor** is a type of program used for editing plain text files.

Text editors are often provided with operating systems or software development packages, and can be used to change configuration files and programming language source code.

Plain text files vs. word processor files

There are important differences between plain text files created by a text editor, and document files created by word processors such as Microsoft Word, WordPerfect, or OpenOffice.org. Briefly:

- A plain text file is represented and edited by showing all the characters as they are present in the file. The only characters usable for 'mark-up' are the control characters of the used character set; in practice this is newline, tab and formfeed. The most commonly used character set is ASCII, especially recently, as plain text files are more often being used for programming and configuration, and less frequently for documentation (e.g. detailed instructions, user guides) than in the past.
- Documents created by a word processor generally contain fileformat-specific "control characters" beyond what is defined in the character set. They enable functions like bold, italic, fonts, columns, tables, etc. These and other common page formatting symbols were once associated only with desktop publishing, but are now commonplace in the simplest word processor.
- Word processor programs can usually edit a plain text file and save it back in the plain text file format. However, one must take care to tell the program that this is what is wanted. Specifying the save format is especially important in cases such as source code, HTML, and configuration and control files. If left to the program's default, the file will contain those "special characters" unique to the word processor's file format, and will not be handled correctly by the utility the files were intended for.

Types of text editors

Some text editors are small and simple, while others offer a broad and complex range of functionality. For example, Unix and Unix-like operating systems have the vi editor (or a variant), but many also include the Emacs editor. Microsoft Windows systems come with the very simple Notepad, though many people—especially programmers—prefer to use one of many other Windows text editors with more features. Under Apple Macintosh's classic Mac OS there was the native SimpleText, which was replaced under OSX by TextEdit. Some editors, such as WordStar, have dual operating modes allowing them to be either a text editor or a word processor.

Text editors geared for professional computer users place no limit on the size of the file being opened. In particular, they start quickly even when editing large files, and are capable of editing files that are too large to fit the computer's main memory. Simpler text editors often just read files into an array in RAM. On larger files this is a slow process, and very large files often do not fit.

The ability to read and write very large files is needed by many professional computer users. For example, system administrators may need to read long log files. Programmers may need to change large source code files, or examine unusually large texts, such as an entire dictionary placed in a single file.

Some text editors include specialized computer languages to customize the editor (programmable editors). For example, Emacs can be customized by programming in Lisp. These usually permit the editor to simulate the keystroke combinations and features of other editors, so that users do not have to learn the native command combinations.

Another important group of programmable editors use REXX as their scripting language. These editors permit entering both commands and REXX statements directly in the command line at the bottom of the screen (can be hidden and activated by a keystroke). These editors are usually referred to as "orthodox editors", and most representatives of this class are derivatives of XEDIT, IBM's editor for VM/CMS. Among them are THE, Kedit, SlickEdit, X2, Uni-edit, UltraEdit, and Sedit. Some vi derivatives such as Vim also support folding as well as macro languages, and have a command line at the bottom for entering commands. They can be considered another branch of the family of orthodox editors.

Many text editors for software developers include source code syntax highlighting and automatic completion to make programs easier to read and write. Programming editors often permit one to select the name of a subprogram or variable, and then jump to its definition and back. Often an auxiliary utility like ctags is used to locate the definitions.

Typical features

- String searching algorithm – search string with a replacement string. Different methods are employed, Global(ly) Search And Replace, Conditional Search and Replace, Unconditional Search and Replace.
- Cut, copy, and paste – most text editors provide methods to duplicate and move text within the file, or between files.
- Text formatting – Text editors often provide basic formatting features like line wrap, auto-indentation, bullet list formatting, comment formatting, and so on.
- Undo and redo – As with word processors, text editors will provide a way to undo and redo the last edit. Often—especially with older text editors—there is only one level of edit history remembered and successively issuing the undo command will only "toggle" the last change. Modern or more complex editors usually provide a multiple level history such that issuing the undo command repeatedly will revert the document to successively older edits. A separate redo command will cycle the edits "forward" toward the most recent changes. The number of changes remembered depends upon the editor and is often configurable by the user.
- Data transformation – Reading or merging the contents of another text file into the file currently being edited. Some text editors provide a way to insert the output of a command issued to the operating system's shell.
- Filtering – Some advanced text editors allow the editor to send all or sections of the file being edited to another utility and read the result back into the file in place of the lines being "filtered". This, for example, is useful for sorting a series of lines alphabetically or numerically, doing mathematical computations, and so on.
- Syntax highlighting – contextually highlights software code and other text that appears in an organized or predictable format.

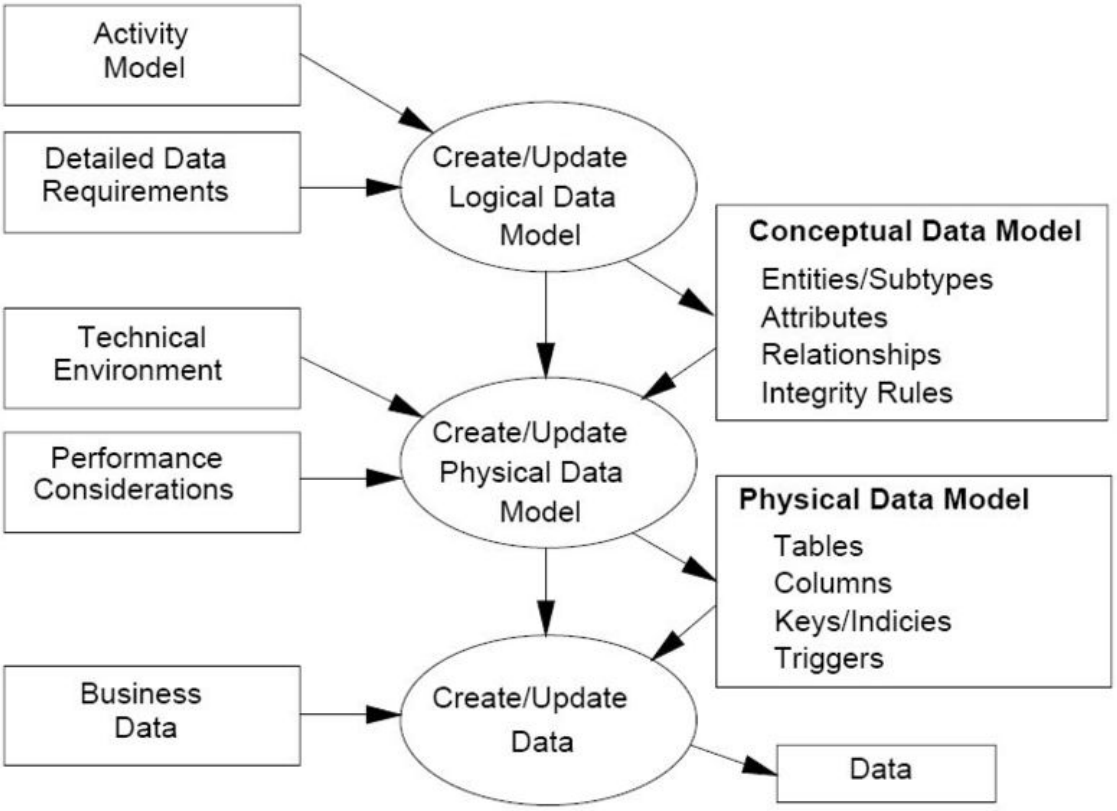
Specialised editors

Some editors include special features and extra functions, for instance,

- Source code editors are text editors with additional functionality to facilitate the production of source code. These often feature user-programmable syntax highlighting, and coding tools or keyboard macros similar to an HTML editor (see below).
- Folding editors. This subclass includes so-called "orthodox editors" that are derivatives of Xedit. The specialized version of folding is usually called outlining (see below).
- IDEs (integrated development environments) are designed to manage and streamline larger programming projects. They are usually only used for programming as they contain many features unnecessary for simple text editing.
- World Wide Web programmers are offered a variety of text editors dedicated to the task of web development. These create the plain text files that deliver web pages. HTML editors include: Dreamweaver, E (text editor), Microsoft FrontPage, HotDog, Homesite, Nvu, Tidy, and GoLive. Many offer the option of viewing a work in progress on a built-in web browser. XML editors share many traits.
- Mathematicians, physicists, and computer scientists often produce articles and books using TeX or LaTeX in plain text files. Such documents are often produced by a standard text editor, but some people use specialized TeX editors.
- Outliners. Also called tree-based editors, because they combine a hierarchical outline tree with a text editor. Folding (see above) can generally be considered a generalized form of outlining.
- Simultaneous editing is a technique in End-user development research to edit all items in a multiple selection. It allows the user to manipulate all the selected items at once through direct manipulation. The Lapis text editor and the *multi edit* plugin for gedit are examples of this technique. The Lapis editor can also create an automatic multiple selection based on an example item.

Chapter 9

Data Modeling



The data modeling process. The figure illustrates the way data models are developed and used today. A conceptual data model is developed based on the data requirements for the application that is being developed, perhaps in the context of an activity model. The data model will normally consist of entity types, attributes, relationships, integrity rules, and the definitions of those objects. This is then used as the start point for interface or database design.

Data modeling in software engineering is the process of creating a data model by applying formal data model descriptions using data modeling techniques.

Overview

Data modeling is a method used to define and analyze data requirements needed to support the business processes of an organization. The data requirements are recorded as a conceptual data model with associated data definitions. Actual implementation of the conceptual model is called a logical data model. To implement one conceptual data model may require multiple logical data models. Data modeling defines not just data elements, but their structures and relationships between them Data modeling techniques and methodologies are used to model data in a standard, consistent, predictable manner in order to manage it as a resource. The use of data modeling standards is strongly recommended for all projects requiring a standard means of defining and analyzing data within an organization, e.g., using data modeling:

- to manage data as a resource;
- for the integration of information systems;
- for designing databases/data warehouses (aka data repositories)

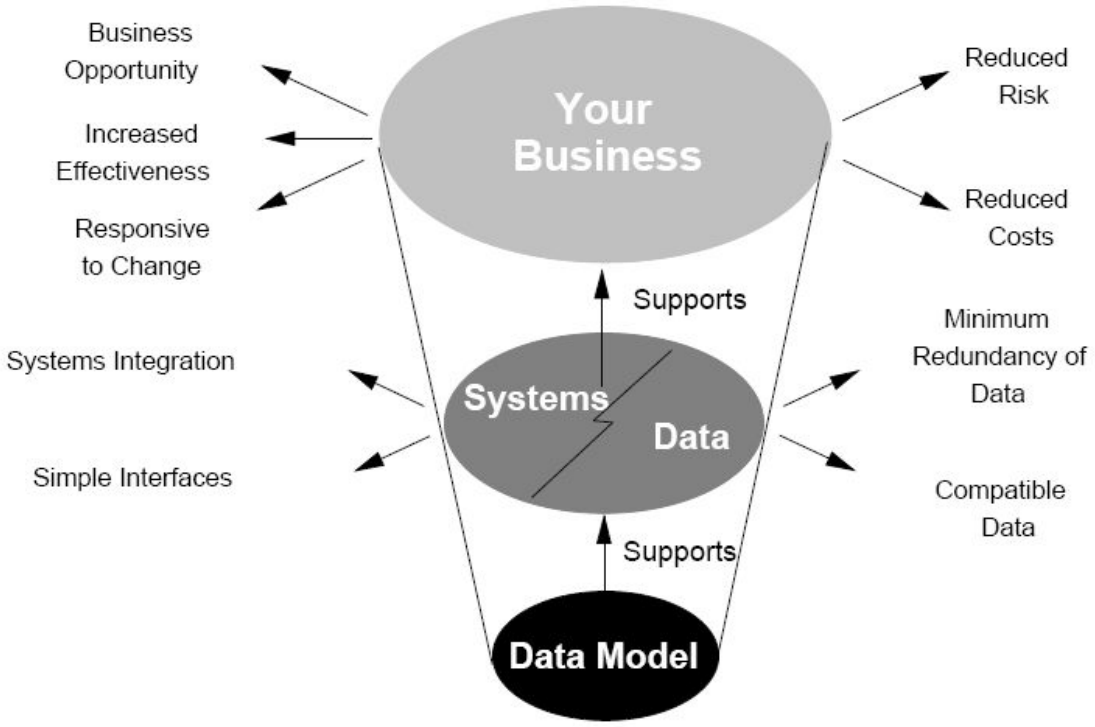
Data modeling may be performed during various types of projects and in multiple phases of projects. Data models are progressive; there is no such thing as the final data model for a business or application. Instead a data model should be considered a living document that will change in response to a changing business. The data models should ideally be stored in a repository so that they can be retrieved, expanded, and edited over time. Whitten (2004) determined two types of data modeling:

- Strategic data modeling: This is part of the creation of an information systems strategy, which defines an overall vision and architecture for information systems is defined. Information engineering is a methodology that embraces this approach.
- Data modeling during systems analysis: In systems analysis logical data models are created as part of the development of new databases.

Data modeling is also a technique for detailing business requirements for a database. It is sometimes called *database modeling* because a data model is eventually implemented in a database.

Data modeling topics

Data models



How data models deliver benefit.

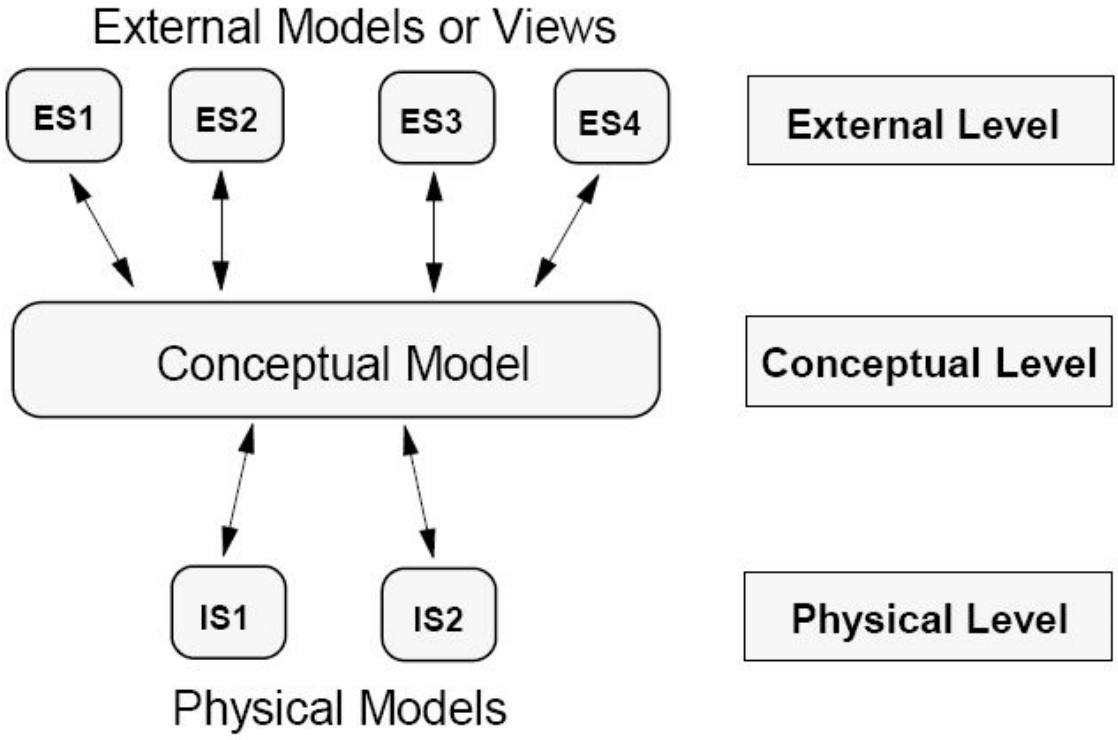
Data models support data and computer systems by providing the definition and format of data. If this is done consistently across systems then compatibility of data can be achieved. If the same data structures are used to store and access data then different applications can share data. The results of this are indicated above. However, systems and interfaces often cost more than they should, to build, operate, and maintain. They may also constrain the business rather than support it. A major cause is that the quality of the data models implemented in systems and interfaces is poor.

- Business rules, specific to how things are done in a particular place, are often fixed in the structure of a data model. This means that small changes in the way business is conducted lead to large changes in computer systems and interfaces.
- Entity types are often not identified, or incorrectly identified. This can lead to replication of data, data structure, and functionality, together with the attendant costs of that duplication in development and maintenance.
- Data models for different systems are arbitrarily different. The result of this is that complex interfaces are required between systems that share data. These interfaces can account for between 25-70% of the cost of current systems.
- Data cannot be shared electronically with customers and suppliers, because the structure and meaning of data has not been standardised. For example,

engineering design data and drawings for process plant are still sometimes exchanged on paper.

The reason for these problems is a lack of standards that will ensure that data models will both meet business needs and be consistent.

Conceptual, logical and physical schemes



The ANSI/SPARC three level architecture. This shows that a data model can be an external model (or view), a conceptual model, or a physical model. This is not the only way to look at data models, but it is a useful way, particularly when comparing models.

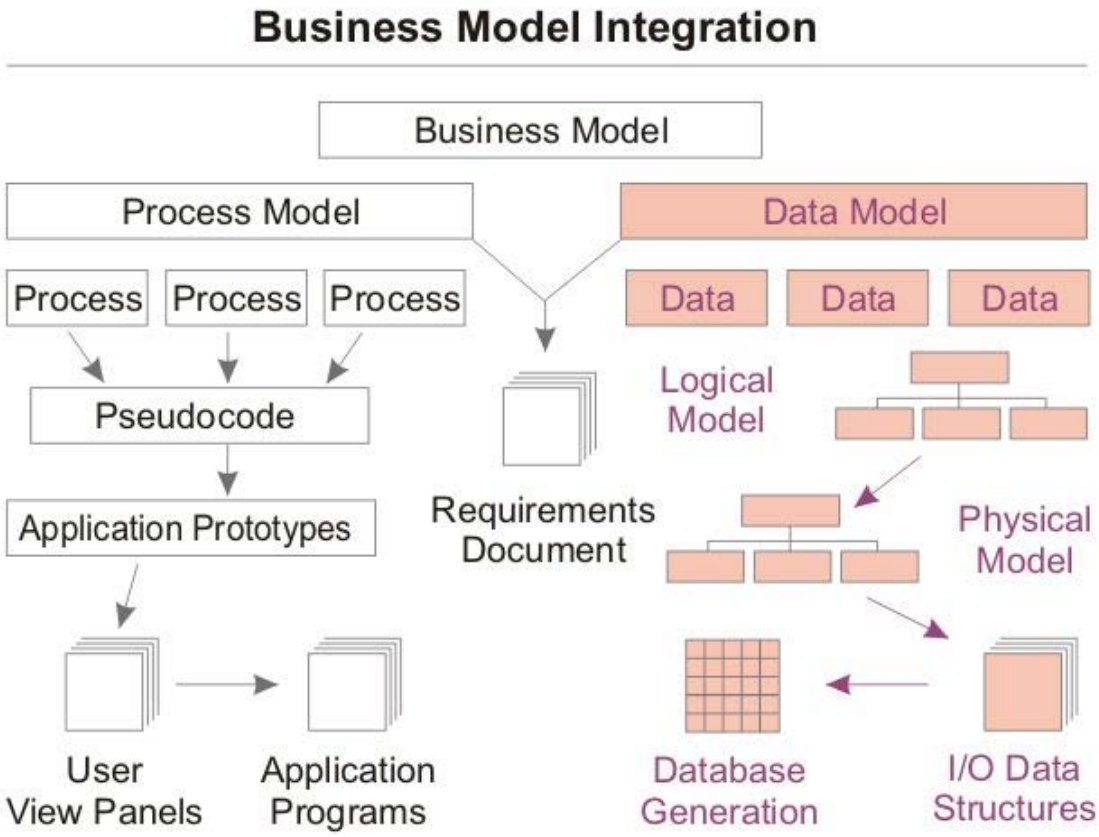
A data model *instance* may be one of three kinds according to ANSI in 1975:

- Conceptual schema: describes the semantics of a domain, being the scope of the model. For example, it may be a model of the interest area of an organization or industry. This consists of entity classes, representing kinds of things of significance in the domain, and relationships assertions about associations between pairs of entity classes. A conceptual schema specifies the kinds of facts or propositions that can be expressed using the model. In that sense, it defines the allowed expressions in an artificial 'language' with a scope that is limited by the scope of the model.
- Logical schema: describes the structure of some domain of information. This consists of descriptions of tables and columns, object oriented classes, and XML tags, among other things.

- Physical schema: describes the physical means by which data are stored. This is concerned with partitions, CPUs, tablespaces, and the like.

The significance of this approach, according to ANSI, is that it allows the three perspectives to be relatively independent of each other. Storage technology can change without affecting either the logical or the conceptual model. The table/column structure can change without (necessarily) affecting the conceptual model. In each case, of course, the structures must remain consistent with the other model. The table/column structure may be different from a direct translation of the entity classes and attributes, but it must ultimately carry out the objectives of the conceptual entity class structure. Early phases of many software development projects emphasize the design of a conceptual data model. Such a design can be detailed into a logical data model. In later stages, this model may be translated into physical data model. However, it is also possible to implement a conceptual model directly.

Data modeling process



Data modeling in the context of Business Process Integration.

In the context of Business Process Integration, see figure, data modeling will result in database generation. It complements business process modeling, which results in application programs to support the business processes.

The actual database design is the process of producing a detailed data model of a database. This logical data model contains all the needed logical and physical design choices and physical storage parameters needed to generate a design in a Data Definition Language, which can then be used to create a database. A fully attributed data model contains detailed attributes for each entity. The term database design can be used to describe many different parts of the design of an overall database system. Principally, and most correctly, it can be thought of as the logical design of the base data structures used to store the data. In the relational model these are the tables and views. In an Object database the entities and relationships map directly to object classes and named relationships. However, the term database design could also be used to apply to the overall process of designing, not just the base data structures, but also the forms and queries used as part of the overall database application within the Database Management System or DBMS.

In the process, system interfaces account for 25% to 70% of the development and support costs of current systems. The primary reason for this cost is that these systems do not share a common data model. If data models are developed on a system by system basis, then not only is the same analysis repeated in overlapping areas, but further analysis must be performed to create the interfaces between them. Most systems contain the same basic components, redeveloped for a specific purpose. For instance the following can use the same basic classification model as a component:

- Materials Catalogue,
- Product and Brand Specifications,
- Equipment specifications.

The same components are redeveloped because we have no way of telling they are the same thing.

Modeling methodologies

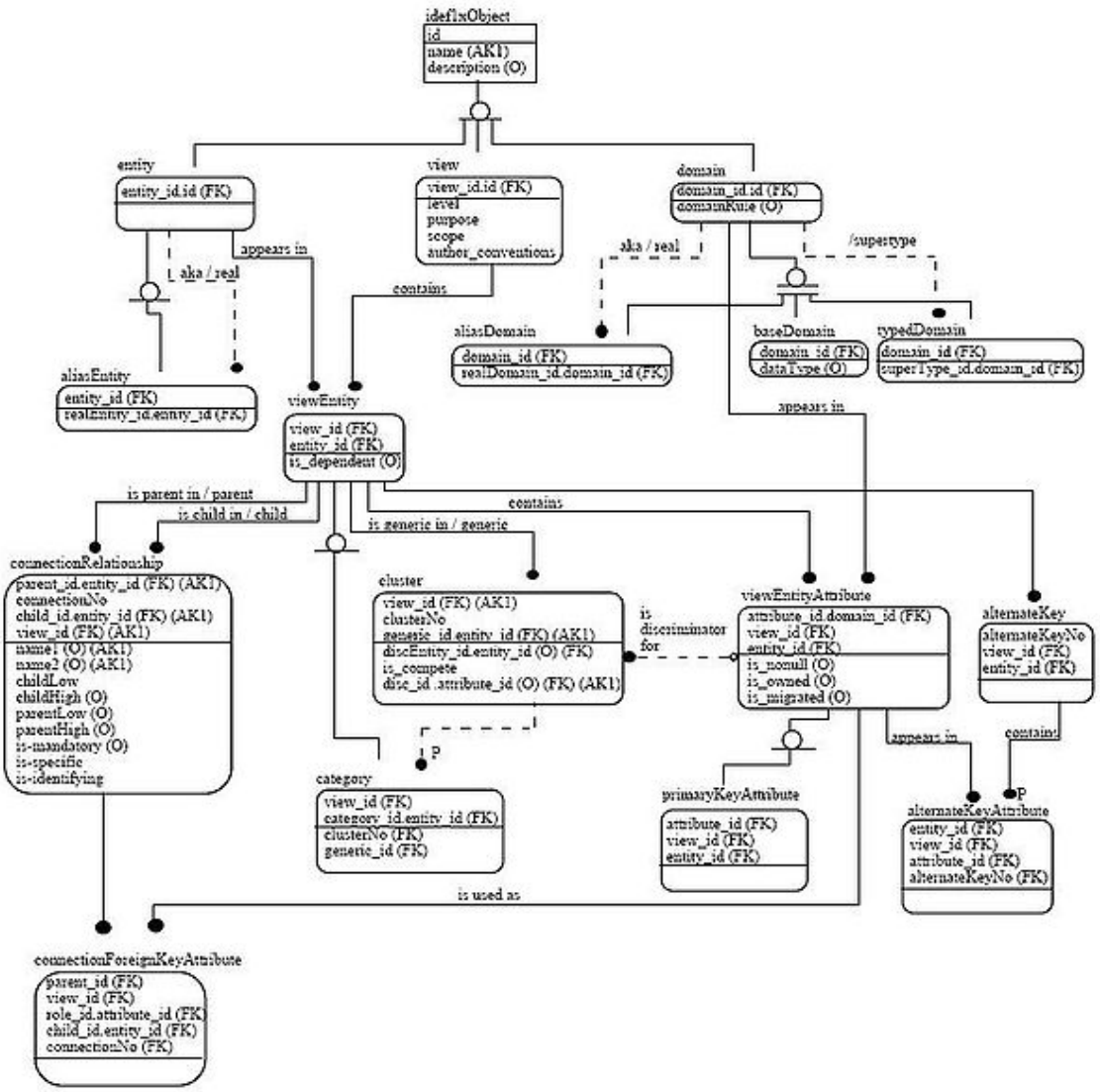
Data models represent information areas of interest. While there are many ways to create data models, according to Len Silverston (1997) only two modeling methodologies stand out, top-down and bottom-up:

- Bottom-up models are often the result of a reengineering effort. They usually start with existing data structures forms, fields on application screens, or reports. These models are usually physical, application-specific, and incomplete from an enterprise perspective. They may not promote data sharing, especially if they are built without reference to other parts of the organization.
- Top-down logical data models, on the other hand, are created in an abstract way by getting information from people who know the subject area. A system may not

implement all the entities in a logical model, but the model serves as a reference point or template.

Sometimes models are created in a mixture of the two methods: by considering the data needs and structure of an application and by consistently referencing a subject-area model. Unfortunately, in many environments the distinction between a logical data model and a physical data model is blurred. In addition, some CASE tools don't make a distinction between logical and physical data models.

Entity relationship diagrams



Example of a IDEF1X Entity relationship diagrams used to model IDEF1X itself. The name of the view is mm. The domain hierarchy and constraints are also given. The constraints are expressed as sentences in the formal theory of the meta model.

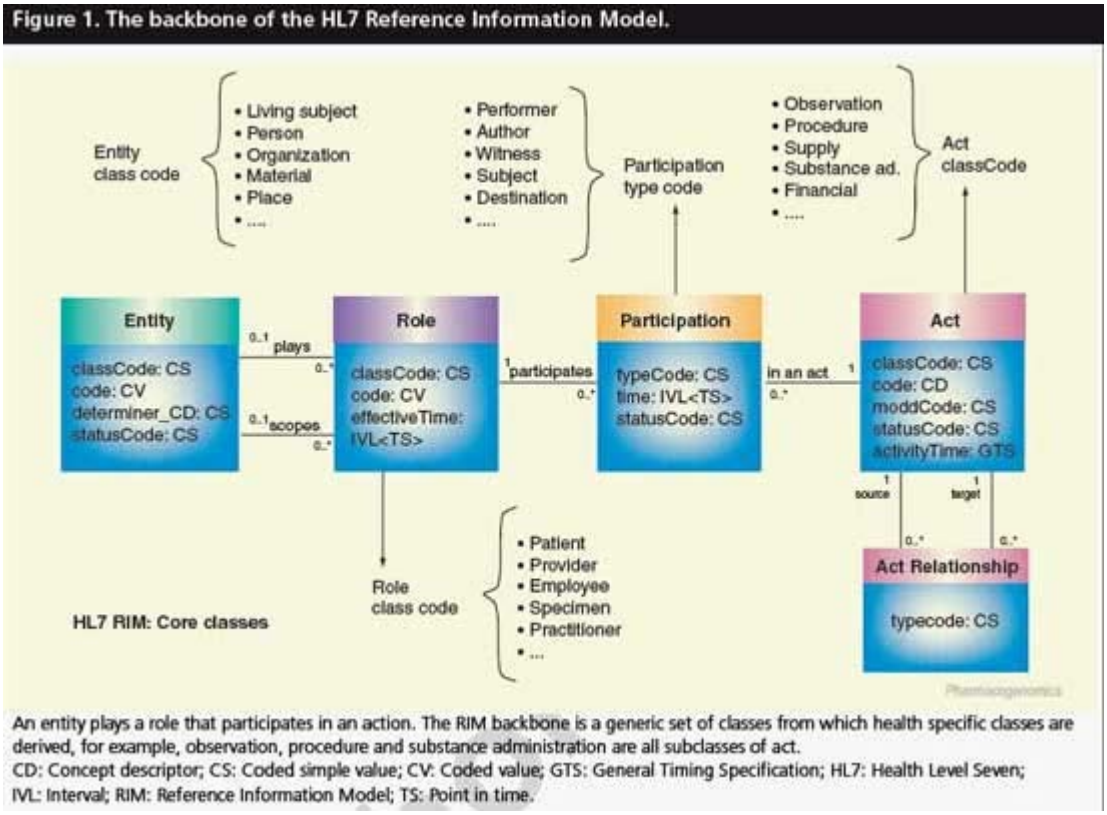
There are several notations for data modeling. The actual model is frequently called "Entity relationship model", because it depicts data in terms of the entities and relationships described in the data. An entity-relationship model (ERM) is an abstract conceptual representation of structured data. Entity-relationship modeling is a relational schema database modeling method, used in software engineering to produce a type of conceptual data model (or semantic data model) of a system, often a relational database, and its requirements in a top-down fashion.

These models are being used in the first stage of information system design during the requirements analysis to describe information needs or the type of information that is to be stored in a database. The data modeling technique can be used to describe any ontology (i.e. an overview and classifications of used terms and their relationships) for a certain universe of discourse i.e. area of interest.

Several techniques have been developed for the design of data models. While these methodologies guide data modelers in their work, two different people using the same methodology will often come up with very different results. Most notable are:

- Bachman diagrams
- Barker's Notation
- Chen's Notation
- Data Vault Modeling
- Extended Backus–Naur form
- IDEF1X
- Object-relational mapping
- Object Role Modeling
- Relational Model

Generic data modeling



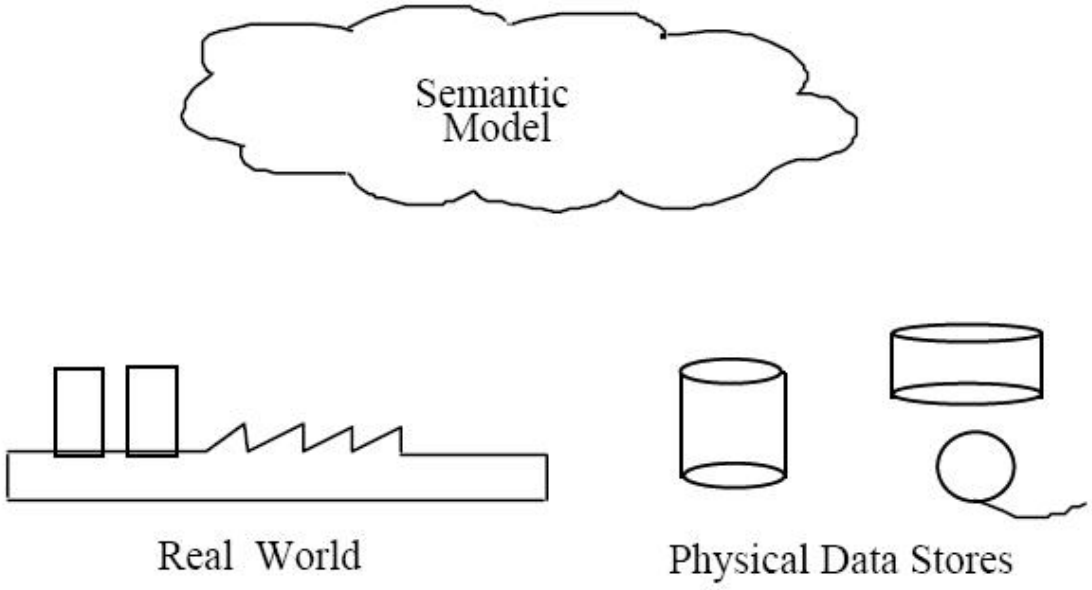
Example of a Generic data model.

Generic data models are generalizations of conventional data models. They define standardized general relation types, together with the kinds of things that may be related by such a relation type. The definition of generic data model is similar to the definition of a natural language. For example, a generic data model may define relation types such as a 'classification relation', being a binary relation between an individual thing and a kind of thing (a class) and a 'part-whole relation', being a binary relation between two things, one with the role of part, the other with the role of whole, regardless the kind of things that are related.

Given an extensible list of classes, this allows the classification of any individual thing and to specify part-whole relations for any individual object. By standardization of an extensible list of relation types, a generic data model enables the expression of an unlimited number of kinds of facts and will approach the capabilities of natural languages. Conventional data models, on the other hand, have a fixed and limited domain scope, because the instantiation (usage) of such a model only allows expressions of kinds of facts that are predefined in the model.

Semantic data modeling

The logical data structure of a DBMS, whether hierarchical, network, or relational, cannot totally satisfy the requirements for a conceptual definition of data because it is limited in scope and biased toward the implementation strategy employed by the DBMS.



Semantic data models.

Therefore, the need to define data from a conceptual view has led to the development of semantic data modeling techniques. That is, techniques to define the meaning of data within the context of its interrelationships with other data. As illustrated in the figure the real world, in terms of resources, ideas, events, etc., are symbolically defined within physical data stores. A semantic data model is an abstraction which defines how the stored symbols relate to the real world. Thus, the model must be a true representation of the real world.

A semantic data model can be used to serve many purposes, such as:.

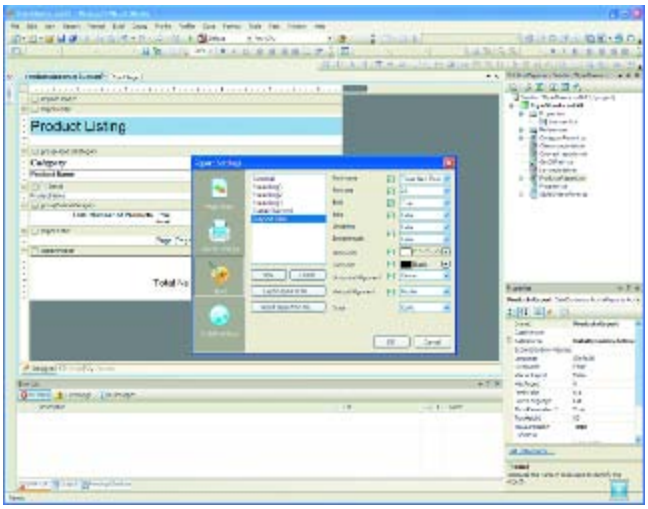
- planning of data resources
- building of shareable databases
- evaluation of vendor software
- integration of existing databases

The overall goal of semantic data models is to capture more meaning of data by integrating relational concepts with more powerful abstraction concepts known from the Artificial Intelligence field. The idea is to provide high level modeling primitives as integral part of a data model in order to facilitate the representation of real world situations.

Chapter 10

ActiveReports

ActiveReports



Developer(s)	Data Dynamics, now GrapeCity
Platform	Windows Forms / ASP.NET / .NET / Microsoft SQL Server
Type	Business Intelligence, Reporting
License	proprietary

ActiveReports (version 6 is the latest) is a .NET reporting tool used by developers of WinForms and ASP.NET applications. It was originally developed by Data Dynamics, which was then acquired by GrapeCity. ActiveReports is a set of components and tools that facilitates the production of reports to display data in documents and web-based formats. It is written in managed C# and allows Visual Studio programmers to leverage their knowledge of C# or Visual Basic.NET when programming with ActiveReports.

Among the components included with ActiveReports are exports to file formats such as PDF, Excel, RTF, and TIFF. The main component is a Visual Studio report designer with banded sections and an API that developers use to create customized reports from a

variety of data sources. ActiveReports also includes a Windows Viewer control with a customizable toolbar that supports split and multi-page views, has a Table of Contents pane with a new Thumbnail view tab, and can perform text searches of reports.

The Professional Edition of ActiveReports adds to the Standard Edition tools an End-User Report Designer control that developers use to host the report designer in their own Microsoft Windows applications to let end users create and modify reports. It also includes a server-side Web viewer with Flash, PDF, and HTML viewer types; ASP.NET HTTP Handlers that export reports to HTML or PDF format without custom code; and PDF security features like time stamping and digital signatures.

The ActiveReports Professional Edition is included in the ActiveReports Reporting and BI Suite that also contains two additional products - Data Dynamics Reports and ActiveAnalysis.

Features New to Version 6

Latest Service Releases

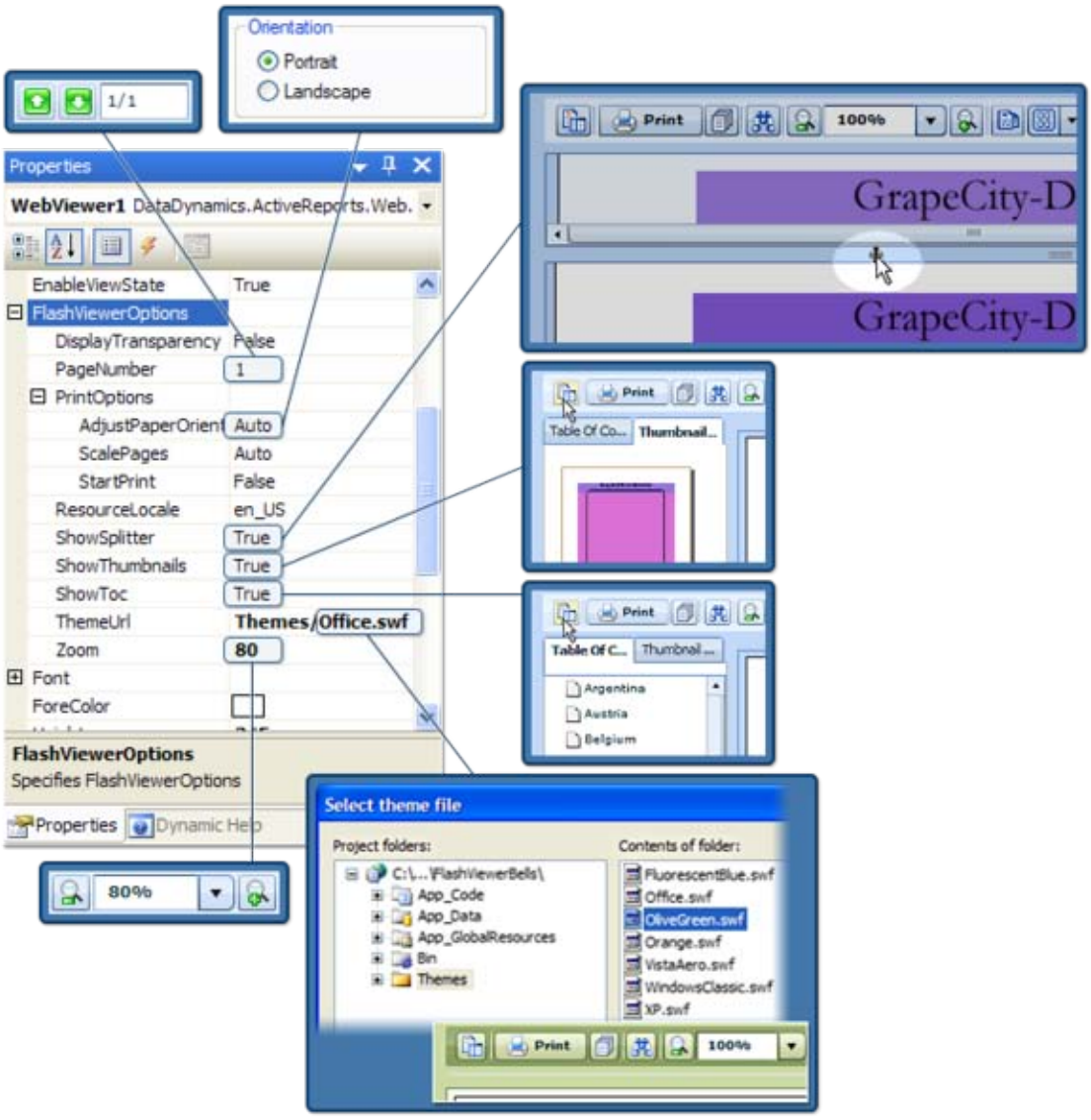
- Windows Azure reporting in full trust mode
- ASP.NET medium trust support
- Support for Visual Studio 2010
- Support for .NET Framework 4.0 Client profile
- Utility to convert from Crystal Reports (2005 and 2008) to ActiveReports

Standard Edition

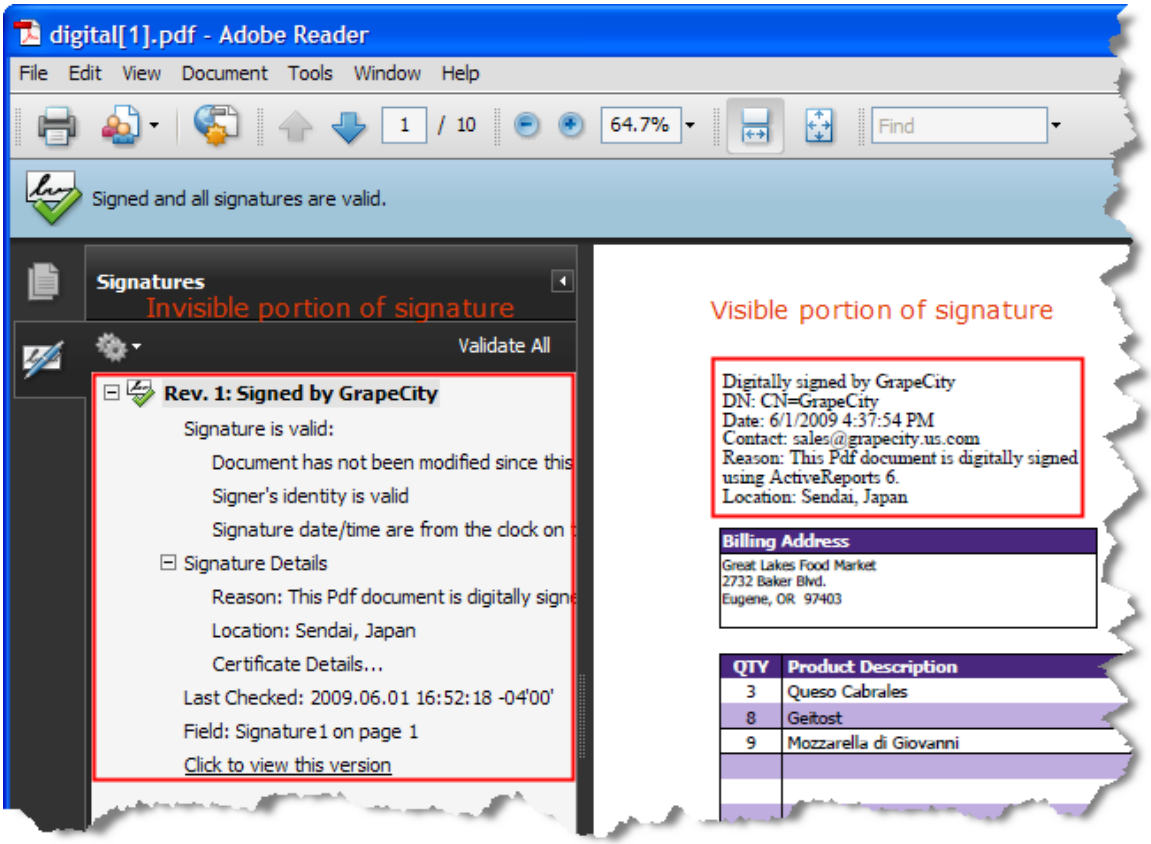
- Cross-section box and line controls that span report sections
- Quality and performance improvements
- New Reduced Space Symbology (RSS) barcode styles
- External style sheets
- Designer snap lines
- Granular control over text boxes and labels
- 64 bit support

Professional Edition

- Includes all Standard Edition new features
- Secure PDF generation with digital signatures and timestamps
- Redistributable help for the End User Report Designer
- Adobe Flash report viewer (But don't work with flash player 10.1)
- Toolstrip controls in End User Report Designer



FlashViewer options in the Properties grid with screenshots of the results.



Screenshot of visible and invisible portions of a PDF digital signature.

Features New to Version 3

ActiveReports Designer

- Design-time preview tab
- Component tray for design-time work with .NET data providers
- Report Explorer support for parameters and calculated fields
- Design-time support for custom parameters
- Design-time unbound fields

Windows Forms Viewer

- Users can add annotations
- Linked reports open in tabbed pages
- Web links open within the viewer

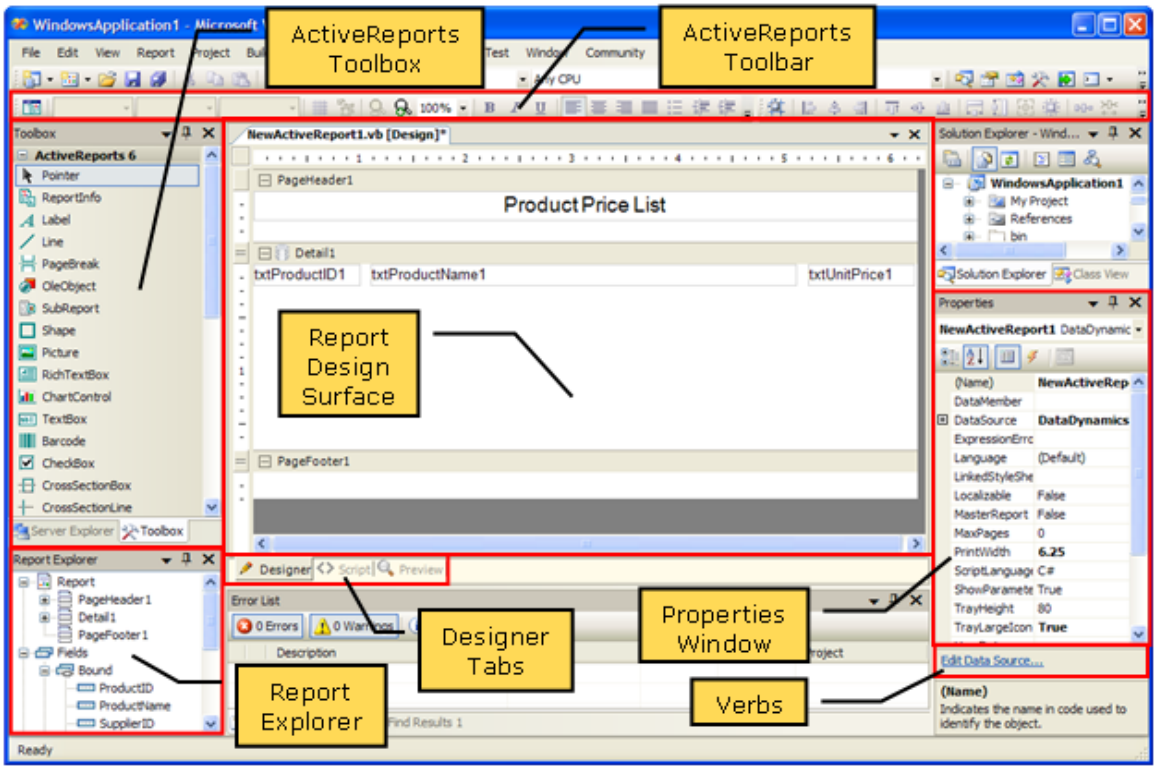
Reporting Engine

- Chart control supports seven new chart types
  - Funnel

    - Pyramid
    - Gantt
    - Kagi
    - Point and figure
    - Renko
    - Three line break
- ReportInfo control for page N of M or report run dates

Features

A number of the chart types supported in the Chart control.



The ActiveReports designer as it appears in Visual Studio, with important areas labeled.

Standard Edition

Report Designer

- Integration with the Visual Studio environment (version 6 supports Visual Studio 2005 and 2008)
- Familiar user interface
  - banded sections
  - properties accessible in the Visual Studio Properties grid
- C# and Visual Basic .NET code behind reports
- 3rd party control and user control support
- Fully exposed object model
- Dynamic report building or modification with report events
- Chart and barcode controls
- Report creation API with complete run-time access to report objects and members

Data Access

- Supports OleDB, SQL Server, and XML Datasources
- Binds to ADO.NET DataReaders, DataTables, DataViews, and DataSets
- Binds to any RowCollection, and any class that implements the IList interface

- DataSource property can be modified at run time to create ad hoc (dynamic) reports
- Flexible unbound data

Report Viewer

- Created with managed C# code
- Supports small deployment assembly, suitable for use on the Internet
- Provides tabs for contents, bookmarks and thumbnail views
- Allows hyperlinking
- Provides export filters for RTF, PDF, EXCEL, HTML, TIFF, and text
- Customizable toolbar

Other

- Report Wizard
- Microsoft Access Report Import Wizard
- Provides HTML, PDF, Excel, RTF, TIFF, and Text exports

Deployment

- Reports compile into applications for speed and security
- Reports can be kept separate from the application in report XML format (RPX) for ease of updating
- Reporting engine is a single, managed, strongnamed assembly
- Assemblies can be distributed using XCopy or the global assembly cache (GAC)
- Per-developer licensing
- Royalty-free distribution

Professional Edition

Includes all of the Standard Edition features, and adds the following:

End-User Report Designer

- Allows developers to host the designer in Windows Forms applications
- Provides end-user report editing capabilities
- Provides easy access for saving and loading report layouts
- Allows developers to monitor and control the end user's design environment
- Allows developers to customize the designer to the needs of end users

ASP.NET

- WebViewer control with Flash, HTML and PDF viewer types can be used on ASP.NET pages
- Allows quick viewing of ActiveReports on the web

- Provides printing capability with the Flash viewer and Acrobat Reader
- Provides an RPX HTTP Handler so that reports on the web server are available via hyperlink and are run and displayed in HTML or PDF
- Provides a Compiled Report HTTP Handler so that reports compiled in an assembly on the server are available via hyperlink

Supported Document Formats

- Adobe Portable Document Format (*.pdf);
- HTML Web Page (*.html);
- MHTML Single-File Web Page (*.mht);
- Rich Text Format (*.rtf);
- Microsoft Excel Workbook (*.xls);
- Plain Text (*.txt);
- Comma-Separated Values (*.csv);
- Image formats: (*.bmp, *.jpeg, *.gif, *.tiff, *.png, *.emf).

Included Report Items

- SubReport
- Chart
- Label
- Line
- Picture Box
- Check Box
- Page Break
- Report Information
- Rich Text Box
- Shape
- Bar Code
- OLE Object
- 3rd party controls (charts, grids, calendars, etc.)

Chapter 11

FarPoint Spread and Hildon

FarPoint Spread

FarPoint Spread

SPREAD

Developer(s)	FarPoint Technologies, now GrapeCity
Initial release	1991
Operating system	Microsoft Windows
Available in	English, Japanese
Type	Business Intelligence Spreadsheets Reporting
License	GrapeCity EULA

FarPoint Spread from GrapeCity is a suite of Microsoft Excel-compatible spreadsheet components available for .NET, COM, and Microsoft BizTalk Server. Software developers use the components to embed Microsoft Excel-compatible spreadsheet features into their applications, such as importing and exporting Microsoft Excel files, displaying, modifying, analyzing, and visualizing data. Spread components handle spreadsheet data at the cell, row, column, or worksheet level.

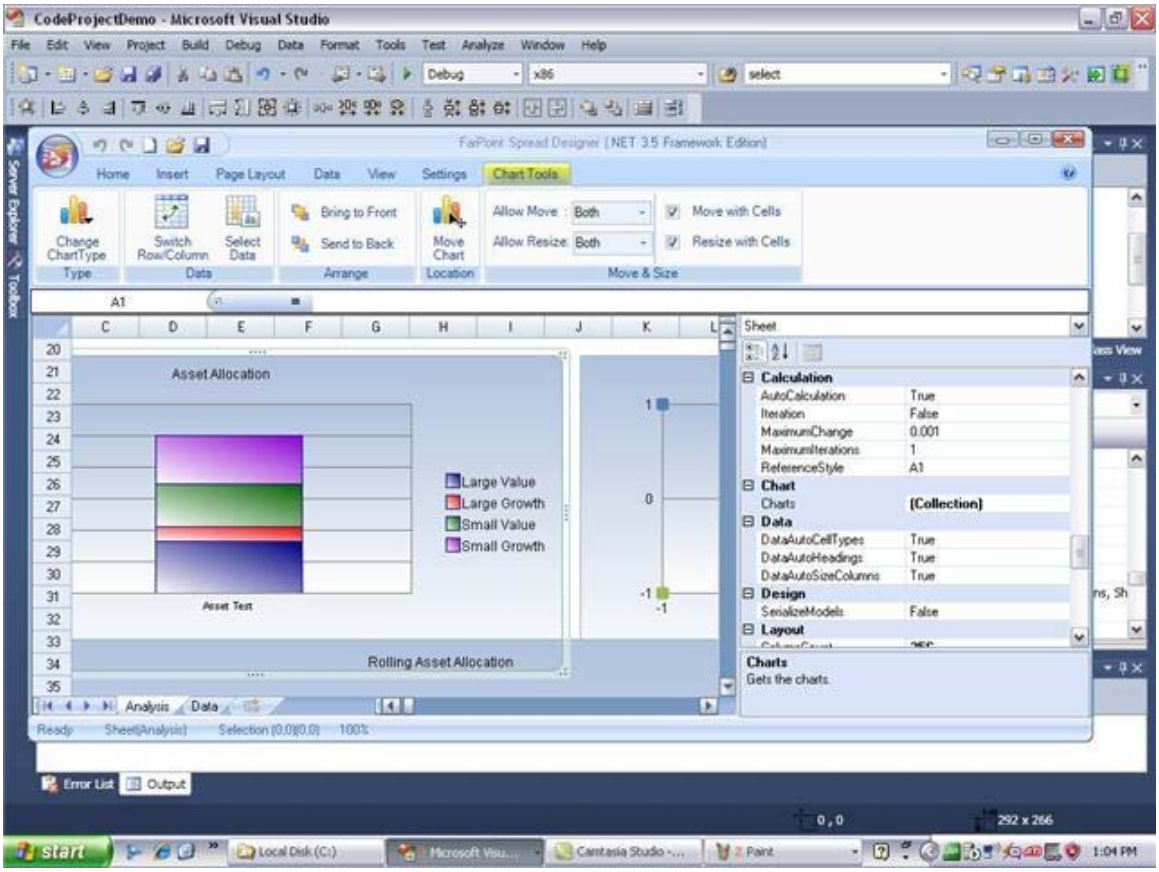
History

- **1991** Spread released as a DLL control as the initial product offering from FarPoint Technologies, Inc.
- **1990s**
  - Spread VBX released.
  - Spread ActiveX released.

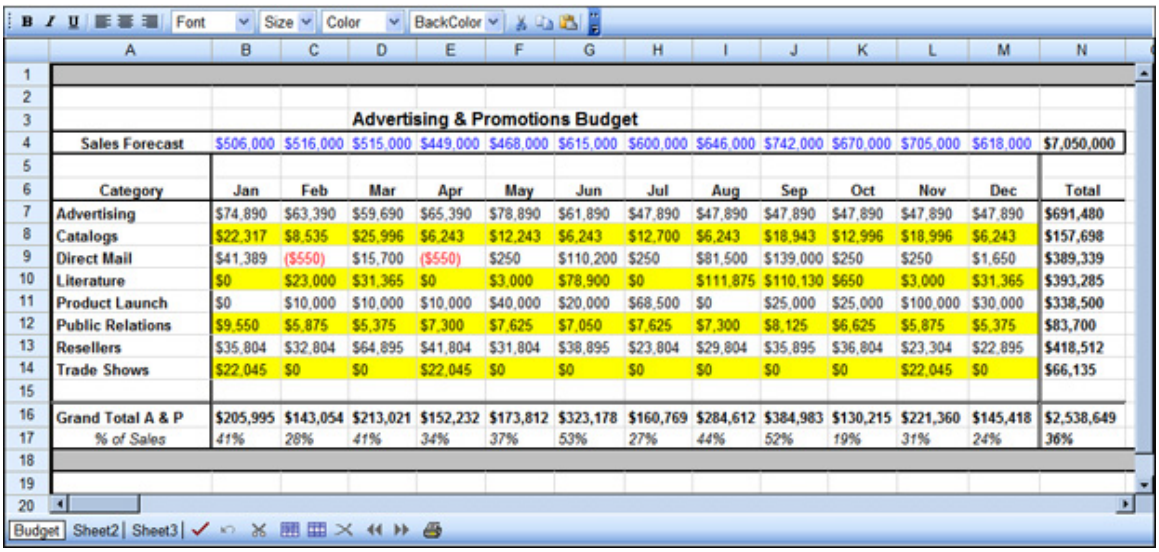
    - These components are now known as Spread COM.
- **2003** Spread for Windows Forms released as a completely new managed C# version prompted by the launch of Visual Studio .NET.
- **2003** Spread for Web Forms (now Spread for ASP.NET) released.
- **2006** Spread for BizTalk released.
- **2009** FarPoint Technologies acquired by GrapeCity.

Versions

- Spread for Windows Forms: 5.0
- Spread for Web Forms: 5.0
- Spread COM: 8.0
- Spread for BizTalk: 3.0

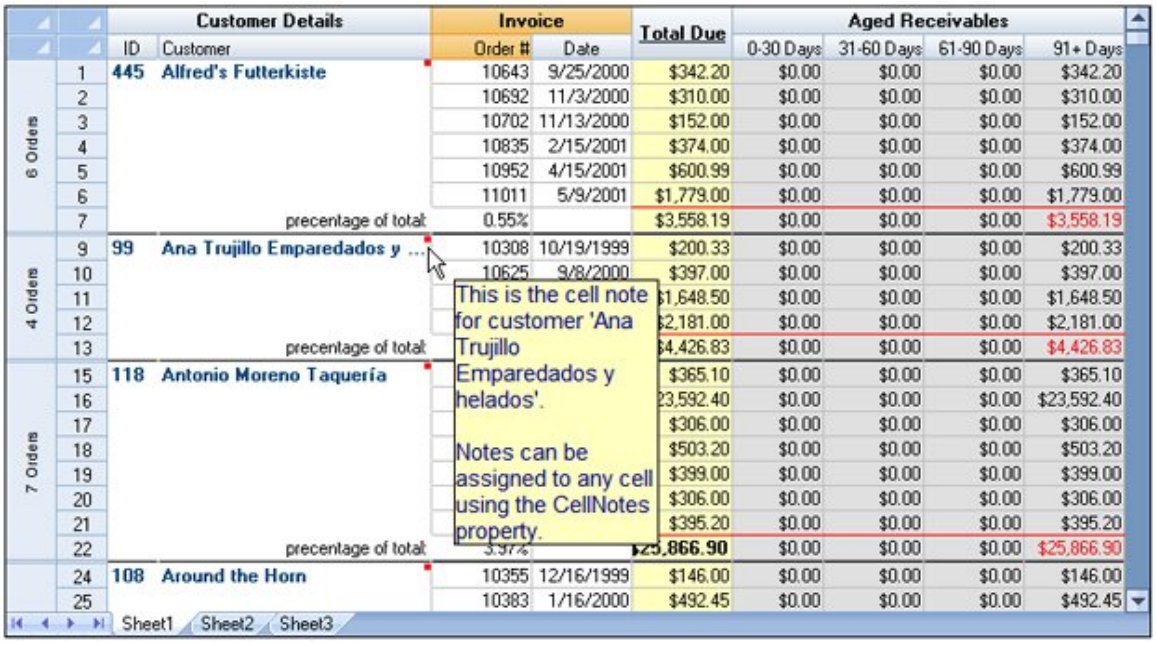


Screeshot of FarPoint Spread for Windows Forms version 5 in Visual Studio 2010.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N
3						Advertising & Promotions Budget								
4	Sales Forecast	$506,000	$516,000	$515,000	$449,000	$468,000	$615,000	$600,000	$646,000	$742,000	$670,000	$705,000	$618,000	$7,050,000
6	Category	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Total
7	Advertising	$74,890	$63,390	$59,690	$65,390	$78,890	$61,890	$47,890	$47,890	$47,890	$47,890	$47,890	$47,890	$691,480
8	Catalogs	$22,317	$8,535	$25,996	$6,243	$12,243	$6,243	$12,700	$6,243	$18,943	$12,996	$18,996	$6,243	$157,698
9	Direct Mail	$41,389	($550)	$15,700	($550)	$250	$110,200	$250	$81,500	$139,000	$250	$250	$1,650	$389,339
10	Literature	$0	$23,000	$31,365	$0	$3,000	$78,900	$0	$111,875	$110,130	$650	$3,000	$31,365	$393,285
11	Product Launch	$0	$10,000	$10,000	$10,000	$40,000	$20,000	$68,500	$0	$25,000	$25,000	$100,000	$30,000	$338,500
12	Public Relations	$9,550	$5,875	$5,375	$7,300	$7,625	$7,050	$7,625	$7,300	$8,125	$6,625	$5,875	$5,375	$83,700
13	Resellers	$35,804	$32,804	$64,895	$41,804	$31,804	$38,895	$23,804	$29,804	$35,895	$36,804	$23,304	$22,895	$418,512
14	Trade Shows	$22,045	$0	$0	$22,045	$0	$0	$0	$0	$0	$0	$22,045	$0	$66,135
16	Grand Total A & P	$205,995	$143,054	$213,021	$152,232	$173,812	$323,178	$160,769	$284,612	$384,983	$130,215	$221,360	$145,418	$2,538,649
17	% of Sales	41%	28%	41%	34%	37%	53%	27%	44%	52%	19%	31%	24%	36%

Screenshot of FarPoint Spread for Web Forms (ASP.NET).



		Customer Details		Invoice		Total Due	Aged Receivables			
		ID	Customer	Order #	Date		0-30 Days	31-60 Days	61-90 Days	91+ Days
6 Orders	1	445	Alfred's Futterkiste	10643	9/25/2000	$342.20	$0.00	$0.00	$0.00	$342.20
	2			10692	11/3/2000	$310.00	$0.00	$0.00	$0.00	$310.00
	3			10702	11/13/2000	$152.00	$0.00	$0.00	$0.00	$152.00
	4			10835	2/15/2001	$374.00	$0.00	$0.00	$0.00	$374.00
	5			10952	4/15/2001	$600.99	$0.00	$0.00	$0.00	$600.99
	6			11011	5/9/2001	$1,779.00	$0.00	$0.00	$0.00	$1,779.00
	7		precentage of total:	0.55%		$3,558.19	$0.00	$0.00	$0.00	$3,558.19
4 Orders	9	99	Ana Trujillo Emparedados y ...	10308	10/19/1999	$200.33	$0.00	$0.00	$0.00	$200.33
	10			10625	9/8/2000	$397.00	$0.00	$0.00	$0.00	$397.00
	11					$1,648.50	$0.00	$0.00	$0.00	$1,648.50
	12					$2,181.00	$0.00	$0.00	$0.00	$2,181.00
	13		precentage of total:			$4,426.83	$0.00	$0.00	$0.00	$4,426.83
7 Orders	15	118	Antonio Moreno Taquería			$365.10	$0.00	$0.00	$0.00	$365.10
	16					23,592.40	$0.00	$0.00	$0.00	$23,592.40
	17					$306.00	$0.00	$0.00	$0.00	$306.00
	18					$503.20	$0.00	$0.00	$0.00	$503.20
	19					$399.00	$0.00	$0.00	$0.00	$399.00
	20					$306.00	$0.00	$0.00	$0.00	$306.00
	21					$395.20	$0.00	$0.00	$0.00	$395.20
	22		precentage of total:	3.37%		$25,866.90	$0.00	$0.00	$0.00	$25,866.90
	24	108	Around the Horn	10355	12/16/1999	$146.00	$0.00	$0.00	$0.00	$146.00
	25			10383	1/16/2000	$492.45	$0.00	$0.00	$0.00	$492.45

This is the cell note for customer 'Ana Trujillo Emparedados y helados'.

Notes can be assigned to any cell using the CellNotes property.

Sheet1 Sheet2 Sheet3

Screeshot of FarPoint Spread COM.

Spread for Windows Forms

FarPoint Spread for Windows Forms is a Microsoft Excel-compatible spreadsheet component for Windows Forms applications developed using Microsoft Visual Studio and the .NET Framework. Developers use it to add grids and spreadsheets to their applications, and to bind them to data sources. In version 4.0, new cell types were added to display barcodes and fractions, and exports for XML and PDF were added.

Spread for ASP.NET

FarPoint Spread for ASP.NET is a Microsoft Excel-compatible spreadsheet component for ASP.NET applications. Developers use it to add grids and spreadsheets to their applications, and to bind them to data sources.

Spread for COM

FarPoint Spread 8 COM allows COM and ActiveX applications to incorporate spreadsheet features. In the 1997 book *Visual Basic 5 for Windows for Dummies*, Wally Wang lists an early version of Spread COM in Chapter 35: The Ten Most Useful Visual Basic Add-On Programs.

Spread for BizTalk

FarPoint Spread for BizTalk Server allows developers to integrate Microsoft Excel documents into Microsoft BizTalk applications. Spread for BizTalk Server includes two components:

- **Spreadsheet Pipeline Disassembler** - Parses data from Microsoft Excel (XLS and Excel 2007 XML, CSV, TXT) documents into XML data for processing through Microsoft BizTalk Server receive pipelines.
- **Spreadsheet Pipeline Assembler** - Assembles data from Microsoft BizTalk applications into Microsoft Excel (XLS or Excel 2007 XML) or PDF documents for transport through Microsoft BizTalk Server send pipelines.

Developers find it a useful tool for organizations with Microsoft BizTalk Server Enterprise Application Integration. Prior to this release, BizTalk users wanting to use Excel data had to manually open the files and copy and paste data between the two applications.

Features

These features are common to all versions.

- Predefined cell types, including:
  - currency
  - date time
  - number
  - percent
  - regular expression
  - button
  - check box
  - combo box
  - hyperlink

    - image
- Formula support, including:
    - cross-sheet referencing
    - over 300 built-in functions
- Import and export:
    - import to Microsoft Excel-compatible files
    - export to Microsoft Excel-compatible files
    - export to HTML files
    - export to XML files
- Design-time spreadsheet designer
- Data-binding with customizable options
- Hierarchical data views, with parent rows and child views
- Grouping of rows or columns
- Sorting by row or column on multiple keys
- Cell spanning
- Multiple row and column headers
- Bound and unbound modes

Version-Specific Features

Spread for Windows Forms

- Support for Microsoft Visual Studio 2010
- Support for Windows Azure AppFabric
- Integrated chart control
- Custom cell types
- Cell notes
- Child controls
- Splitter bars
- Built-in and custom skins and styles
- PDF export
- Microsoft Excel 2007 XML Support (Office Open XML, XLSX)
- Floating Formula Bar
- Range Selection for Formula
- Automatic Completion (type ahead)

Spread for ASP.NET

- Support for Microsoft Visual Studio 2010
- Support for Windows Azure AppFabric
- Integrated chart control
- AJAX-enabled
- Support for Open Document Format (ODF) files
- Multiple edits on multiple rows without server round trips
- Client-side column and row resizing
- Load on demand, which loads data from the server as needed for viewing

- Native Microsoft Excel import and export
- In-cell editing
- Multiple edits on multiple rows without server round trips
- Client-side column and row resizing
- Multiple sheets
- Searching
- Filtering
- Validations
- Cell spans
- PDF export

Spread COM

- Custom cell types
- Cell notes
- Virtual mode for data loading
- Unicode support
- Customizable printing
- Text tips
- Import and export:
    - Microsoft Excel 97
    - Excel 2000
    - Excel 2007 (requires the .NET Framework)
- Enhanced printing
- 64 bit DLL

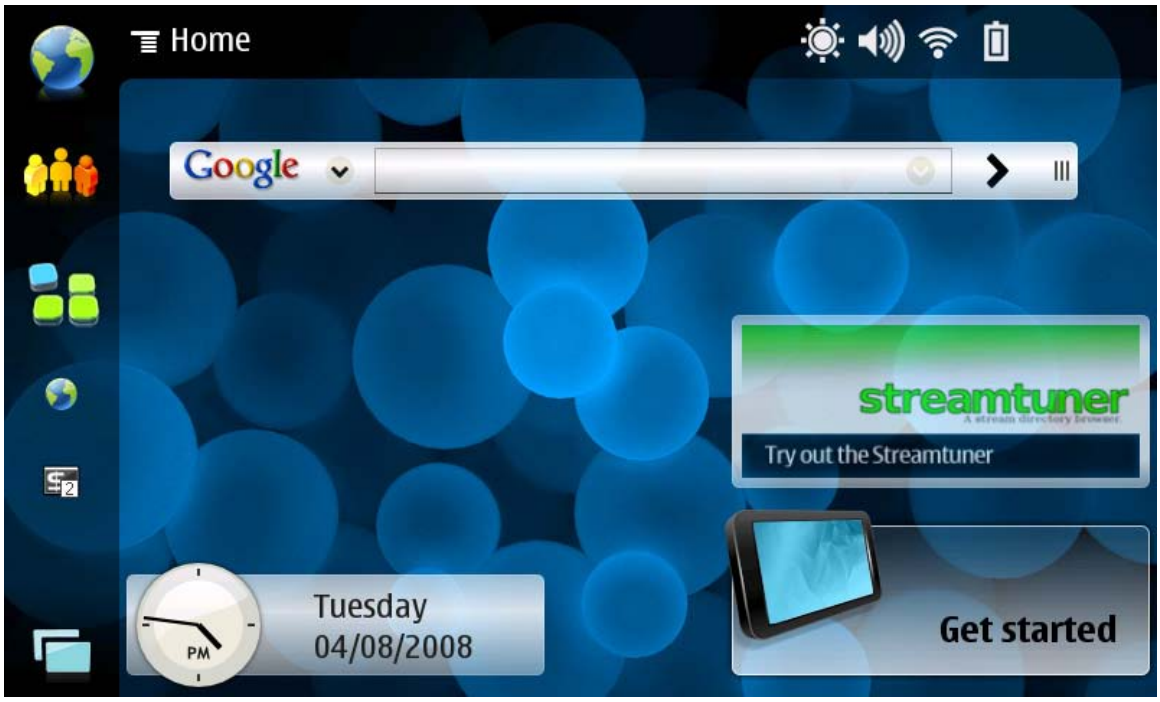
Spread for BizTalk

- Integration of Microsoft Excel data into Microsoft BizTalk applications
- Design-time spreadsheet schema wizard and spreadsheet format designer

Supported document formats

- Adobe Portable Document Format PDF (*.pdf)
- HTML Web Page (*.html)
- Microsoft Excel Workbook (*.xls)
- Plain Text (*.txt)
- Comma-Separated Values (*.csv)
- Open Document Format (Spread for ASP.NET)

Hildon



Hildon in Maemo

Hildon in Ubuntu Mobile and Embedded Edition

Hildon is an application framework originally developed for mobile devices (PDAs, mobile phones, etc.) running the Linux operating system. It was developed by Nokia for the Maemo operating system and is now a part of GNOME. It focuses on providing a finger-friendly interface. It is primarily a set of GTK+ extensions that provide mobile-device–oriented functionality, but also provides a desktop environment that includes a task navigator for opening and switching between programs, a control panel for user settings, and status bar, task bar and home applets. It is standard on the Maemo platform used by the Nokia Internet Tablets.

Hildon has also been selected as the framework for Ubuntu Mobile and Embedded Edition.

Components

The Hildon framework includes components that effectively provide a desktop environment.

Hildon Application Manager

Hildon Application Manager is the Hildon graphical package manager, it uses the Debian package management tools (APT and dpkg) and provides a graphical interface for installing, updating and removing packages. It is a limited package manager, designed specifically for end-users, in that it doesn't directly offer the user access to system files and libraries. With the Diablo release of Maemo, Hildon Application Manager now supports "Seamless Software Update" (SSU), which implements a variety of features to allow system upgrades to be easily performed through it.

Hildon Control Panel

Hildon Control Panel is the user settings interface for Hildon. It provides simple access to control panels used to change system settings.

Hildon Desktop

Hildon Desktop is the primary UI component of Hildon, so makes up the bulk of what a user will see as "Hildon". It controls application launching and switching, general system control, and provides interfaces for task bar (application menu and task switcher), status bar (brightness and volume control), and home (internet radio and web search) applets.

Hildon Library

The Hildon library, originally developed by Nokia but since Maemo 5, developed by Igalia and Lanedo (who developed MaemoGTK+, the Maemo version of Gtk). It is a set of mobile specific GTK+ widgets for applications in Maemo. Up to Maemo 4, these widgets were designed for stylus usage. However, in Maemo 5, most widgets were

deprecated and new widgets for direct finger manipulation were introduced, including a kinetic panning container.

Chapter 12

JBoss Tools and XLeratorDB

JBoss Tools

JBoss Tools	
Developer(s)	JBoss, a division of Red Hat
Stable release	3.1.0.GA / March 10, 2010; 9 months ago
Written in	Java
Operating system	Cross-platform
Type	Software development
License	Various Licenses, but predominantly EPL and LGPL

JBoss Tools is a set of Eclipse plugins and features designed to help JBoss and J2EE developers develop their applications quickly and painlessly. It is an umbrella project for the JBoss developed plugins that will make it into JBoss Developer Studio.

Modules

The JBoss Tools modules are:

- **Visual Page Editor** The visual editor contributed by Exadel provides visual editing support for HTML and JSF (JSP and Facelets) pages. VPE also includes visual support for specific JSF component libraries including JBoss RichFaces.
- **Seam Tools**. Includes support for seam-gen, RichFaces VE integration, Seam related code completion and refactoring and more.
- **Hibernate Tools**. Supporting mapping files, annotations and JPA with reverse engineering, code completion, project wizards, refactoring, interactive HQL/JPA-

QL/Criteria execution and more. In short a merger of Hibernate Tools and Exadel ORM features.

- **JBoss AS Tools**. Easy start, stop and debug of JBoss AS 4+ servers from within Eclipse. Also includes features for efficient packaging and deployment of any type of Eclipse project.
- **Drools IDE**. Rules file editing, Rete View, working memory debugging/inspection and more.
- **jBPM Tools**. jBPM workflow editing, deployment and more.
- **JBossWS Tools**. Inspecting, invoking, developing and functional/load/compliance testing of web services over HTTP, base tooling provided by soapUI with the addition of JBossWS specific features/support.
- **JBoss ESB Tools**. The structured xml editor for the jboss-esb.xml file used in JBoss ESB.
- **Birt Tools**. Hibernate and Seam extensions for Eclipse BIRT.
- **Portal Tools**. JBoss Tools supports the JSR-168 Portlet Specification (Portlet 1.0), JSR-286 Portlet Specification (Portlet 2.0) and works with PortletBridge for supporting Portlets in JSF/Seam applications. To enable these features, you need to add the JBoss Portlet facet to a new or an existing web project.
- **Core/General Tools**. To reduce the UI clutter the most of the "configure project" menu items are moved into the Configure menu introduced in Eclipse 3.5 instead of always have a static JBoss Tools menu entry show up even on non-JBoss Tools related projects.
- **Smooks Tools**. The editor for Smooks configuration files.
- **JBoss ESB Tools**. The ESB project Wizard which creates a project that can be deployed as an .esb archive to a JBoss AS based server with JBoss ESB installed.
- **JMX Tools**. JMX Tools allows you to setup multiple JMX connections and provides view for exploring the JMX tree and execute operations directly from Eclipse. The JMX Tools replaces the JMX node we previously had in the JBoss Server View.
- **JST/JSF Tools**. RichFaces Support, Code Assists, Web XML/JSP/XHTML Editors, CSS Style Editing, web.xml validation, Faceleted taglib in *taglib.xml is supported with XSD schema location.
- **Project Examples**. The experimental feature called Project Example wizard that is intended to allow users to download example projects from a remote site and have it working out of the box.
- **ESB Tools**. There is a ESB project Wizard which creates a project that can be deployed as an .esb archive to a JBoss AS based server with JBoss ESB installed.
- **AS/Project Archives Tools** . Sometimes though you are working where you would like the projects to be deployed compressed. This is now possible to enable in the server editor. If you enable this all projects that are deployed to that server will be compressed instead of in an exploded folder.
- **Maven Tools**. The optional integration with m2eclipse to provide Maven support for projects created by JBoss Tools and to some extent core WTP projects too.
- **BPEL Tools**. A BPEL Editor based on the Eclipse BPEL project has been added to JBoss Tools. This means you can now create, edit and deploy BPEL artifacts for the Riftsaw BPEL Runtime.

- **CDI (JSR-299) Tools**. Support of the Contexts and Dependency Injection annotations and it works on any Eclipse Java project, you simply need to go to the Configure menu and enable CDI.

XLeratorDB

XLeratorDB is a suite of database function libraries that enable Microsoft SQL Server to perform a wide range of additional (non-native) business intelligence and ad hoc analytics. The libraries, which are embedded and run centrally on the database, include more than 400 individual functions similar to those found in Microsoft Excel spreadsheets. The individual functions are grouped and sold as six separate libraries based on usage: finance, statistics, math, engineering, unit conversions and strings. WestClinTech, the company that developed **XLeratorDB**, claims it is "the first commercial function package add-in for Microsoft SQL Server."

Company history

WestClinTech (LLC), founded by software industry veterans Charles Flock and Joe Stampf in 2008, is located in Irvington, New York, USA. Flock was a co-founder of The Frustum Group, developer of the OPICS enterprise banking and trading platform, which was acquired by London-based Misys, PLC in 1996. Stampf joined Frustum in 1994 and with Flock remained active with the company after acquisition, helping to develop successive generations of OPICS now employed by over 150 leading financial institutions worldwide.

Following a full year of research, development and testing, WestClinTech introduced and recorded its first commercial sale of **XLeratorDB** in April 2009. In September 2009, **XLeratorDB** became available to all Federal agencies through NASA's Strategic Enterprise-Wide Procurement (SEWP-IV) program, a government-wide acquisition contract.

Technology

XLeratorDB uses Microsoft SQL CLR(Common Language Runtime) technology. SQL CLR allows managed code to be hosted by, and run in, the Microsoft SQL Server environment. SQL CLR relies on the creation, deployment and registration of .NET Framework assemblies that are physically stored in managed code dynamic-link libraries (DLL). The assemblies may contain .NET namespaces, classes, functions, and properties. Because managed code compiles to native code prior to execution, functions using SQL CLR can achieve significant performance increases versus the equivalent functions written in T-SQL in some scenarios.

XLeratorDB requires Microsoft SQL Server 2005 or SQL Server 2005 Express editions, or later (compatibility mode 90 or higher). The product installs with PERMISSION_SET=SAFE. SAFE mode, the most restrictive permission set, is accessible by all users. Code executed by an assembly with SAFE permissions cannot access external system resources such as files, the network, the internet, environment variables, or the registry.

Functions

In computer science, a function is a portion of code within a larger program which performs a specific task and is relatively independent of the remaining code. As used in database and spreadsheet applications these functions generally represent mathematical formulas widely used across a variety of fields. While this code may be user-generated, it is also embedded as a pre-written sub-routine in applications. These functions are typically identified by common nomenclature which corresponds to their underlying operations: e.g. **IRR** identifies the function which calculates Internal Rate of Return on a series of periodic cash flows.

Function uses

As sub-routines functions can be integrated and used in a variety of ways, and in a wide variety of larger, more complicated applications. Within large enterprise applications they may, for example, play an important role in defining business rules or risk management parameters, while remaining virtually undetected by end users. Within database management systems and spreadsheets, however, these kinds of functions also represent discrete sets of tools; they can be accessed directly and utilized on a stand-alone basis, or in more complex, user-defined configurations. In this context, functions can be used for business intelligence and ad hoc analysis of data in fields such as finance, statistics, engineering, math, etc.

Function types

XLeratorDB uses three kinds of functions to perform analytic operations: scalar, aggregate, and a hybrid form which WestClinTech calls ***Range Queries***. Scalar functions take a single value, perform an operation and return a single value. An example of this type of function is **LOG**, which returns the logarithm of a number to a specified base. Aggregate functions operate on a series of values but return a single, summarizing value. An example of this type of function is **AVG**, which returns the average of values in a specified group.

In **XLeratorDB** there are some functions which have characteristics of aggregate functions (operating on multiple series of values) but cannot be processed in SQL CLR using single column inputs, such as **AVG** does. For example, irregular internal rate of return (**XIRR**), a financial function, operates on a collection of cash flow values from one column, but must also apply variable period lengths from another column and an initial iterative assumption from a third, in order to return a single, summarizing value.

WestClinTech documentation notes that ***Range Queries*** specify the data to be included in the result set of the function independently of the WHERE clause associated with the T-SQL statement, by incorporating a SELECT statement into the function as a string argument; the function then traps that SELECT statement, executes it internally and processes the result.

Some **XLeratorDB** functions that employ ***Range Queries*** are: **NPV**, **XNPV**, **IRR**, **XIRR**, **MIRR**, **MULTINOMIAL**, and **SERIESSUM**. Within the application these functions are identified by a "**_q**" naming convention: e.g. **NPV_q**, **IRR_q**, etc.

Analytic functions

SQL Server functions

Microsoft SQL Server is the #3 selling database management system (DBMS), behind Oracle and IBM. (While versions of SQL Server have been on the market since 1987 , **XLeratorDB** is compatible with only the 2005 edition and later.) Like all major DBMS, SQL Server performs a variety of data mining operations by returning or arraying data in different views (also known as drill-down). In addition, SQL Server uses Transact-SQL (T-SQL) to execute four major classes of pre-defined functions in native mode. Functions operating on the DBMS offer several advantages over client layer applications like Excel: they utilize the most up-to-date data available; they can process far larger quantities of data; and, the data is not subject to exporting and transcription errors .

SQL Server 2008 includes a total of 58 functions that perform relatively basic aggregation (12), math (23) and string manipulation (23) operations useful for analytics; it includes no native functions that perform more complex operations directly related to finance, statistics or engineering.

Excel functions

Microsoft Excel, a component of Microsoft Office suite, is one of the most widely used spreadsheet applications on the market today. In addition to its inherent utility as a stand-alone desktop application, Excel overlaps and complements the functionality of DBMS in several ways: storing and arraying data in rows and columns; performing certain basic tasks such as pivot table and aggregating values; and facilitating sharing, importing and exporting of database data. Excel's chief limitation relative to a true database is capacity; Excel 2003 is limited to some 65k rows and 256 columns; Excel 2007 extends this capacity to roughly 1million rows and 16k columns. By comparison, SQL Server is able to manage over 500k terabytes of memory.

Excel offers, however, an extensive library of specialized pre-written functions which are useful for performing ad hoc analysis on database data. Excel 2007 includes over 300 of these pre-defined functions, although customized functions can also be created by users, or imported from third party developers as add-ons. Excel functions are grouped by type:

Excel Functions			
Financial	Statistical	Engineering	Math and trig
Information	Date and time	Text and data	Logical
Add-ins and automation	Lookup and reference	Cube	Database and list management

Excel business intelligence functions

Operating on the client computing layer Excel plays an important role as a business intelligence tool because it:

- performs a wide array of complex analytic functions not native to most DBMS software
- offers far greater ad hoc reporting and analytic flexibility than most enterprise software
- provides a medium for sharing and collaborating because of its ubiquity throughout the enterprise

Microsoft reinforces this positioning with Business Intelligence documentation that positions Excel in a clearly pivotal role.

XLeratorDB vs. Excel functions

While operating within the database environment, **XLeratorDB** functions utilize the same naming conventions and input formats, and in most cases, return the same calculation results as Excel functions. **XLeratorDB**, coupled with SQL Server's native capabilities, compares to Excel's function sets as follows:

	Excel 2007	**XLeratorDB + SQL Server**		
Function Type	**Total**	**Total**	**Match**	**New**	**Native**
Financial	52	63	50	13	0

Statistics	83	171	65	94	12
Math	59	76	34	19	23
Engineering	39	44	38	6	0
Conversions*	49	78	0	78	0
Strings	26	63	11	29	23